00321

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

26

FACULTAD DE CIENCIAS

USO EFECTIVO DE LA RED COMO VEHÍCULO PARA LA REQUISICIÓN MASIVA DE INFORMACIÓN

T E S I S

QUE PARA OBTENER EL TITULO DE:

A C T U A R I O

P R E S E N T A:

JORGE MARTÍN GARCIA ANAYA

DIRECTOR DE TESIS:
MAT. MATEO JAVIER SÁNCHEZ FLORES

NO
Acompañada de un CD

2003

FACULTAD DE CIENCIAS
SECCION ESCOLAR

1

PAGINACIÓN

DISCONTINUA

DRA. MARÍA DE LOURDES ESTEVA PERALTA
Jefa de la División de Estudios Profesionales de la
Facultad de Ciencias
Presente

Comunicamos a usted que hemos revisado el trabajo escrito:

"USO EFECTIVO DE LA RED COMO VEHICULO PARA LA REQUISICION MASIVA DE INFORMACION"

realizado por GARCIA ANAYA JORGE MARTIN

con número de cuenta 08507717-3 , quien cubrió los créditos de la carrera de:

ACTUARIA

Dicho trabajo cuenta con nuestro voto aprobatorio.

A t e n t a m e n t e

Director de Tesis Propietario	MAT. MATEO JAVIER SANCHEZ FLORES
Propietario	M. EN C. MARIA GUADALUPE ELENA IBARGÜENGOITIA GONZALEZ
Propietario	DRA. AMPARO LOPEZ GAONA
Suplente	MAT. SALVADOR LOPEZ MENDOZA
Suplente	DRA. HANNA OKTABA

Consejo Departamental de Matemáticas

M. EN C. JOSE ANTONIO FLORES DIAZ

FACULTAD DE CIENCIAS
CONSEJO DEPARTAMENTAL
DE
MATEMATICAS

1 - A

Uso efectivo de la red como vehículo para la requisición masiva de información

Tesis que para obtener el título de

ACTUARIO

Presenta: Jorge Martín García Anaya

Introducción

En la vida cotidiana aparecen frases como las siguientes: "hágalo por Internet", "pague desde su oficina" o "trámites desde la comodidad de su hogar o negocio", desafortunadamente es frecuente que estas frases no tengan soporte alguno o la realidad no les corresponda.

Los sistemas o aplicaciones que pretenden dar sustento a estos comentarios, no pocas veces hacen uso desordenado y poco óptimo de la red global, que llamaremos *internet*. Esto se traduce en un usuario final que mira internet en este caso más como estorbo que como ayuda.

Es común encontrar situaciones como las siguientes:

- Ofrecimientos, típicamente de instituciones de gobierno, para llevar a cabo algún trámite vía internet. La sorpresa viene cuando resulta necesario mantener en línea la captura de formularios complejos o extensos. Una desconexión resulta fatal y desalentadora.

- Uso de internet como medio de distribución de aplicaciones. Aplicaciones cuya ejecución generalmente permite preparar información que posteriormente se retroalimentará también vía internet. Existen aplicaciones que miden más de 10 megabytes y que pueden ser ejecutadas únicamente bajo cierta configuración del equipo. Se vuelve casi un martirio buscar la versión adecuada al equipo que se usará y después descargar 10 megabytes de la red.

- Al ejecutar las aplicaciones antes descritas existen serias limitaciones de validación de la información, invalidez de rangos de fechas y de valores, valores derivados redundantes, ausencia de máscaras de captura, ausencia de catálogos. Y en general poca solidez para la validación de información, que al ser enviada con deficiencias no sirve para nada.

Estos ejemplos muestran la existencia de un problema serio en el uso de internet como medio para realizar trámites o requerir información. Esta tesis se limita a la problemática que surge al usar internet como medio para requerir información de manera masiva, atendiendo procedimientos como los siguientes:

- Declaraciones patrimoniales
- Declaraciones de impuestos
- Declaraciones de ingresos.

Se busca en concreto y como lo dice el título de la tesis, lograr un "**Uso efectivo de la red como vehículo para la requisición masiva de información**".

En esta búsqueda se transitará por todas las etapas del ciclo de vida de la construcción de sistemas, se buscarán y abordarán las herramientas tecnológicas disponibles para alcanzar la meta planteada.

Algunas etapas del ciclo de desarrollo se revisarán muy brevemente debido a que el paso por cada etapa puede generar por si mismo una gran cantidad de material.

Por ejemplo, durante el diseño se hará uso de una herramienta CASE (por sus siglas en inglés "Computer Aided Software Engineering") con el lenguaje de modelado unificado UML, y se describirá brevemente UML y lo que se considera la problemática significativa durante el uso del CASE.

El capítulo I corresponde con la etapa de análisis y básicamente se ocupa del Lenguaje de Modelado Unificado UML y una herramienta CASE que lo utiliza.

El capítulo II aborda la elección del lenguaje de programación, que es Java, y sus características.

Durante el capítulo III se profundiza en seguridad con Java.

El capítulo IV presenta otro lenguaje, para el intercambio de información, que al momento de buscar portabilidad es de gran valor. Se trata de XML o eXtensible Markup Language

En el capítulo V se presentan las especificaciones de la solución propuesta, un cliente universal para la requisición masiva de información.

Finalmente se presentan las conclusiones obtenidas de este trabajo.

Capítulo I.- Modelado

A lo largo de los años el desarrollo de software se ha vuelto cada vez más complicado. El incremento en la capacidad de cómputo de las computadoras y la aparición de modernos dispositivos y tecnologías ha generado la necesidad de nuevos sistemas operativos y aplicaciones de mayor complejidad. Simultáneamente, esta complejidad ha traído consigo la demanda de nuevos métodos y herramientas que ayuden en la construcción de este software.

A nadie que trabaje o haya trabajado en el ámbito relacionado con el desarrollo de software le extraña que un proyecto sufra retrasos, o incluso deba cancelarse. La situación es caótica y actualmente, lo que sorprende es encontrar un proyecto que marche tal y como estaba previsto en los planes de trabajo.

Los culpables de esta situación, podemos buscarlos en todos los niveles de la industria del software, desde las áreas de ventas que cotizan y venden los proyectos en los tiempos que quiere el cliente sin conocer siquiera las características de los requerimientos del o los sistemas a realizar, hasta las áreas consultoras que dictaminan el tiempo de construcción, pasando por los propios clientes que regularmente no saben lo que quieren o necesitan hasta que finalizan los proyectos o éstos se encuentran en etapas muy avanzadas de su desarrollo.

Particularizando, uno de los problemas más comunes en el desarrollo de software es que la mayoría de los programadores y analistas es reacia a documentar las decisiones tomadas, ya sea por falta de tiempo o bien por pereza y cuando se hace, suele ser algo incompleto, no actualizado y poco consistente pues es común que cada miembro del equipo utilice una serie de símbolos familiares para él pero no para los demás.

Otro problema habitual, relacionado con el anterior, es la absoluta escasez de procedimientos establecidos dentro de una empresa para desarrollar software o, en su caso, supervisar a empresas consultoras que lo desarrollen. En el mejor de los casos, las empresas suelen seguir el método de desarrollo en cascada, es decir requisitos, análisis, diseño, implementación y pruebas.

Estos dos problemas y otros más surgen principalmente por la interpretación que todos hacemos de la Ingeniería del Software, que la mayoría de las veces no es ingeniería ni nada que se le parezca.

Que es UML

En este contexto el Lenguaje Unificado de Modelado (UML por sus siglas en inglés) surge como respuesta al primer problema reseñado, contar con un lenguaje estándar para escribir planos de software. Muchos han creído ver UML como solución para todos sus problemas sin saber en muchos casos de lo que se trataba en realidad.

El Lenguaje Unificado de Modelado, UML es una notación estándar para el modelado de sistemas de software, resultado de una propuesta de estandarización promovida por el consorcio OMG (Object Management Group), del cual forman parte las empresas más importantes que se dedican al desarrollo de software, en 1996.

UML representa la unificación de las notaciones de los métodos Booch, Objectory (Ivar Jacobson) y OMT (James Rumbaugh) siendo de éste su sucesor directo y compatible.

Igualmente, UML incorpora ideas de otros metodólogos entre los que podemos incluir a Peter Coad, Derek Coleman, Ward Cunningham, David Harel, Richard Helm, Ralph Johnson, Stephen Mellor, Bertrand Meyer, Jim Odell, Kenny Rubin, Sally Shlaer, John Vlissides, Paul Ward, Rebecca Wirfs-Brock y Edward Yourdon.

En septiembre de 2001 se publicó la especificación de la versión 1.4 de UML.

Es importante recalcar que, erróneamente a lo que mucha gente piensa, sólo se trata de una notación, es decir, de una serie de reglas y recomendaciones para representar modelos. En otras palabras UML no es un proceso de desarrollo, es decir, no describe los pasos sistemáticos (que tienen que ver con el segundo problema descrito con anterioridad) a seguir para desarrollar software. UML sólo permite documentar y especificar los elementos creados mediante un lenguaje común describiendo modelos.

En todos los ámbitos de la ingeniería se construyen modelos, en realidad, simplificaciones de la realidad, para comprender mejor el sistema que se va a desarrollar: los arquitectos utilizan y construyen planos (modelos) de los edificios, los diseñadores de coches preparan modelos en sistemas CAD/CAM con todos los detalles y los ingenieros de software deberían igualmente construir modelos de los sistemas de software.

Un enfoque sistemático permite construir estos modelos de una forma consistente demostrando su utilidad en sistemas de cierto tamaño. Cuando se trata de un programa de cincuenta, cien líneas, la utilidad del modelado parece discutible pero cuando se involucra a decenas de desarrolladores trabajando y compartiendo información, el uso de modelos y el proporcionar información sobre las decisiones tomadas, es vital no sólo durante el desarrollo del proyecto, sino una vez finalizado éste, cuando se requiere algún cambio en el sistema. En realidad, incluso en el proyecto más simple los desarrolladores hacen algo de modelado, si bien informalmente.

Para la construcción de modelos, hay que centrarse en los detalles relevantes mientras se ignoran los demás detalles[1], por lo cual con un único modelo no tenemos bastante. Varios modelos aportan diferentes vistas de un sistema los cuales ayudan a comprenderlo desde varios frentes. Así, UML recomienda la utilización de nueve diagramas, para representar las distintas vistas de un sistema. Estos diagramas de UML son los siguientes:

[1] Enterprise Architect for J2EE, Mark Cade and Simon Roberts, Sun Microsystems

- *Diagrama de Casos de Uso*: modela la funcionalidad del sistema agrupándola en descripciones de acciones ejecutadas por un sistema para obtener un resultado.
- *Diagrama de Clases*: muestra las clases (descripciones de objetos que comparten características comunes) que componen el sistema y cómo se relacionan entre sí.
- *Diagrama de Objetos*: muestra una serie de objetos (instancias de las clases) y sus relaciones.
- *Diagrama de Secuencia*: enfatiza la interacción entre los objetos y los mensajes que intercambian entre sí junto con el orden temporal de los mismos.
- *Diagrama de Colaboración*: igualmente, muestra la interacción entre los objetos resaltando la organización estructural de los objetos en lugar del orden de los mensajes intercambiados.
- *Diagrama de Estados*: modela el comportamiento de acuerdo con eventos.
- *Diagrama de Actividades*: simplifica el diagrama de estados modelando el comportamiento mediante flujos de actividades.
- *Diagrama de Componentes*: muestra la organización y las dependencias entre un conjunto de componentes.
- *Diagrama de Distribución*: muestra los dispositivos que se encuentran en un sistema y su distribución en el mismo.

Qué debería ofrecer una herramienta CASE para UML

Durante el diseño del sistema que se describirá en el capítulo V se utilizó como apoyo una herramienta CASE, herramienta orientada a automatizar parte del ciclo de construcción de un sistema.

Los problemas o limitaciones durante su uso fueron serios y abundantes, durante la etapa de diseño fue posible notar problemas en la aplicación del CASE Rational Rose v. 7.7.0204. En forma de inventario y con óptica de aportación se presenta lo que se considera que una herramienta CASE debería ofrecer para lograr verdadero apoyo en esta etapa, con esto no se pretende decir que una herramienta CASE no es un apoyo actualmente o que funcione mal o que esté mal diseñada, lo que se busca decir es que *existen limitaciones en ella* y la industria deberá progresar en ese sentido.

La herramienta se utilizó orientada a UML, lenguaje de modelado utilizado durante el diseño.

El CASE usado cumple en cierto grado todas las necesidades aparecidas, hecho que naturalmente ocurre con cualquier CASE del mercado. De hecho en algunas situaciones fue necesario experimentar con otras herramientas con la finalidad de resolver deficiencias importantes del CASE principal.

Este inventario puede ser de utilidad al momento de evaluar una herramienta CASE a ser utilizada para soportar UML.

Repositorio

Para un proyecto grande, es necesario un repositorio que permita a los analistas compartir los diseños de los componentes. Dos o más analistas pueden compartir los componentes de un modelo o incluso pueden colaborar en el desarrollo de uno solo si se define la "propiedad" y la autorización para compartir a un nivel apropiado.

Generalmente se construye un repositorio sobre una base de datos que proporciona las funciones para compartir información y para el control concurrente.

Al proporcionar protección de acceso y de acceso sólo para lectura, el repositorio permite a un analista tomar posesión del modelo y a la misma vez deja a otros leerlo, permitiendo así la incorporación de componentes en sus propios diseños. Importante: La herramienta debe permitir capturar sólo los componentes que desea de otro modelo, sin tener que importar la totalidad del mismo.

Otra característica importante para el repositorio es que permita la incorporación del propio código fuente de un proyecto acorde a lo modelado, naturalmente proporcionando un sistema adecuado de control de accesos concurrentes. El beneficio de este método es un mayor grado de sincronización entre el código y el modelo, un beneficio adicional es la eliminación de una segunda fuente de datos.

No se debe perder de vista que si se utiliza una base de datos para un repositorio, es necesario salvaguardar estos datos almacenados separadamente y realizar una sincronización trilateral entre el modelo, el repositorio y el código fuente en lugar de tan sólo una sincronización bilateral entre el código y el modelo.

Con las herramientas de modelado que soporten un repositorio, los cambios efectuados a cualquier componente serán propagados automáticamente a cualquier diseño que importe el componente.

Ingeniería hacia delante e inversa (round-trip)

La capacidad de realizar la ingeniería hacía adelante e ingeniería inversa al código fuente (Java, C++, VB, etc.), es un requerimiento complejo que las herramientas CASE proporcionan con diferentes grados de éxito. La exitosa combinación de ambas funciones, la ingeniería hacia adelante e inversa, se conoce como round-trip engineering.

La ingeniería hacia adelante es muy útil la primera vez que se genera el código de un modelo. Esto evita en gran parte él tener que escribir clases, atributos y métodos.

La ingeniería inversa es muy útil tanto en la transformación del código en un modelo cuando no existe ningún modelo previo, como en la sincronización de un modelo con el código al final de una iteración.

Durante un ciclo de desarrollo iterativo, una vez que un modelo ha sido actualizado como parte de la iteración, otra iteración de ingeniería hacia adelante debe permitir que el código se actualice con todas las nuevas clases, métodos o atributos añadidos al modelo.

Este paso es adoptado con menor frecuencia por los analistas debido a que muchas herramientas pueden estropear totalmente el código fuente en el proceso. El problema reside en que el código fuente contiene mucho más que el modelo; las herramientas deberán ser capaces de reconstruir el código fuente que existía antes de la nueva iteración de ingeniería hacia adelante.

Como mínimo, la herramienta de modelado deberá apoyar con éxito la ingeniería hacia adelante en primera instancia y la ingeniería inversa a lo largo del proceso. Asimismo, la herramienta no deberá tener ningún problema al aplicar la ingeniería inversa al lenguaje Java, es oportuno verificar esta función en el propio código fuente.

La documentación HTML

La herramienta de modelado de objetos deberá proporcionar la generación continua de documentación HTML para un modelo de objetos y sus componentes. La documentación HTML proporciona una visualización estática del modelo de objetos a través de un navegador por cualquier analista que utilice el modelo para uso de referencia rápida sin tener que utilizar la herramienta de modelado.

Asimismo, al producir HTML como documentación, el número de licencias requeridas para la herramienta de modelado puede verse reducido por el número de personas que necesiten acceso sólo para lectura a la información del modelo.

La documentación HTML deberá incluir una imagen (bitmap) de cada uno de los diagramas del modelo y deberá proporcionar navegación a través del modelo mediante el uso de enlaces hipertexto. El tiempo requerido para generar la documentación HTML deberá ser razonable.

Soporte completo a UML 1.3

Aunque muchas herramientas afirman soporte completo para UML 1.3, en realidad éste es un requerimiento complejo y algunas herramientas pueden no corresponder con las pretensiones anunciadas. Como mínimo, los diagramas que deberán ser incluidos son los de casos de uso (use case), de clase (class), colaboración (collaboration), secuencia (sequence), paquete (package), y estado (state).

Listas de selección para Clases y Métodos

La herramienta de modelado deberá proporcionar listas de selección en varios puntos clave:

Diagramas de Colaboración y Secuencia - La herramienta deberá permitir la asignación de un objeto a una clase proveniente de una lista conteniendo todas las clases del modelo. Deberá permitir la selección de los mensajes enviados entre objetos a través de una lista válida de métodos para el objeto/clase que ha de recibir el mensaje.

Diagrama de Clase - La herramienta deberá permitir la importación de clases provenientes de otros paquetes o modelos a través de la selección de una clase perteneciente a una lista de clases del paquete.

La función "listas de selección" contribuye en gran manera al fácil uso de la herramienta de modelado y puede considerarse como una función esencial.

El desarrollo de diagramas de secuencia (sequence diagrams) y de diagramas de colaboración (collaboration diagrams) se facilita en gran medida al ser posible el seleccionar rápidamente el mensaje que se desea enviar de un objeto a otro.

Integración con el modelado de datos

La herramienta de modelado de objetos deberá permitir la integración con funciones de modelado de datos. Existen muchas formas de proporcionar esta funcionalidad. Una de ellas consiste en que la herramienta orientada al objeto proporcione una función que permita la transformación de un modelo del objeto a DDL (Data Definition Language), que es en si el conjunto de las instrucciones de SQL requeridas para crear tablas para clases.

Otra forma es que la herramienta de objetos exporte metadatos a una herramienta de modelado de datos y esta última importe los metadatos y los utilice como base para un modelo de datos. Un conjunto de herramientas integrado y avanzado deberá permitir que los modelos de datos y de objetos se sincronicen tras cada iteración del diseño.

Creación de versiones

La herramienta de modelado deberá permitir salvaguardar las versiones de manera que cuando comiencen las nuevas iteraciones, la versión previa se encuentre disponible para su restauración o para la conservación del código existente que depende de esa versión.

La navegación del modelo

La herramienta de modelado deberá proporcionar un muy buen nivel de navegación para que el analista pueda navegar a través de todos los diagramas y clases del modelo. Un directorio o lista de selección de clases, ordenadas por nombre, es una forma de permitir que el diseñador vaya a la clase deseada dentro de un diagrama.

Para diagramas grandes, la herramienta deberá facilitar la navegación cuando se efectúe una visualización con zoom y panorámica. Asimismo, la herramienta deberá facilitar la navegación al código fuente de una clase cuando se utilice ingeniería hacia adelante e inversa.

Apoyo de impresión

La herramienta de modelado deberá proporcionar imágenes adecuadas de diagramas grandes para ser reproducidos mediante la impresión en páginas múltiples. Deberán soportarse las funciones de visualización previa a la impresión y de escala para así facilitar el ajuste del diagrama al deseado número de páginas.

La capacidad de ajuste de un diagrama en una sola página es muy importante. Desgraciadamente, se constató que muchas herramientas presentan dificultades en la ejecución de esta importante tarea.

Visualización de diagramas

La herramienta de modelado deberá proporcionar la facilidad para personalizar la visualización de una clase y sus detalles. Por ejemplo, deberá ser posible excluir todos los métodos get/set del diagrama, puesto que tienden a aglomerarlo en lugar de aclararlo.

Deberá permitirse de un modo fácil mostrar u ocultar la firma completa de los métodos, dependiendo del nivel de detalle deseado. La visibilidad de atributos y métodos (privado, protegido, público) deberá ser otra dimensión con la cual seleccionar lo que se debe mostrar u ocultar en el diagrama.

Exportación de diagramas

Una característica clave que a menudo se pasa por alto es la capacidad de exportar diagramas en un formato que pueda ser importado ya sea a un documento word, o a una página web. Los formatos gráficos más populares utilizados para la exportación son el GIF y el JPEG.

En la exportación, la herramienta deberá permitir definir la resolución y el tamaño preferidos para la gráfica que ha de producirse. Esta funcionalidad se origina por el requerimiento de algunos autores que quieren escribir un libro de UML que incluya diagramas, o que desean mostrar su trabajo en una página web.

Scripting

El scripting[2] es otra poderosa función que deberá proporcionar una herramienta de modelado. Con el scripting, el usuario experto puede crear scripts que tengan acceso directamente al modelo de objetos que radica dentro de la herramienta de modelado con el fin de crear otros archivos.

[2] Scripting es el uso o ejecución de un programa escrito con algún lenguaje de tipo script como JavaScript, HTML, XML o Perl. Por otra parte script se define como un lenguaje de programación muy corto, es decir que consiste de pocas instrucciones, con sintaxis simple.

Ejemplos de estos archivos son las hojas de cálculo para manejo de proyectos, documentación personalizada, código personalizado, reportes. Un ejemplo de código personalizado son las firmas para las clases de colección y los métodos get/set utilizados para acceder a éstas.

Para facilitar el scripting, la herramienta de modelado deberá exponer interfaces al modelo de objetos, internas a este mismo, que proporcionen acceso a los componentes del modelo de objetos en elaboración.

Por ejemplo, el editor de script deberá ser capaz de acceder la colección de clases en un diagrama de clases a través de una iteración, y entonces deberá poder acceder las propiedades de las clases a través de métodos de acceso sobre el objeto de la clase. Naturalmente, el propio lenguaje de scripting deberá estar basado en objetos; una elección obvia es el mismo lenguaje Java y otro es el lenguaje de scripting Python.

Robustez

Un objetivo esencial es tener una herramienta UML de sólida confiabilidad para prevenir que los usuarios pierdan horas potenciales de productividad cuando la herramienta deje de funcionar en medio de una sesión de diseño o que corrompa un modelo que no ha sido guardado, un ejemplo de esto es Rational Rose 7.7, pues en sesiones con más de 10 diagramas abiertos es común que deje de operar.

Nuevas versiones

Es importante contar con una herramienta de modelado que continúe siendo mejorada activamente a través de la corrección de errores, mejoras en su rendimiento y la inclusión de nuevas funciones.

Es recomendable también tener cuidado con productos que hayan sido adquiridos por compañías grandes. Muchas veces los desarrolladores originales han renunciado y es muy difícil encontrar programadores talentosos que puedan aprender un pedazo de software complicado y que además quieran mantenerlo sin haber ellos desarrollado el código original.

Funcionalidad fuera del alcance de la mayoría de herramientas CASE actuales

Ahora se citará la lista de deseos para el futuro. Un CASE que atienda esta funcionalidad deberá considerarse en el camino de la evolución y por tanto buen candidato a ser usado.

Editor Integrado

Durante el desarrollo iterativo de un modelo, es muy conveniente mirar en ventanas adyacentes el diagrama UML y el código fuente concordante. Los productos que soporten esta coordinación de perspectivas podrán añadir una dimensión adicional muy poderosa a sus características.

Aunque la herramienta de modelado no necesita ser el editor primario del diseñador, es bastante útil poder cambiar el nombre o firma de un método

directamente en el código y dejar que el cambio se refleje inmediatamente en el modelo.

A la cabeza de la lista de funciones deseables se encuentra la emulación de teclado para editores populares tales como "Emacs". Otra importante función es la del uso de colores para subrayar en el código palabras clave, comentarios, etc.

Esta función facilita la legibilidad del código. Una función esencial es la capacidad de saltar a la línea concordante del código cuando se seleccione una clase, atributo o método en un diagrama de clase. Por encima de todo, el editor debería ser rápido y fácil de utilizar.

Como variación de este tema, otra solución es permitir que la herramienta de modelado se comunique con el editor preferido del analista. Un ejemplo sería: una tecla personalizada podría permitir que la herramienta de modelado cambie la ventana activa al editor externo con el cursor apuntando en la línea concordante del código.

Autogeneración

Esta función es la capacidad de las herramientas de modelado para asistir en la generación de diagramas de estado e interacción.

Así es como debería funcionar: la herramienta de trabajo debería facilitar la creación de un "archivo de rastreo" durante la ejecución de un programa existente. El propósito de este archivo sería el de capturar la interacción entre objetos al pasarse mensajes entre uno y otro.

Después de la creación del "archivo de rastreo", la herramienta de modelado sería utilizada para analizar el rastreo con el fin de hallar patrones de interacción entre los objetos.

La herramienta de modelado permitiría al usuario efectuar la selección para análisis entre un grupo de clases. La herramienta podría entonces presentar cada conjunto único de interacciones que hayan sido registradas a través del "archivo de rastreo" para estas clases y podría permitir que el usuario seleccione las interacciones que desee modelar.

Finalmente, la herramienta permitiría la generación de un diagrama de secuencia o un diagrama de colaboración basándose en interacciones del objeto reales y registradas.

Si parece demasiado futurista, en realidad no lo es, porque la técnica de rastreo ya ha sido implementada con bastante éxito en herramientas que ayudan a los analistas a localizar fallas de rendimiento en sus programas. Un buen ejemplo de productos en esta categoría es Jprobe[3], que es usado para analizar el rendimiento de los programas Java.

[1] Para una mayor referencia consultar: http://java.quest.com/jprobe/jprobe.shtml

Asimismo debería ser posible auto generar diagramas de estado utilizando la misma técnica: una modificación de la secuencia descrita previamente sería permitir al usuario especificar el nombre de una clase base para los estados en la máquina de estado.

La herramienta de modelado rastrearía las interacciones entre clases derivadas de la clase base. A partir de dicho rastreo, la herramienta de modelado podría crear un diagrama de estado mediante la elaboración de un diagrama de cada una de las transiciones de estado registradas.

Herramientas de administración

Lo más probable es que se desee poder medir en qué medida progresa un proyecto orientado a objetos. Una función útil que debería ser integrada en las herramientas de modelado es la capacidad de exportar la información del modelo a una herramienta que le permita verificar el progreso tanto del diseño como el de la ejecución de su proyecto. La hoja de cálculo es una herramienta ideal para aplicar a esta solución debido a su familiaridad y flexibilidad. De igual modo, las herramientas de administración de proyecto son candidatas ideales.

¿Cómo trabajaría esta función? En el nivel más elevado, normalmente lo que se requiere localizar son las clases en el modelo y las personas asignadas para trabajar en ellas. Se desea saber cuando alguien comenzó a trabajar en una clase y qué tan cerca de completar el trabajo está. En el siguiente nivel de detalle, es deseable tener conocimiento acerca de los métodos para cada clase.

En este nivel quizá es necesario saber qué métodos han sido incluidos en los diagramas de interacción o, durante la fase de ejecución, la cantidad de código terminado para cada método.

Para que esta función sea eficaz, necesitará una actualización de tipo avanzado para la información utilizada en la administración del proyecto. A diferencia de las herramientas para preparación de informes, que siempre generan un nuevo informe desde el inicio, sólo se desea exportar todo la primera vez.

Tras la exportación inicial, la herramienta de modelado sólo tendrá que actualizar la herramienta de administración con nueva información. Dependiendo del nivel de control deseado por el usuario, la herramienta de modelado podría presentar al usuario una lista de actualizaciones antes de efectuar la exportación.

Uno de los apreciables dividendos que resultarían al tener un enlace para la administración de proyecto es la capacidad de elegir fechas de finalización de las fases de análisis y de diseño de un proyecto. La forma en la cual esto funcionaría sería mediante el cálculo de un índice de progreso para calcular y anticipar las fechas de finalización basándose en las tareas restantes para completar el modelo.

Métricas

Cuando un proyecto comienza a madurar, se desea tener acceso a métricas para el modelo. Estas métricas pueden proporcionar al arquitecto de la solución información inmediata sobre la viabilidad de un modelo en particular. Algunas métricas de particular interés incluyen: el número de super-clases en una jerarquía de clases, el número de métodos por clase, el número de atributos por clase, el número de métodos "get/set", el número de métodos anulados, las líneas de código para cada método, el porcentaje de métodos públicos, privados y protegidos, el grado de acoplamiento por clase (el número de clases diferentes que conoce esta clase), y el porcentaje de métodos comentado.

La métrica podría proporcionarse a través de una interfaz para preparación de informes, o aún mejor, mediante un enlace a una hoja de cálculo similar al enlace para administración del proyecto descrita anteriormente.

SVG: Gráficas de Vector

Para lograr ofrecer verdadera funcionalidad de importación y exportación de gráficas de vector, basada en estándares, las herramientas UML pronto tendrán la opción SVG (Scalable Vector Graphics) que es una gramática XML para gráficas estilizables que ha progresado hasta lograr una especificación madura en versión 1.0. Una vez que esté completamente aprobada, se podrá buscar esta funcionalidad en la nueva generación de navegadores.

¿Por qué SVG? porque un grupo exportado de diagramas usando el formato de gráficas de vector, puede ser enlazado con páginas web. El lector de un documento de diseño, a través del web, podría utilizar técnicas de navegación gráficas tales como zoom y panorámica dentro del mismo navegador para así poder explorar un diagrama UML grande. Además, este formato mejorará radicalmente la velocidad a la cual diagramas grandes puedan ser montados a través del web, comparado a diagramas de formato gif.

XMI: Enlazándolo todo

El estándar XMI (eXtensible Metadata Interchange) creado por el OMG es uno de los recientes desarrollos de la comunidad de analistas de UML.

XMI es un formato de intercambio basado en XML (que será tratado en el capítulo IV) y que permite tener una representación para modelos de UML.

Así entonces, XMI es un formato de intercambio con el potencial de finalmente permitir a las mejores herramientas de desarrollo compartir modelos fácilmente.

Por ejemplo, en lugar de escribir scripts para crear informes con una herramienta de modelado UML, un usuario simplemente podría exportar el modelo en desarrollo utilizando XMI e importarlo a una herramienta especializada para la creación de informes. De hecho, este paradigma se aplicaría igualmente bien a las funciones discutidas anteriormente: rastreo de métricas orientadas a objetos y la administración de proyectos.

Además, como XMI utiliza XML para representar la información del modelo, un grupo de soluciones basadas en XML estará inmediatamente disponible, tales como las hojas de estilo XSL para la presentación basada en navegadores y las herramientas de explotación (herramientas de consulta) XQL para funciones de búsqueda.

El estándar XMI es complejo y tomará tiempo reconciliar muchas cuestiones de compatibilidad que inevitablemente se suscitarán antes de que su uso pueda generalizarse.

Bibliografía utilizada en este capítulo

Enterprise Architect for J2EE Technology
Mark Cade, Simon Roberts
Sun Microsystems

UML Distilled: A brief guide to the standard object modeling language (2nd Edition)
Martin Fowler; Kendall Scott
Addison Wesley

UML y Patrones: Introducción al análisis y diseño orientado a objetos.
Craig Larman.
Prentice Hall, 1999.

Capítulo II.- Java

La elección de lenguaje para la construcción de la solución es Java, en este capítulo se procura establecer la razón de esta decisión y presentar los cuidados que deben tenerse al momento de buscar portabilidad por medio de este lenguaje.

Asimismo se considera un espacio para hacer una similitud de Java con C++ que debería servir como referencia a los programadores que ya han trabajado con C++.

Características originales de Java

Un hecho coincidente durante 2002 en publicaciones que citan a Java: la inmensa mayoría lo considera como el principal fenómeno actual del World Wide Web.

Esta apreciación respecto a Java no es fortuita. Por el contrario, todo esto tiene relación con las características propias del lenguaje y, sobre todo, con su búsqueda de la verdadera portabilidad.

Atendiendo cuidadosamente las particularidades de Java, se prevé una determinante influencia de este lenguaje en la forma de distribuir y comercializar las aplicaciones informáticas, seguramente este lenguaje permitirá distribuir software gratuito a través de internet, gracias a lo cual millones de usuarios dejarán de pagar por las tradicionales aplicaciones comerciales.

Sobre este aspecto otros analistas, menos categóricos, opinan que dado que cualquier programa podrá implementarse en Java, cada usuario sólo tendrá que pagar por lo que use, lo que de llevarse a la práctica, también implicará un cambio notorio respecto a los actuales mecanismos y estructuras de comercialización de las aplicaciones comerciales y, con ello, en la estructura ocupacional del sector informático.

El origen de Java poco tiene que ver con su actual proyección como lenguaje de programación de uso casi universal y por excelencia, suerte de estándar, para internet.

¿Por qué Java ha sido reconocido y adoptado tan rápidamente por los principales proveedores de tecnologías para internet y por los desarrolladores?

Si se busca una respuesta concisa a esta pregunta se puede decir que por sus características principales: *es simple, pequeño, orientado a objetos, soporta redes, es robusto, seguro, con una arquitectura neutral, portable, interpretado, con desempeño aceptable, multihilos y dinámico.*

Este trabajo se limitará a detallar las características del lenguaje en su versión original lo que, en alguna medida, tiene carácter general y, por lo tanto, de una

u otra forma encuentran lugar en las diferentes versiones de Java. Por este motivo, básicamente seguiremos el esquema de las características técnicas del lenguaje dado por su fabricante principal (Sun), dejando a un lado las características de las otras versiones: Café, Latté y Visual J++.

Simple

Al ver el código de Java, rápidamente identificaremos muchos aspectos y sintaxis iguales a las del lenguaje C/C++ pero, si se analiza el mismo con un poco más de detenimiento, se notará que son lenguajes diferentes.

Sin lugar a dudas que encontraremos a Java más sencillo o simple, en cuanto éste omite aquellos elementos de C++ que raramente son usados, que confunden a los programadores, o que son muy difíciles de comprender y dominar.

Por ejemplo, Java no soporta sobrecarga de operadores (si soporta sobrecarga de métodos), herencia múltiple y algunos otros males. Además, Java dispone de un recolector automático de basura (*garbage collector*) que se encarga de liberar periódicamente las partes de la memoria que no están siendo referenciadas, lo cual es un sueño realizado para los programadores de C/C++, dado que éstos tienen que dedicarle atención a las operaciones típicas de gestión de la memoria dentro de sus programas, con lo que se eleva sensiblemente la complejidad de la programación y, con ello, las posibilidades de error.

Como dato curioso sobre esta característica podemos destacar que los programadores que usan Java lo consideran comparativamente menos complejo que otras herramientas de desarrollo para internet.

Pequeño

Java está diseñado para crear aplicaciones que se ejecutarán en máquinas pequeñas y, principalmente, serán aplicaciones para ser cargadas desde la red y luego ejecutadas en las máquinas cliente.

Por ello, él interprete de Java, al igual que sus librerías, deberán ser muy optimizadas y pequeñas. Esta característica, válida a la fecha, puede variar con la evolución del lenguaje y sus futuros usos. En la medida en que la evolución del lenguaje conserve o no esta característica, limitará o no la aceptación masiva del lenguaje por los desarrolladores.

Orientado a objetos

Esto no es nada nuevo, pues desde hace ya algunos años que los lenguajes más potentes del mercado han demostrado la importancia de la programación orientada a objetos (OOP por sus siglas en inglés) y la posibilidad que ésta ofrece a los desarrolladores de aplicaciones para crear nuevos componentes o usar los fabricados por otras empresas. C++, Pascal, Visual FoxPro, Delphi, Visual Objects, Visual Basic, son ejemplos de la adaptación de la teoría OOP a la práctica en productos y lenguajes comerciales de gran éxito en el mercado actual.

Obviamente Java, al partir de C++, soporta las características de la implementación OOP en este lenguaje y también gran parte de su sintaxis.

Como si ello fuera poco, Java posee una serie de extensiones OOP que lo hacen de cierta forma similar a SmallTalk. Como resultado, la implementación de OOP en Java es muy robusta, capaz de satisfacer hasta los más rigurosos programadores OOP.

Redes

Realizar aplicaciones que manejen conexiones, transferencia de archivos, abrir y acceder a objetos dentro de la red, simplemente indicando su URL (Uniform Resource Locutor por sus siglas en inglés), de la misma forma en que accedemos a los archivos en el disco duro, es bastante más sencillo con Java, debido a que soporta los protocolos TCP/IP (como HTTP y FTP), bastante conocidos por los aficionados a internet y otras redes.

Esto convierte a Java en una de las mejores opciones actuales para el desarrollo de aplicaciones distribuidas (internet/intranet) y, sin lugar a dudas, en una de las causales más importantes para el desarrollo que se operará en las redes de área local a corto y mediano plazo.

Robusto

Java hace una verificación temprana y luego una dinámica para eliminar, hasta donde sea posible, las situaciones de error, lo que hace que las aplicaciones escritas con él sean muy robustas. Por ejemplo, una de las diferencias importantes a este respecto entre Java y C/C++ es que Java tiene un modelo de apuntadores que elimina la posibilidad de sobrescribir o corromper los datos en memoria.

Gracias a esto, los desarrolladores Java no necesitan preocuparse de los típicos problemas de liberación o corrupción de la memoria, debido a que los programas desarrollados en este lenguaje no tienen ninguna forma de acceder a la memoria en forma no autorizada o incorrecta.

Seguro

Java está diseñado para ser utilizado como lenguaje de uso general, apto para crear aplicaciones que funcionen eficientemente en entornos de red. Esto exige, por ejemplo, que existan formas de garantizar la producción y ejecución de software libre de virus o intrusiones.

Existe una relación muy importante entre la robustez y la seguridad de Java. Por ejemplo el modelo Java hace imposible que una aplicación acceda a estructuras o a los datos privados de los objetos para los que no tiene autorización a través del encapsulamiento.

Esta característica es de gran importancia, ya que el ritmo de desarrollo y difusión de internet en la actualidad depende, en gran medida, de los grados de

seguridad eficiente que ésta le ofrezca a sus usuarios (programadores y usuarios finales) para el tratamiento de la información privada.

Sin embargo, ésta es una de las características de Java que, más allá de lo que puedan prometer los productores al respecto, tendrá que ganarse en la práctica de la programación la confianza de los desarrolladores

Arquitectura neutral

El lenguaje está diseñado para crear aplicaciones que puedan ser ejecutadas en diferentes plataformas de hardware y en diferentes sistemas operativos, pues esto es lo más común dentro de una red de computadoras.

Por lo anterior el compilador Java con el que se trabaje tendrá que ser capaz de generar código compilado ejecutable en varias plataformas, en las cuales deberá existir un sistema *Java Runtime* o una máquina virtual de Java conocida más como Java Virtual Machine o JVM.

Esto no sólo es positivo para potencialidad y conectividad de las redes, sino también para los desarrolladores de software quienes, con Java, pueden ofrecer sus aplicaciones sin tener que preocuparse porque sus clientes sean usuarios de sistemas operativos como Windows, Unix, Linux, Macintosh u otros.

Como bien es conocido, en los lenguajes tradicionales el desarrollador está obligado a escribir una versión para cada plataforma de trabajo. En cambio con Java, gracias a su diseño, la misma versión del programa puede ser ejecutada en cada una de las plataformas sin que sea necesario realizarle ninguna modificación. Así de simple.

Java logra esto a través de su compilador, el cual genera una serie de instrucciones (bytecode) que no dependen de una arquitectura determinada y que están especialmente pensadas para su fácil interpretación y, también, para realizar un concepto muy interesante: *convertirlo en código nativo al vuelo* (*compiling on the fly*) por el *Just In Time Compiler* (JIT). Esto es posible gracias a que la máquina virtual de Java (JVM por sus siglas en inglés) está asociada, por lo general, con un navegador de páginas de internet/intranet.

El código Java (contenido en archivos *.Java*), una vez compilado (archivos *.class*), es transmitido por la red hasta llegar al navegador y es éste el que interpretará los bytecodes de Java, ejecutando así el código. En el mejor de los casos se dispone de la implementación de un compilador JIT, lo cual mejorará notablemente el desempeño de ejecución.

Es de destacar que la máquina virtual no tiene porqué estar embebida dentro de un navegador, ya que puede formar parte del sistema operativo y así permitir la ejecución de una aplicación Java como una aplicación normal.

Por lo tanto, es evidente que esta característica de Java termina con los límites de los tradicionales sistemas operativos, abriendo nuevas perspectivas y posibilidades para el desarrollo de aplicaciones.

Dado que ésta es una de las características más novedosas, evidentes y esperadas en el mundo de la programación, la misma cuenta con una elevada aceptación entre los programadores quienes generalmente la destacan como una de las mayores ventajas comparativas de Java respecto al resto de las herramientas de desarrollo.

Portable

Java ya tiene mucho terreno ganado como portable Al tener una arquitectura neutra. Pero portable significa un poco más que eso, por ejemplo, si es portable tiene que garantizar que los tamaños y las reglas aritméticas de los tipos de datos primitivos sean siempre iguales y además que no varíen en función de la plataforma en la que se ejecute el programa.

Hoy en día ya existen librerías que definen las interfaces portables. Por ejemplo, existe una clase abstracta llamada Window que tiene implementaciones especificas para Windows NT/95, Macintosh y UNIX.

Por supuesto, el futuro de la portabilidad de Java y toda la potencia que ello encierra dependerá (como lo ha sido hasta ahora), de la capacidad de los fabricantes de llegar a acuerdos que estandaricen los elementos que definen al lenguaje.

Es deseable que, más allá de las pretensiones de cada fabricante de sacar la máxima ventaja para su empresa, la portabilidad de Java no sea reversible. Y es posible pensar así porque se cree que a partir de lo ya logrado con la portabilidad de Java nada será igual en el mundo del desarrollo de aplicaciones.

Interpretado

Los bytecodes generados por el compilador de Java son convertidos en instrucciones nativas y ejecutadas, siendo almacenadas únicamente en memoria. Dentro de estos bytecodes de Java hay mucha información del proceso de compilación, la cual está disponible en tiempo de ejecución, por lo que permite una mas fácil depuración y varios controles sobre el código.

En este aspecto es de destacar que, a pesar de la fuerte tendencia en el mercado Java de implementar y utilizar los cada vez más poderosos JIT, lo cierto es que esta característica también le brinda a Java las conocidas y no despreciables ventajas de los lenguajes interpretados, por lo que en este sentido el lenguaje brinda opciones para todos los gustos.

Desempeño

Según resultados de pruebas ya difundidos, la velocidad actual de los programas Java es de 10 a 20 veces más lenta que la de las Aplicaciones C++[4].

[4] Thinking in Java, Bruce Eckel, Prentice Hall

Pero al analizar esta situación debemos también destacar que Java aún se encuentra en lo que se puede definir como la infancia de su desarrollo, por lo que el mismo aún está por mostrar el desempeño propio de un lenguaje maduro. Además, y como parte de las soluciones normales para este tipo de limitaciones en la programación, ya existen varias tecnologías (tal vez transitorias, tal vez no), para mejorar este problema.

Gracias a la posibilidad de que los bytecodes de Java pueden ser convertidos en el vuelo en código nativo de la CPU donde se ejecute la aplicación, el desempeño del código Java puede ser mejorada notablemente (en general, de 2 a 10 veces e incluso la mejoría puede ser superior). Obviamente, el nivel de mejoría alcanzado en cada caso dependerá de la implementación del compilador JIT para Java que se use en cada plataforma determinada.

En una óptima relación de esos componentes, la mejoría en desempeño puede llegar a niveles similares a los que se puede alcanzar con C++. No obstante estas soluciones, sin lugar a dudas que la velocidad de los programas Java es una de las limitaciones actuales más notorias de su desempeño y, por lo tanto, en breve debemos esperar algunas novedades tecnológicas para superar de forma más eficiente y definitiva la misma.

Multihilos

Las Aplicaciones Java pueden tener múltiples hilos o vías de ejecución, es decir, varios procesos que se ejecutan al mismo tiempo, lo cual es muy importante para la realización de tareas típicas en las redes tales como la transferencia de archivos de forma más optimizada.

Como es conocido por los programadores, este tipo de aplicaciones son muy difíciles de implementar con lenguajes como C/C++. Sin embargo, en el lenguaje Java existe un conjunto de conceptos (por ejemplo la clase *thread*) que hacen posible crear este tipo de software de una manera mucho más sencilla y robusta.

La importancia de esta característica se manifiesta directamente en que, gracias a la capacidad multihilos, las aplicaciones Java pueden ser interactivas. Esta característica del lenguaje permitirá sustituir, a corto plazo, el carácter estático de la inmensa mayoría de las páginas web por un carácter dinámico.

La interactividad de Java, una capacidad muy esperada por los programadores, es valorada como una ventaja comparativa respecto de este lenguaje al resto de las herramientas de desarrollo. De hecho el recolector de basura de Java aprovecha este concepto al ejecutarse en multihilo.

Dinámico

Cada vez más, al desarrollar software se usan componentes de terceras partes o del sistema operativo todo lo cual, dada la velocidad de los cambios tecnológicos, obligan a estar pendientes de las nuevas versiones y/o actualizaciones para poder poner a tono nuestras aplicaciones con el mercado.

Quienes desarrollan con lenguajes como C/C++ conocen, por sufrirla, esta realidad.

Por su parte Java resuelve las interconexiones entre los módulos de una aplicación en tiempo de ejecución, por lo que se adapta muy bien a los entornos de desarrollo en los cuales se utilizan librerías de clases o componentes desarrollados por otras empresas, entornos de desarrollo en continua evolución, etc., liberando al desarrollador de las típicas preocupaciones que generan la posibilidad de incompatibilidades con los nuevos componentes y versiones de terceros.

Parecido a C++

Muy parecido a C++ y sensiblemente más fácil. Esto, sin dudas, es lo primero que podemos destacar luego de conocer y experimentar la programación con Java.

Además, y por ese motivo, la curva de aprendizaje de Java no es prolongada (esto es mucho más evidente en los programadores con experiencia en C++ y en técnicas de programación orientada a objetos), lo que de alguna manera es reconocido. Esto se logra gracias a que Java, a pesar de ser un nuevo lenguaje, no presenta ningún concepto o tecnología radicalmente novedosa o poco probada.

Java, que tiene toda la potencia y flexibilidad requerida para ser un lenguaje de uso general, elimina una serie de problemas a la programación en lenguajes tradicionales en los cuales, hasta los programadores más experimentados, pierden mucho tiempo. Un buen ejemplo de los problemas eliminados es el relacionado con la gestión de la memoria.

Java es un lenguaje basado en lo mejor de los lenguajes y tecnologías actuales, que pretende ser la herramienta de desarrollo más importante en todo lo que se relaciona con las necesidades de los nuevos paradigmas de interconectividad: internet/intranet.

Por ello, y a pesar de sus pocos años de existencia, Java no sólo ha despertado tanto interés en las empresas productoras de herramientas de desarrollo (Borland, Microsoft, Symantec, etc.), sino que se ha convertido en un elemento estratégico para el futuro de dichas empresas.

Por esto estas tres empresas están trabajando activamente con Java para tratar de convertirlo, o adaptarlo a su tecnología y, de esta forma, garantizar para sus respectivos productos una mejor relación con el mundo Java el cual, según se proyecta, será determinante en el futuro inmediato de la informática.

Uno de los casos más notables al respecto es la integración del modelo de objetos componentes *COM* de Microsoft con Java. *COM* o *Component Object Model* ahora *DCOM*, *Distributed Component Object Model* es el centro de todo la tecnología OLE ahora llamada ActiveX (ActiveX Controls, ActiveX Documents, etc.).

En la implementación de la máquina virtual de Java de Microsoft un programa escrito en Java puede incorporar la tecnología ActiveX (controles, documentos), y con Visual J++ (Java de Microsoft) se puede crear controles ActiveX (controles OLE) para utilizarlos desde Delphi, Visual Basic, etc.

En fin, que si bien Java está lejos de ser la panacea que resuelva todos los problemas, complejidades y limitaciones de los tradicionales lenguajes de programación, también es posible pensar que el lenguaje representa un nuevo hito en el desarrollo de las tecnologías de programación, capaz de responder a las exigencias tecnológicas y de comunicación del presente, gracias al cual las formas y métodos de programación sufrirán cambios importantes.

En realidad no hay certeza de que Java sea el lenguaje del futuro pero, lo que si es evidente, es que las tecnologías y filosofía de programación del futuro inmediato de la programación contendrán mucho de lo que hoy caracteriza Java.

Si Java se difunde en el ámbito internacional de acuerdo a las proyecciones actuales, influirá notoriamente en el sector informático y a partir de él en otros sectores, sacudiendo hasta sus cimientos sus estructuras de producción, lo que lógicamente afecta directamente al lugar y papel de los desarrolladores, distribución y consumo.

La orientación a objetos en Java

Una de las características citadas previamente, el soporte a la orientación a objetos, es muy importante. Java soporta las características esenciales del paradigma de la programación a objetos: encapsulamiento, herencia y polimorfismo. Java hace uso de la definición de entidades formadas por métodos y atributos que reciben el nombre de clases, la instancia de alguna clase cualquiera en Java recibe el nombre de objeto.

Clases en Java.

El elemento básico de la programación orientada a objetos en Java es la clase. Las clases en Java definen las entidades que conformarán la aplicación. Una clase es definida por medio de atributos y métodos.

En Java todo elemento o es parte de una clase o bien es una clase o bien describe el funcionamiento de una clase.

Supóngase, únicamente para propósitos de ejemplo, que queremos implementar una aplicación que utiliza objetos puntos con componentes "x, y", que pueden desplazarse tales puntos con un radio constante de 10 unidades, para poder crear estos objetos, en Java definimos primero el molde Punto de donde posteriormente podemos instanciar los objetos requeridos. El siguiente código puede ser una solución a este problema:

```
class Punto
{
int x;
int y;
static int radio = 10;
Punto (int a, int b)
{
x= a;
y= b;
}
void desplazarX(int t)
{
x= x + t;
}
void desplazarY (int h)
{
y = y + h;
}
```

Los atributos de la clase en este caso son: "x, y". El comportamiento de esta clase se define en los métodos desplazarX, desplazarY. El pseudo método Punto, recibe el nombre de constructor y es utilizado para inicializar valores de los atributos de una clase.

En Java, todas las clases son por omisión derivadas de una clase madre llamada Object. Las clases pueden ser de tipo: **public**, lo que significa que son accesibles desde otras clases; **abstract**, las cuales definen sus métodos pero no los implementan sino que son utilizadas para servir como clases madres a otras clases que se crean a partir de éstas; **final**, las cuales ya no pueden ser clases madre para ninguna subclase; **synchronizable**, las clases de este tipo especifican que sus miembros (variables y métodos) no pueden ser accedidos desde diferentes tareas al mismo tiempo, lo que evita que haya sobreescritura de datos.

Los miembros de una clase pueden ser definidos de dos formas, de instancia y de clase. Si una variable es definida como variable de instancia significa que cada vez que se genere la instancia de un objeto de dicha clase, el objeto puede contener cualquier valor para dicha variable. Retomando el ejemplo anterior, si se crean diferentes instancias de la clase punto, cada instancia puede tomar valores diferentes para sus variables "x, y".

En Java, por omisión todos los miembros de una clase son de instancia. Si es necesario que un componente del objeto no cambie sino que permanezca constante para todos los objetos del mismo tipo, es conveniente declararlo como un miembro de clase, en el ejemplo anterior se vio que todos los puntos creados tenían el radio constante, en este caso definimos la variable radio como variable de clase, en Java un miembro clase se define por medio de la palabra reservada "static".

Objetos en Java

Como se ha mencionando, en Java el componente principal para implementar la orientación a objetos es la clase, la clase es el molde o prototipo de donde se generan las instancias de los objetos. El objeto es entonces, un caso particular de la clase.

Se revisará nuevamente el ejemplo que se trabajó en párrafos anteriores, supóngase que es necesario instanciar dos objetos punto para una cierta aplicación, en Java esto puede hacerse de la siguiente manera:

```
. . .
Punto p0 = new Punto(5,10);
Punto p1 = new Punto(6,12);
. . .
```

En este caso los objetos p0 y p1 son instancias de la clase Punto, cada cual con valores diferentes en sus variables de instancia: "x, y", con el mismo valor en su variable de clase: radio y con las mismas capacidades de comportamiento desplazarX y desplazarY.

Una vez que se ha creado un objeto, se pueden modificar sus miembros de la siguiente manera:

```
. . .
p1.desplazarX(15);
. . .
```

En este caso, se modifica el componente "x" del objeto p1 haciendo un llamado al método desplazarX con el valor 15, lo que da por resultado añadir 15 unidades al valor de la variable "x", únicamente del punto p1.

La creación de un objeto en Java se realiza de acuerdo a los siguientes puntos:

- Declaración. Nombre del objeto.
- Instanciación. Asignación de memoria al objeto.
- Inicialización. Opcionalmente, asignar valores iniciales al objeto.

Herencia.

En Java una clase puede definirse como la extensión de una clase madre y a su vez esta clase puede servir como clase madre para que otras subclases deriven de esta misma.

Suponiendo que se necesita definir objetos "punto" muy semejantes a los que se definieron en el ejemplo anterior pero que además tengan otras propiedades como peso y color y que se desplacen las mismas unidades tanto en "x" como en "y", se tienen dos opciones, una es construir otra clase totalmente nueva que tenga una implementación muy similar a la de la clase "Punto" pero además agregue los nuevos miembros, opción que no es muy recomendable puesto que es trabajar sobre algo que ya se hizo, la segunda opción es crear una nueva clase partiendo de la clase "Punto".

El siguiente código muestra una forma de implementar esta nueva clase a la que se llamará PuntoPC que derivará de la clase Punto.

```
class PuntoPC extends Punto
{
String color;
float peso;
PuntoPC(int a, int b, String c, float p)
{
super (a,b);
color = c;
peso = p;
}
void desplazarAmbos(int d)
{
x= x+d;
y= y+d;
}
}
```

En este caso, el único comentario adicional es especificar que el constructor PuntoPC toma los valores iniciales a y b y los pasa a la super clase Punto y asigna los valores peso y color a sus correspondientes variables.

En Java no existe la herencia múltiple como tal, es decir una clase no puede tener más de una clase madre, sin embargo los beneficios de la herencia múltiple pueden ser implementados mediante el uso de interfaces, mismas que se discutirán a continuación.

Interfaces

De manera informal se puede definir a una interfaz como una clase en la que se declaran valores constantes y métodos pero no se implementan.

Tanto las clases como las interfaces tienen prácticamente el mismo formato general, los siguientes puntos definen las diferencias que existen entre una clase y una interfaz.

- Una interfaz al igual que una clase abstracta, declara los métodos pero no los implementa.
- Una clase puede implementar varias interfaces, manteniendo la estructura del lenguaje simple y al mismo tiempo, proveyendo las ventajas de la herencia múltiple.
- No se pueden instanciar objetos a partir de una interfaz.
- Todos los métodos de una interfaz son de forma implícita públicos y abstractos.
- Todas las variables de una interfaz son de forma implícita públicas, estáticas y finales.
- Cuando una clase implementa una interfaz, debe implementar todos y cada uno de sus métodos, siempre y cuando, la clase no sea abstracta.

- Una interfaz no tiene una clase padre antecesora (como en el caso de las clases que por omisión tienen a la clase Object), en vez de eso, las clases tienen una jerarquía independiente que puede ser aplicada a cualquier nivel del árbol de clases.

Así pues, la clase es el concepto fundamental de Java para implementar el paradigma de la tecnología orientada a objetos, por medio de la clase, Java logra modularidad, polimorfismo, herencia y encapsulamiento, determinantes en la concepción de un lenguaje orientado a objetos.

Similitudes y diferencias de Java con C++

La elección de lenguaje en este trabajo es Java y por tomar este lenguaje algunas características de C++ es oportuno precisar sus similitudes y diferencias. Comparación que habrá de servir como referencia respecto a los cambios que representa Java sobre C++ y donde se debe cambiar el concepto de las cosas.

Lo anterior es necesario fundamentalmente porque hay características que siguen utilizándose en Java heredadas de C++, pero el sentido con que se aplican es ligeramente diferente, lo que muchas veces lleva a los programadores C++ que atacan Java, a confusiones innecesarias.

Por ello, aunque un poco denso y tedioso, es una sección obligada para tener en cuenta las cosas que se deben alterar en la forma de concebir la programación cuando se utiliza Java.

Java no soporta **typedefs**, **defines** o instrucciones de *preprocesador*. Al no existir un preprocesador, no está prevista la inclusión de archivos de cabecera, tampoco tiene cabida el concepto de **macro** o **constante**. Sin embargo, sí se permite cierto uso de *constantes enumeradas* a través de la utilización de la palabra clave **final**. Java tampoco soporta **enums**, aunque soporte constantes enumeradas.

Java soporta **clases**, pero no soporta **estructuras** o **uniones**.

Todas las aplicaciones C++ necesitan una función de entrada llamada *main()* y puede haber multitud de otras funciones, tanto funciones miembros de una clase como funciones independientes. En Java no hay funciones independientes, absolutamente todas las funciones han de ser miembros (métodos) de alguna clase. Funciones globales y datos globales en Java no están permitidos.

En C++ pueden ser creados árboles de herencia de clases independientes unos de otros. En Java esto no es posible, en última instancia hay una clase **Object**, de la que hereda todo lo que el programador necesite.

Todas las funciones o definiciones de métodos en Java deben estar contenidas dentro de la definición de la clase. Para un programador C++ puede parecerle que esto es semejante a las funciones **inline**, pero no es así. Java no permite, al menos directamente, que el programador pueda declarar funciones **inline**.

Tanto Java como C++ soportan funciones o métodos estáticos de clases, que pueden ser invocados sin necesidad de tener que instanciar ningún objeto de la clase.

En Java se introduce el concepto de **interfaz**, que no existe en C++. Una **interfaz** se utiliza para crear una clase base abstracta que contenga solamente las constantes y las declaraciones de los métodos de la clase. No se permite que haya variables miembro ni definiciones de métodos. Además, en Java también se pueden crear verdaderas clases abstractas.

Java no soporta **herencia múltiple**, aunque se pueden utilizar las posibilidades que ofrece el uso de interfaces para emplear las ventajas que ofrece la herencia múltiple, evitando los inconvenientes que se derivan de su uso. La herencia simple es similar en Java y en C++, aunque la forma en que se implementa es bastante diferente, especialmente en lo que respecta a la utilización de los constructores en la cadena de herencia.

Java no soporta la sentencia **goto**, aunque sea una palabra reservada. Sin embargo, soporta las sentencias **break** y **continue** con etiquetas, que no están soportadas por C++. Bajo ciertas circunstancias, estas sentencias etiquetadas se pueden utilizar como un *go encubierto*.

Java no soporta la **sobrecarga de operadores**, aunque la utilice internamente, pero no está disponible para el programador. Tampoco soporta la **conversión automática de tipos**, excepto en las *conversiones seguras*.

Al contrario que C++, Java dispone de un tipo **String** y los objetos de este tipo no pueden modificarse. Las cadenas que se encuentren definidas entre comillas dobles son convertidas automáticamente a objetos **String**. Java también dispone del tipo **StringBuffer**, cuyos objetos sí se pueden modificar, y además se proporcionan una serie de métodos para permitir la manipulación de cadenas de este tipo.

Al contrario que C++, Java trata a los **arrays** (arreglos) como objetos reales. Disponen de un atributo, **length**, que indica la longitud del arreglo. Se genera una excepción cuando se intenta sobrepasar el límite indicado por esta longitud. Todos los arreglos son instanciados en memoria dinámica y se permite la asignación de un arreglo a otro; sin embargo, cuando se realiza una asignación, simplemente hay dos referencias a un mismo arreglo, no hay dos copias del arreglo, por lo que sí se altera un elemento de un arreglo, también se alterará en el otro. A diferencia de C++, el tener dos apuntadores o referencias a un mismo objeto en memoria dinámica no representa necesariamente un problema, aunque sí puede provocar resultados imprevistos. En Java, la memoria dinámica es liberada automáticamente, pero esta liberación no se lleva a cabo hasta que todas las referencias a esa memoria son NULL o dejan de existir. Por tanto, a diferencia de C++, una zona de memoria dinámica nunca puede ser inválida mientras esté siendo referenciada por alguna variable.

Java no soporta **apuntadores**, al menos en el sentido que lo hace C++, es decir, no permite modificar el contenido de una zona de memoria marcada por un apuntador, o realizar operaciones aritméticas con apuntadores. La mayor necesidad de uso de apuntadores deriva de la utilización de cadenas y arreglos, y esto se evita al ser objetos de primera clase en Java.

Por ejemplo, la declaración imprescindible en C++, char *apuntador, para referenciar al primer elemento de una cadena no se necesita en Java, al ser la cadena un objeto String.

La **definición de clase** es semejante en Java y C++, aunque en Java no es necesario el punto y coma (;) final. El operador de ámbito (::) necesario en C++ no se usa en Java, sino que se utiliza el punto (.) para construir referencias. Y, como no hay soporte de apuntadores, el operador flecha (->) tampoco está soportado en Java.

En C++, las funciones y atributos se invocan utilizando el nombre de la clase y el nombre del miembro estático conectados por el operador de ámbito. En Java, se utiliza el punto (.) para conseguir el mismo propósito.

Al igual que C++, Java dispone de tipos primitivos como **int**, **float**, etc. Pero, al contrario de C++, los tamaños de estos tipos son iguales independientemente de la plataforma en que se estén utilizando. No hay tipos sin signo en Java, y la comprobación de tipos es mucho más restrictiva en Java que en C++. Java dispone de un tipo booleano verdadero.

Las expresiones condicionales en Java se evalúan a booleano en vez de a entero como en el caso de C++. Es decir, en Java no se permiten sentencias del tipo if(x+y), porque la expresión que va dentro del paréntesis no se evalúa a booleano.

El tipo **char** en C++ es un tipo de 8 bits que mapea el conjunto completo de caracteres ASCII. En Java, el tipo **char** es de 16 bits y utiliza el conjunto de caracteres Unicode (los caracteres del 0 al 127 del set Unicode, coinciden con el conjunto ASCII).

Al contrario que en C++, el operador desplazamiento (>>) es un operador con signo, insertando el bit de signo en la posición vacía. Por ello, Java incorpora el operador >>>, que inserta ceros en las posiciones que van quedando vacías tras el desplazamiento.

C++ permite la instanciación de variables u objetos de cualquier tipo en tiempo de compilación sobre memoria estática o, en tiempo de ejecución, sobre memoria dinámica. Sin embargo, Java requiere que todas las variables de tipos primitivos sean instanciadas en tiempo de compilación, y todos los objetos sean instanciados en memoria dinámica en tiempo de ejecución.

Java proporciona clases de envoltura para todos los tipos primitivos, excepto para **byte** y **short**, que permiten que estos tipos primitivos puedan ser

instanciados en memoria dinámica como objetos en tiempo de ejecución, si fuese necesario.

C++ requiere que las clases y funciones estén declaradas antes de utilizarlas por primera vez. Esto no es necesario en Java. C++ también requiere que los miembros estáticos de una clase se redeclaren fuera de la clase. Esto tampoco es necesario en Java.

En C++, si no se indican valores de inicialización para las variables de tipos primitivos, éstas pueden contener basura. Aunque las variables locales de tipos primitivos se pueden inicializar en la declaración, los datos miembros de tipo primitivo de una clase no se pueden inicializar en la definición de la clase.

En Java, se pueden inicializar estos datos miembros de tipo primitivo en la declaración de la clase. También se pueden inicializar en el constructor. Si la inicialización no se realiza explícitamente, o falla por lo que sea, los datos son inicializados a cero (o su equivalente) automáticamente.

Al igual que ocurre en C++, Java también soporta constructores que pueden ser sobrecargados. Y, del mismo modo que sucede en C++, si no se proporciona un constructor explícitamente, el sistema proporciona un constructor por defecto.

En Java todos los objetos se pasan por referencia, eliminando la necesidad del constructor copia utilizado en C++.

No hay destructores en Java. La memoria que no se utiliza es devuelta al sistema a través del *reciclador de memoria*, que se ejecuta en una tarea (hilo) diferente al del programa principal. Ésta es una de las diferencias más importantes entre C++ y Java.

Como C++, Java también soporta la sobrecarga de funciones. Sin embargo, el uso de argumentos por defecto no está soportado en Java. Al contrario que C++, Java no soporta *templates o plantillas*, por lo que no existen funciones o clases genéricas.

El *multithreading* o multihilo, es algo característico de Java, que se proporciona por defecto.

Aunque Java utiliza las mismas palabras clave que C++ para indicar el control de acceso: private, public y protected, la interpretación es significativamente diferente entre C++ y Java.

La implementación del manejo de *excepciones* en Java es más completo y bastante diferente al empleado en C++.

Al contrario que C++, Java no soporta la sobrecarga de operadores. No obstante, los operadores + y += son sobrecargados automáticamente para concatenar cadenas, o para convertir otros tipos a cadenas.

Como en C++, las Aplicaciones Java pueden hacer llamadas a funciones escritas en otros lenguajes, llamadas *métodos nativos*.

A diferencia de C++, Java dispone de un sistema interno de generación de documentación. Si se utilizan comentarios escritos de determinada forma, se puede utilizar la herramienta **Javadoc** para generar la documentación de la Aplicación Java, e incluso se pueden programar *doclets* para generar tipos específicos de documentación.

Errores de programación Java mas frecuentes relativos a portabilidad

Java ofrece alcanzar la portabilidad pero ésta no es automática, por lo que el lenguaje debe utilizarse adecuadamente. Existen muchos "errores frecuentes" que muestran que la portabilidad no es automática, en breve se listarán algunos. Los errores mostrados no tienen ningún orden de dificultad, no son más que un grupo de errores en los que puede caer cualquier programador.

Hay bastantes formas de cometer fallos a la hora de programar en Java; algunas se deben simplemente a malos hábitos y son muy difíciles de encontrar, mientras que otros saltan a la vista al instante. Los errores de programación más obvios, también son los que con más frecuencia cometen los programadores.

Quizá, muchos de los fallos se evitarían si los programadores intentarán aplicar calidad a sus programas desde el momento mismo de concebir el programa, y no la tendencia de aplicar pureza a la aplicación en el último momento.

Itinerario de hilos de ejecución

El itinerario de los hilos de ejecución, es decir, el tiempo que el sistema destina a la ejecución de cada uno de los hilos de ejecución, puede ser distinto en diferentes plataformas. Si no se tienen en cuenta las prioridades o se deja al azar la prevención de que dos hilos de ejecución accedan a un mismo objeto al mismo tiempo, el programa no será portable.

El siguiente programa, por ejemplo, no es portable.

```
class Contador implements Runnable {
   static long valor = 0;

   public void run() {
      valor += 1;
   }

   public static void main( String args[] ) {
      try {
         Thread hilo1 = new Thread( new Contador() );
         hilo1.setPriority( 1 );
         Thread hilo2 = new Thread( new Contador() );
         hilo2.setPriority( 2 );

         hilo1.start();
         hilo2.start();

         hilo1.join();
         hilo2.join();

         Systtem.out.println( valor );
      } catch( Exception e ) {
         e.printStackTrace();
      }
   }
}
```

Este programa puede no imprimir "2" en todas las plataformas, porque los dos hilos de ejecución no están sincronizados y, desgraciadamente, éste es un problema muy profundo y no hay forma de detectar su presencia ni adivinar el momento en que va a ocurrir.

Una solución simple, y drástica, es hacer todos los métodos sincronizados, aunque esto puede tener problemas también. Así que, el itinerario de los hilos de ejecución es uno de los aspectos más problemáticos de la programación Java, porque la naturaleza del problema se vuelve global, al intervenir varios hilos de ejecución. No se puede buscar el problema en una parte del programa, es imprescindible entender y tratar el programa en su globalidad.

Además, hay ejemplos de contención de hilos que no serán detectados. Por ejemplo, en la clase **Contador** anterior no se detectará el problema ya que la contención está en el acceso al campo, en lugar de en el acceso al método.

Errores en el uso de las características de portabilidad de Java

Hay características de portabilidad en el API de Java. Es posible, pero menos portable, escribir código que no haga uso de estas características. Muchas de las propiedades del sistema proporcionan información sobre la portabilidad; por ejemplo, se pueden utilizar las propiedades del sistema para conocer cuál es el

carácter definido como fin de línea o el que se emplea como terminador de archivo, para emplear el adecuado a la plataforma en que se está ejecutando el programa.

Java proporciona dos métodos para facilitar la escritura de programas portables en este sentido. Por un lado, utilizar el método *println()* en vez de imprimir las cadenas seguidas del terminador de cadena embebido; o también, utilizar la expresión *System.getProperty("line.separator")* para conocer cuál es el terminado de línea que se utiliza en la plataforma en que se está ejecutando el programa. En general, el uso de las propiedades facilita en gran modo la portabilidad y debería extenderse su uso siempre que fuese aplicable.

De igual modo, la API AWT de Java abstrae muchas de las operaciones que se hacen con el sistema de ventanas de la plataforma. Usar esta abstracción siempre; por ejemplo, utilizar **LayoutManager** en vez de posicionar en coordenadas absolutas, utilizar los diferentes métodos *getSize()*, utilizar los colores disponibles en **Java.awt.SystemColor**.

Uso directo de las clases de bajo nivel del AWT

Los programas que se pretende sean portables y que vayan a implementar interfaces basadas en AWT no deberían utilizar las clases base del AWT directamente. El protocolo para la interacción con estas clases es muy dependiente de la plataforma.

La solución es sencilla, basta con mantenerse en el lado cliente del AWT. Quizá esto haya sido un problema de documentación, porque en versiones antiguas del JDK se han descrito estas clases, cuando nunca se ha pretendido que sean de uso público en programas cliente.

Uso de directorios definidos

Un error muy común y fácil de cometer entre los programadores, aunque igual de fácil de corregir es la designación en el código de nombre de archivos, que pueden dar lugar a problemas de portabilidad, pero cuando se añade el directorio en que se sitúan, seguro que estos problemas aparecerán.

Estos fallos son más comunes entre programadores con viejos hábitos, que eran dependientes del sistema operativo, y que son difíciles de olvidar.

La forma más portable de construir un **File** para un archivo en un directorio es utilizar el constructor *File(File, String)*. Otra forma sería utilizar las propiedades para conocer cuál es el separador de archivos y el directorio inicial; o también, preguntarle al operador a través de una caja de diálogo.

Otro problema es la noción de camino absoluto, que es dependiente del sistema. En UNIX los caminos absolutos empiezan por /, mientras que en Windows pueden empezar por cualquier letra. Por esta razón, el uso de caminos absolutos que no sean dependientes de una entrada por operador o de la consulta de las propiedades del sistema no será portable.

El ejemplo siguiente proporciona una clase útil para la construcción de nombres de archivos. La última versión del JDK (poner la última versión) es mucho más exhaustiva, y detecta más fácilmente los errores cometidos en los directorios y nombres de archivos.

```
import Java.io.File;
import Java.util.StringTokenizer;

public class UtilArchivo {
 /* Crea un nuevo archivo con el nombre de otros. Si la base inicial es
  * nula, parte del directorio actual
  */
 public static File dirInicial( File base,String path[] ) {
  File valor = base;
        int i=0;

        if( valor == null && path.length == 0 ) {
          valor = new File( path[i++] );
          }

        for( ; i < path.length; i++ ) {
          valor = new File( valor,path[i] );
          }

  return( valor );
        }

 public static File desdeOrigen( String path[] ) {
  return( dirInicial( null,path ) );
        }

 public static File desdeProp( String nombrePropiedad ) {
  String pd = System.getProperty( nombrePropiedad );

        return( new File( pd ) );
        }

 // Utilizando la propiedad del sistema "user.dir"
 public static File userDir() {
  return( desdeProp( "user.dir" ) );
        }

 // Utilizando la propiedad del sistema "Java.home"
 public static File JavaHome() {
  return( desdeProp( "Java.home" ) );
        }

 // Utilizando la propiedad del sistema "user.home"
```

```
public static File userHome() {
  return( desdeProp( "user.home" ) );
        }

/* Separa el primer argumento, utilizando el segundo argumento como
 * carácter separador.
 * Es muy útil a la hora de crear caminos de archivos portables
 */
public static String[] split( String p,String sep ) {
  StringTokenizer st = new StringTokenizer( p,sep );
  String valor[] = new String[st.countTokens()];

  for( int i=0; i < valor.length; i++ ) {
    valor[i] = st.nextToken();
   }

        return( valor );
  }
}
```

Carga de manejadores JDBC

La interfaz JDBC (Java DataBase Connectivity[5]), definido por el paquete **Java.sql**, proporciona gran flexibilidad a la hora de codificar la carga del manejador JDBC a utilizar. Esta flexibilidad permite la sustitución de diferentes manejadores sin que haya que modificar el código, a través de la clase **DriverManager**, que selecciona entre los manejadores disponibles en el momento de establecer la conexión.

Los manejadores se pueden poner a disposición de **DriverManager** a través de la propiedad del sistema **jdbc.drivers** o cargándolos explícitamente usando el método *Java.lang.Class.forName()*.

También es posible la carga de una selección de manejadores, dejando que el mecanismo de selección de **DriverManager** encuentre el adecuado en el momento de establecer la conexión con la base de datos. Hay que tener siempre en cuenta los siguientes puntos:

- La prueba de manejadores se intenta siempre en el orden en que se han registrado, por lo que los primeros tienen prioridad sobre los últimos cargados, con la máxima prioridad para los listados en **jdbc.drivers**.
- Un manejador que incluya código nativo fallará al cargarlo sobre cualquier plataforma diferente de la que fue diseñado; por lo que el programa deberá atrapar la excepción **ClassNotFoundException**.
- Un manejador con código nativo no debe registrarse con **DriverManager** hasta que no se sepa que la carga ha tenido éxito.
- Un manejador con código nativo no está protegido por la caja negra de Java, así que puede presentar potenciales problemas de seguridad.

[5] JDBC es el API desarrollado por SUN para acceso a base de datos, fue definido siguiendo la idea del estándar ODBC (Open DataBase Connectivity).

Terminación de líneas

Las distintas plataformas de Sistemas operativos tienen distintas convenciones para la terminación de líneas en un archivo de texto. Por esto debería utilizarse el método *println()*, o la propiedad del sistema **line.separator**, para la salida; y para la entrada utilizar los métodos *readLine()*.

Java internamente utiliza *Unicode*, que al ser un estándar internacional, soluciona el problema a la hora de codificar; pero el problema persiste al leer o escribir texto en un archivo.

En el JDK 1.1 se utilizan las clases **Java.io.Reader** y **Java.io.Writer** para manejar la conversión del conjunto de caracteres, pero el problema puede surgir cuando se leen o escriben archivos ASCII planos, porque en el ASCII estándar no hay un carácter específico para la terminación de líneas; algunas máquinas utilizan \n, otras usan \r, y otras emplean la secuencia \r\n.

Enarbolando la bandera de la portabilidad, deberían utilizarse los métodos *println()* para escribir una línea de texto, o colocar un marcador de fin de línea.

También, usar el método *readLine()* de la clase **Java.io.BufferedReader** para recoger una línea completa de texto. Los otros métodos *readLine()* son igualmente útiles, pero el de la clase **BufferedReader** proporciona al código también la traslación.

Archivos de entrada/salida

Las clases de entrada y salida del JDK 1.2 no son portables a plataformas que no soporten formatos nativos de archivos no-ASCII. Es fácil para el programador suponer arbitrariamente que todo el mundo es ASCII.

Pero la realidad no es esa, los chinos y los japoneses, por ejemplo, no pueden escribir nada con los caracteres ASCII. Hay que tener esto en cuenta si se quiere que los programas viajen fuera del país propio.

Tamaño de los elementos de la interfaz gráfica

El tamaño exacto de los elementos del AWT es diferente de una plataforma a otra, como tampoco son iguales la pantalla y los tamaños máximo y mínimo que toma por defecto una ventana. Un botón renderizado para un "Mac" con un tamaño de pixel fijo, seguro que no será renderizado con ese mismo rango en cualquier otro sistema operativo.

Fuentes de caracteres

El tamaño y disponibilidad de varios tipos de fuentes varía de pantalla a pantalla, incluso en una misma plataforma de hardware, dependiendo de la instalación que se haya hecho. Esto es algo que no descalifica totalmente el programa, porque se verá de manera defectuosa, pero el programa podrá seguir usándose; sin embargo debería prevenirse, porque se presupone que el programador desea que su software aparezca de la mejor manera posible en cualquier plataforma.

El modo mejor de evitar todo esto es no codificar directamente el tamaño de los textos, dejar que los textos asuman su tamaño con relación al *layout*, y utilizar los métodos de la clase **FontMetrics** para encontrar el tamaño en que aparecen los caracteres de una cadena sobre un **Canvas**.

Cuando se coloca una fuente que no se encuentra entre las de defecto, hay que asegurarse siempre de colocar alguna de respaldo en el bloque catch.

Cuando se crea un menú para seleccionar fuentes de caracteres, se debería utilizar el método *Java.awt.Toolkit.getFontList()*, en lugar de especificar una lista de fuentes.

El protocolo paint()

Los métodos del AWT *Component.paint()* y *Component.update()* tienen como parámetro de entrada un objeto de tipo **Graphics**. Este objeto no debe ser persistente, sino solamente válido durante la acción del método; la implementación del AWT es libre de destruir este objeto **Graphics** después de que el método *paint()* le devuelva el control, lo que hace peligroso retener el objeto.

No se debe retener pues el objeto **Graphics**; en general, no es seguro pintar fuera de los métodos *update()* o *paint()*; si se necesita una vida más larga para el objeto **Graphics**, se debe crear uno a partir del argumento de entrada al método, pero teniendo sumo cuidado, porque esto no funciona en todas las plataformas.

```
Graphics miGraphics = null;  // retenido

void paint( Graphics g ) {
  if( miGraphics == null ) {
    miGraphics = g.create();
    }
  // El código de pintado iría aquí
}
```

Bibliografía utilizada en este capítulo

Thinking In Java
Bruce Eckel
Prentice Hall

Professional Java Programming
Brett Spell
Wrox

Java I/O
Elliote Rusty Harold
O'Reilly

The Java Programming Language (Third Edition)

Ken Arnold, James Gosling, David Holmes
Sun Microsystems

Database Programming with JDBC and Java
George Reese
O'Reilly

Capítulo III.- Seguridad en Java

Una de las características de Java que se debe detallar es la seguridad, a continuación se verá que ofrece este lenguaje en este sentido.

El modelo de seguridad de Java se conoce como modelo del patio de juegos, aludiendo a esos rectángulos con arena donde se deja jugando a los niños pequeños, de manera que puedan hacer lo que quieran dentro del mismo, pero no puedan salir al exterior.

En concreto, este modelo se implementa mediante la construcción de cuatro barreras o líneas de defensa:

- Primera línea de defensa: Características del lenguaje/compilador
- Segunda línea de defensa: Verificador de código de bytes
- Tercera línea de defensa: Cargador de clases
- Cuarta línea de defensa: Gestor de seguridad

Es importante señalar que aunque se hable de barreras de defensa, no se trata de barreras sucesivas. Es decir, no se trata de que si se traspasa la primera barrera, hay que superar la segunda, y luego la tercera y por fin la última, como muros uno detrás de otros.

Más bien hay que imaginar una fortaleza con cuatro muros, y basta que se penetre uno de ellos para que la fortaleza caiga en manos del enemigo. Así que más que de líneas de defensa, habría que hablar de varios frentes.

En las próximas páginas se verá cada una de ellas en detalle. Se comenzará con la primera barrera: las características del lenguaje/compilador.

Características del lenguaje/compilador

Java fue diseñado con las siguientes ideas en mente:

- Evitar errores de memoria
- Imposibilitar acceso al SO
- Evitar que caiga la máquina sobre la que corre

Con el fin de llevar a la práctica estos objetivos, se implementaron las siguientes características:

Ausencia de apuntadores

Protege frente a imitación de objetos, violación de encapsulamiento, acceso a áreas protegidas de memoria, ya que el programador no podrá referenciar posiciones de memoria específicas no reservadas, a diferencia de lo que se puede hacer en C y C++.

Gestión de memoria

Ya no se puede gestionar la memoria de forma tan directa como en C, (no hay "malloc"). En cambio, se crean instancias de objetos, no se reserva memoria directamente (**new** siempre devuelve un manejador), minimizando así la interacción del programador con la memoria y con el SO.

Recolección de basura

El programador ya no libera la memoria manualmente mediante **free** (fuente muy común de errores en C y C++, que podría llegar a producir el agotamiento de la memoria del sistema).

El recolector de basura de Java se encarga de reclamar la memoria usada por un objeto una vez que éste ya no es accesible o desaparece. Así, al ceder parte de la gestión de memoria a Java en vez de al programador, se evitan las grietas de memoria (no reclamar espacio que ya no es usado más) y los apuntadores huérfanos (liberar espacio válido antes de tiempo).

Arreglos con comprobación de límites

En Java los arreglos son objetos, lo cual les confiere ciertas funciones muy útiles, como la comprobación de límites. Para cada subíndice, Java comprueba si se encuentra en el rango definido según el número de elementos del arreglo, previniendo así que se haga referencia a elementos fuera de límite.

Referencias a objetos fuertemente tipados

Impide conversiones de tipo y casteos para evitar accesos fuera de límites de memoria (resolución en compilación).

Casteo seguro

Sólo se permite casteo entre ciertas primitivas de lenguaje (ints, longs) y entre objetos de la misma rama del árbol de herencia (uno desciende del otro y no al revés), en tiempo de ejecución.

Control de métodos y variables de clases

Las variables y los métodos declarados privados sólo son accesibles por la clase o subclases herederas de ella y los declarados como protegidos, sólo por la clase.

Métodos y clases final

Las clases y los métodos (e incluso los datos miembro) declarados como **final** no pueden ser modificados o sobrescritos. Una clase declarada final no puede ser ni siquiera extendida.

Verificador de código de bytes

¿Qué ocurriría si se modifica un compilador de C para producir código de bytes (bytecode) de Java, pasando por alto todas las protecciones suministradas por el lenguaje y el compilador de Java que se describió en la sección previa?, para

evitar esa forma de ataque se construyó la segunda línea de defensa, el *verificador de código de bytes* (BCV por sus siglas en inglés).

El BCV sólo permite ejecutar código de bytes de programas Java válidos, buscando intentos de:

- fabricar apuntadores,
- ejecutar instrucciones en código nativo,
- llamar a métodos con parámetros no válidos,
- usar variables antes de inicializarlas,
- etc.

El BCV efectúa cuatro pasadas sobre cada archivo de clase:

- En la primera, se valida el formato del archivo.
- En la segunda, se comprueba que no se crean instancias subclases de clases finales.
- En la tercera, se verifica el código de bytes: la pila, registros, argumentos de métodos.
- En la cuarta, se finaliza el proceso de verificación, realizándose las últimas pruebas

Si el verificador aprueba un archivo .class, se le supone que cumple ya con las siguientes condiciones:

- Acceso a registros y memoria válidos
- No hay desbordamientos de pila
- Consistencia de tipo en parámetros y valores devueltos
- No hay conversiones de tipos ni casteos ilegales

Aunque estas comprobaciones sucesivas deberían garantizar que sólo se ejecutarán applets[6] legales, ¿qué pasaría si un applet cargara una clase propia que reemplazara a otra crítica del sistema, por ejemplo SecurityManager?

Para evitarlo, se erigió la tercera línea defensa, el cargador de clases.

Cargador de clases

A la hora de ejecutarse un applet en la máquina, se consideran tres dominios con diferentes niveles de seguridad:

- La máquina local (el más seguro)
- La red local guardada por el firewall (seguro)
- La Internet (inseguro)

En este contexto, no se permite a una clase de un dominio de seguridad inferior sustituir a otra de un dominio superior, con el fin de evitar que un applet cargue

[6] Un applet es un programa escrito en Java el cual reside en un servidor de internet y se ejecuta en la computadora cliente al conectarse a ese servidor y solicitar el servicio que se ofrece.

una de sus clases para reemplazar una clase crítica del sistema, soslayando así las restricciones de seguridad de esa clase.

Este tipo de ataque se imposibilita asignando un espacio de nombres distinto para clases locales y para clases cargadas de la red. Siempre se accede antes a las clases del sistema, en vez de a clases del mismo nombre cargadas desde un applet.

Además, las clases de un dominio no pueden acceder métodos no públicos de clases de otros dominios.

Aun así, ¿sería posible que algún recurso del sistema resultase fácilmente accesible por cualquier clase?

Para evitarlo, se creó la cuarta línea de defensa, el gestor de seguridad.

Gestor de seguridad

La gestión de seguridad la realiza la clase abstracta "SecurityManager", que limita lo que los applets pueden hacer. Para prevenir que ésta sea modificada por un applet malicioso, no puede ser heredada por un applet.

Entre sus funciones de vigilancia, se encuentran el asegurar que los applets no accedan al sistema de archivo, no abran conexiones a través de internet, etc.

Para ello se introduce una serie de restricciones de seguridad que se detallan en las siguientes secciones.

Restricciones de seguridad

En primer lugar, es importante hacer notar que las restricciones son distintas para cada navegador, ya que cada uno ofrece su propia política de seguridad. La política de Netscape Navigator, por ejemplo, es mucho más restrictiva que la de HotJava de Sun.

En cualquier caso, la mayoría de los navegadores presentarán las siguientes restricciones a la hora de ejecutar applets:

- No se puede trabajar con el sistema de archivos: leer, escribir, borrar, renombrar, listar, conseguir información, ejecutar programas, etc.
- No se pueden establecer conexiones de red a máquinas distintas que la que envió el applet.
- No se permite acceso al sistema.
- No se permite manipulación de hilos.
- No se pueden cargar métodos nativos.
- No se pueden evitar mensajes de alerta en las ventanas creadas por el applet.

Para abreviar, todas las restricciones anteriores pueden reducirse a dos reglas:

- No se puede trabajar con archivos en la máquina del usuario a menos que éste lo permita
- No se puede conectar a nada en la red más que a la máquina que envió el applet

En la práctica, estas restricciones son tan estrictas para los applets descargados remotamente que sólo pueden acceder a los recursos muy limitados disponibles dentro del "patio de juegos".

Restricción de acceso al sistema de archivos

No cuesta trabajo comprender por qué se le niega el acceso al sistema de archivos a cualquier applet desconocido descargado desde la red (posiblemente sin ni siquiera nuestro conocimiento, cuánto menos nuestro consentimiento).

Si no existiera esta restricción, los applets plantearían los siguientes problemas:

- Acceso a información de seguridad: como por ejemplo el archivo /etc/passwd de UNIX.
- Acceso a información privada: correo, documentos, etc., almacenados en nuestro disco duro.
- Si los applets pudieran escribir ficheros, podrían inocular virus (especialmente caballos de Troya, como aplicaciones aparentemente de utilidad, pero que abren la puerta a problemas).
- Al escribir un archivo se puede llegar a llenar el disco, causando denegación de servicio.
- Se podrían modificar los archivos, cifrarlos o destruirlos.
- Se podría crear archivos de engaño.
- Se podría conocer la estructura de directorios puede revelar información e identificar debilidades acerca del sistema (especialmente en UNIX):
    - Identificar nombres de programas CGI ocultos.
    - Identificar debilidades de seguridad.
    - Revelar archivos .rhost.
    - Etc.

  Desgraciadamente, existen métodos subrepticios para explorar la estructura de directorios.

Restricción de conexiones de red

Si los applets fueran capaces de establecer conexiones indiscriminadamente, se podrían producir los siguientes tipos de ataque:

- Ataques desde dentro de un firewall: el applet hostil podría descargarse desde una inocente página web, pasando los controles del firewall, y una vez dentro, hacerse pasar por la máquina que la ha descargado para establecer conexiones o recibirlas.
- Suplantación de usuario, explotando la confianza de otras máquinas: una vez que el usuario incauto ha descargado inadvertidamente el applet y se encuentra en ejecución en su sistema, el applet podría

comunicarse con otras máquinas usando la dirección IP de la máquina del usuario y enviar por ejemplo correos ofensivos.
- Escucha de puertos que suplanten a un servicio (por ejemplo servidor web, escuchando en el puerto 80 y respondiendo a todas las peticiones de páginas con sus propias páginas), algo terrible para el sitio web suplantado.

Para evitarlo, el gestor de seguridad se cuida de comprobar todos los intentos de conexión. En concreto, la clase "SecurityManager" define los siguientes métodos:

Método	***Descripción***
checkAccept(String, int)	Se llama antes de aceptar una conexión con un socket de la máquina especificada en el puerto especificado.
checkConnect(String, int)	Se llama antes de abrir una conexión con un socket a la máquina especificada en el puerto especificado.
checkConnect(String, int, Object)	Se llama antes de abrir una conexión con un socket a la máquina especificada en el puerto especificado para el contexto de seguridad actual.
checkListen(int)	Se llama para determinar si el proceso en curso tiene permiso para quedarse a la escucha de conexiones en el puerto especificado.
checkSetFactory()	Determina si el proceso en curso tiene permiso para establecer la factoría de manejadores de socket o streams de URL.

Restricción de acceso al sistema

A las applets se les niega la capacidad de ejecutar programas en la máquina local y de acceder a propiedades del sistema, ya que estas capacidades podrían usarse para robo o destrucción de información o denegación de servicio:

- Terminar la sesión del navegador (simplemente llamando a System.exit()).
- Ejecutar comandos que manipulen los archivos u otros servicios.
- Conocer información privada del sistema o que puede revelar debilidades.

Restricción de manipulación de hilos

- Las applets podrían destruir trabajo cerrando o deshabilitando otros componentes de las aplicaciones en que corren.
- Ataques de denegación de servicio, aumentando su propia prioridad.

Restricción de acceso a métodos nativos

Se conoce como métodos nativos a código escrito en otro lenguaje distinto de Java, como en C o en ensamblador. Las bibliotecas de métodos nativos, por un lado, resultan de crucial importancia para extender la funcionalidad base de la máquina virtual de Java; por otro, constituyen la última puerta trasera de

entrada, ya que pueden sortear los mecanismos de seguridad de Java, aunque no tienen por qué.

Por este motivo, cargar una biblioteca de métodos nativos podría comprometer la seguridad de todo el sistema, ya que al no estar escritos en Java, no necesitan respetar las reglas de protección de Java y ni siquiera se les exige (aunque lo deberían hacer) el usar las clases existentes. Así, pueden acceder a los recursos locales del sistema, bien proveyendo acceso a nuevos recursos no contemplados en Java y sin protegerlos adecuadamente, bien burlando las comprobaciones de seguridad de Java.

En primer lugar, se debe estudiar si realmente resulta necesario implementar una biblioteca de métodos nativos. En caso afirmativo, tanto porque Java no esté capacitado para realizar la tarea, como por baja eficiencia en Java (como en rutinas matemáticas), si su uso está justificado, es importante integrar dicha biblioteca en el modelo de seguridad del lenguaje.

Para ello, en segundo lugar, se deben identificar los recursos sensibles que podrían resultar expuestos por la biblioteca. Después, se puede diseñar una interfaz de seguridad para la misma e implementar una política de seguridad conservadora.

Desgraciadamente, estas tareas sólo están al alcance de programadores muy avanzados.

Restricción de creación de ventanas

Es evidente la razón por la cual a las ventanas creadas desde un applet se le añaden una serie de advertencias acerca de su inseguridad. Si no existiese ninguna señal de alerta, podrían suplantar ventanas de aplicaciones locales de confianza, como la conocida ventana de Windows 95 ó NT que piden el nombre de usuario y la contraseña.

A pesar de todo el trabajo que se han tomado, existen maneras de engañar a Java, como en las típicas ventanas maliciosas.

Ataques comunes en Java

Existen muchos tipos de ataques a la seguridad que podrían organizarse contra un sistema de computadores. Aunque el usuario medio podría ignorar que tales amenazas existen, lo cierto es que los ataques suceden, a veces con consecuencias devastadoras. A continuación se verá cuáles son los principales ataques y cómo Java se defiende (a veces sin éxito) contra ellos.

Entre los ataques más comunes se citan:

- Robo de información
- Destrucción de información
- Robo de recursos
- Denegación de servicio
- Enmascaramiento

- Engaño

Otras clasificaciones de ataques que se pueden encontrar en la literatura más relacionada con Java son las siguientes:

- Modificación o alteración de un sistema o sus recursos.
- Denegación del uso legítimo de los recursos de la máquina.
- Ataques a la intimidad del individuo.
- Simple molestia.

No resulta complicado establecer una correspondencia entre ambas clasificaciones.

Robo de información

En principio, todas las computadoras contienen alguna información de interés para alguien. Es cierto que no siempre tendrá el mismo valor, pero siempre puede existir alguien interesado en conseguirla. Por consiguiente, uno de los ataques más comunes está dirigido a extraer información confidencial de un sistema.

Con el fin de que ningún applet de Java pudiera hacerlo, se decidió que no tendrían acceso al sistema de archivos, lo que significa que no se puede leer, escribir, borrar, renombrar, o listar archivos. Además se les negó el acceso al sistema, para que no pudieran ejecutar comandos que manipulen archivos (como el famoso rm -rf / de UNIX) ni acceder a propiedades del sistema que contengan información confidencial (nombres de usuario, contraseñas, versiones de programas, etc.) o que pudiera revelar debilidades.

Desgraciadamente, existen métodos indirectos que permiten explorar los contenidos del disco duro.

Destrucción de información

Todo el mundo posee datos y archivos en su computadora que no quiere perder. Por esta razón es muy importante que ningún applet sea capaz de modificar ni mucho menos destruir los datos de la máquina cliente en la que se ejecuta.

Los diseñadores de Java introdujeron una serie de controles de seguridad que impiden el acceso al disco del equipo cliente con esto se garantiza que no se pueda manipular o destruir la información desde un applet. Estas funciones son:

Método	*Descripción*
checkDelete(String)	Comprueba si se puede borrar el archivo de nombre String.
checkRead (FileDescriptor)	Comprueba si se puede leer el archivo indicado por FileDescriptor.
checkRead(String)	Comprueba si se puede leer el archivo de nombre String.
checkRead(String,	Comprueba si se puede leer el archivo en el contexto de

Object)	seguridad actual.
checkWrite (FileDescriptor)	Comprueba si se puede escribir el archivo indicado por FileDescriptor.
checkWrite(String)	Comprueba si se puede escribir el archivo de nombre String.
checkFileDialog()	Las applets no pueden abrir cuadros de diálogo de archivos.

Robo de recursos

Las computadoras, además de almacenar datos, poseen valiosos recursos como espacio en disco y en memoria, ciclos de CPU, capacidad de cálculo, etc. Un applet descargado silenciosamente desde una página web podría estar funcionando en modo oculto, sin hacer ruido, realizando pesados cálculos y enviando los resultados al servidor desde el que se bajó, sin siquiera darse cuenta.

Es éste el aspecto más difícil de controlar en Java, ya que las applets pueden hacer uso ilimitado de ciclos de CPU, en algunos casos disminuyendo el funcionamiento de la máquina y generando pérdida de tiempo.

Se trata en definitiva de un problema sin fácil solución.

Denegación de servicio

En general, estos ataques se basan en utilizar la mayor cantidad posible de recursos del sistema objetivo, de manera que nadie más pueda usarlos, perjudicando así seriamente la actuación del sistema, especialmente si debe dar servicio a muchos usuarios.

Ejemplos típicos de este ataque son:

- El consumo de memoria de la máquina víctima, hasta que se produce un error general en el sistema por falta de memoria, lo que la deja fuera de servicio.
- El consumo de ciclos de CPU, de forma que el resto de procesos que corren en la misma máquina apenas dispongan de tiempo de ejecución, por lo que su rendimiento decae rápidamente.
- La apertura de cientos o miles de ventanas, con el fin de que se pierda el foco del ratón y del teclado, de manera que la máquina ya no responda a pulsaciones de teclas o de los botones del ratón, siendo así totalmente inutilizada.
- El monopolio de algún periférico, de manera que nadie más pueda acceder a él, obligando así a desconectarlo o a terminar con alguno de los procesos que corren en la máquina.

Es obvio que en máquinas que deban funcionar ininterrumpidamente, cualquier interrupción en su servicio por ataques de este tipo puede acarrear consecuencias desastrosas. Desgraciadamente, Java no posee ninguna política estricta que permita evitar estos ataques, ya que resulta muy complicado trazar la frontera en la cantidad de recursos que un applet puede reservar para sí.

Enmascaramiento

Los applets de Java proporcionan a cualquier persona que escriba páginas web la capacidad de hacerse pasar por otra persona a la hora de enviar correos (*spoofing*). Todos los usuarios de UNIX saben que una forma muy sencilla de falsificar correos electrónicos es hablar directamente con el programa de envío de correo (*sendmail*) a través del puerto 25 (correspondiente al protocolo SMTP).

Un applet de Java es capaz de abrir un socket a dicho puerto 25 y a partir de ahí, seguir todos los pasos del protocolo y enviar así un correo falso.

Engaño

Otro ataque muy común consiste en engañar a una persona para que facilite información confidencial. Un ejemplo típico es el que se conoce como de *ingeniería social*: hacerse pasar por el administrador de una red o servicio y pedir a los usuarios incautos que envíen su login y password para alguna labor de actualización, seguramente que un porcentaje elevado envía ciegamente sus datos.

Una forma más sofisticada de engaño consiste en falsificar ventanas del sistema, para obligar al usuario a que ejecute una acción que de otro modo no realizaría, como reiniciar la computadora (perdiendo los datos) o suministrar su login y password.

Agujeros tipo Línea Maginot

Vale la pena hacer un poco de historia. Previo a la Segunda Guerra Mundial, los franceses construyeron una barrera de defensa a todo lo largo de su frontera con Alemania, que con la tecnología bélica de su época era imposible de franquear.

La bautizaron con el nombre de línea Maginot. Lo que ocurrió fue que los alemanes, ya en la guerra, invadieron Holanda y desde ahí pasaron a Bélgica, aliado de Francia y que no le iba a traicionar. Sin embargo, por la fuerza, los alemanes atacaron Francia a través de la frontera de Francia con Bélgica, por lo que la línea Maginot no sirvió de nada.

La moraleja de la historia es que a veces se espera al enemigo en un frente y éste es reforzado y defendido al máximo posible, mientras el enemigo ataca desde otro frente que se consideraba de confianza.

Evolución del modelo de seguridad de Java

En vista de las restricciones impuestas por el modelo de seguridad de Java, muchos desarrolladores de software se encuentran ante la impotencia de escribir aplicaciones realmente útiles, puesto que se les niega la capacidad de leer/escribir en disco o de abrir conexiones a otras máquinas.

Si se quiere que las applets sean algo más que dibujos animados para realzar el impacto visual de una página y lleguen a convertirse en verdaderas

aplicaciones productivas y realmente útiles, los usuarios de la Aplicación Java necesitan garantizarle el acceso a los recursos, y antes de concederle estos privilegios, necesitan de un mecanismo que les permita saber de donde y de quién proviene el applet al que le van a permitir acceso a su sistema.

La evolución lógica del modelo anterior llegó a través del JDK 1.1.x, al introducir el concepto de applet firmado. En este modelo extendido, un applet con firma digital válida se trata como si fuera código de confianza cargado localmente siempre y cuando la firma digital sea reconocida como de confianza por el sistema final que recibe el applet. Estos applets firmados, junto con sus firmas, son enviadas en formato JAR (Java Archive).

Nueva arquitectura de seguridad de Java

La arquitectura de seguridad del JDK 1.2 se introdujo con el fin de superar limitaciones del modelo anterior:

- Control de acceso de grano fino: libera al programador de la carga de extender y adaptar las clases SecurityManager y ClassLoader.
- Política de seguridad fácilmente configurable: permite a los desarrolladores y usuarios configurar políticas de seguridad sin necesidad de programar una línea.
- Estructura de control de acceso fácilmente extensible: acaba con la necesidad de añadir nuevos métodos a la clase SecurityManager para crear nuevos permisos de acceso, ya que permite hacerlo automáticamente.
- Extensión de las comprobaciones de seguridad a todos los programas en Java, incluyendo tanto Aplicaciones como applets: el código local ya no es de confianza, sino que también está sujeto a los controles de seguridad, aunque estas políticas pueden flexibilizarse.

La piedra angular de esta nueva arquitectura la constituye el dominio de protección, mecanismo para agrupar y aislar unidades de protección, a los cuales se les asigna una serie de permisos de acceso a recursos. De esta manera, cada clase pertenece a un único dominio, que tendrá sus permisos asignados. Antes de acceder a un objeto, se comprueba la política para ver qué permisos tiene y si puede accederlo.

Entre otras características añade soporte para SSL v3.0 y una nueva extensión criptográfica para Java. La verificación de firmas digitales soporta completamente el procesamiento de los certificados X.509v3.

Recomendaciones de Seguridad en Java

A continuación se listará una serie de recomendaciones para los usuarios que van desde usuarios de aplicaciones Java hasta desarrolladores en Java. Estas recomendaciones pueden atenderse o no y su uso no es una garantía de que la seguridad no será violada, sin embargo pueden mitigar las posibilidades de que se viole la seguridad en distintos niveles.

- Navegar siempre con la última versión de software.
- Desactivar Java y conectarlo (si es necesario) sólo en aquellos sitios web que sean confiables.
- Conocer el entorno Java del navegador utilizado, para evitar que esté manipulado.
- Utilizar compiladores de fuentes fiables.
- Utilizar bibliotecas de Java de fuentes fiables.
- Utilizar ofuscadores de código para ocultar el código fuente.
- Navegar anónimamente.
- Mantener copias de seguridad de todo lo que sea de importancia.
- Nunca ejecutar el navegador desde una máquina que almacene información sensible o sea crítica para el sistema, como un servidor, por ejemplo.

Bibliografía utilizada en este capítulo

Java Security (2nd Edition)
Scott Oaks
O'Reilly

Hacking Exposed (Second Edition)
Joel Scambray, Stuart McClure, George Kurtz
McGraw Hill

Capítulo IV.- XML para el intercambio de información

XML (eXtensible Markup Language) ha surgido como uno de los formatos de intercambio de información más aceptado hoy en día, inclusive en ocasiones es designado: "El ASCII de internet"

Esta designación se debe a la flexibilidad y uniformidad con que puede ser intercambiada Información, desde transacciones financieras, aplicaciones inalámbricas hasta aplicaciones de servidor, todo partiendo de *un solo documento maestro* con la confiabilidad de que el formato sea *ampliamente aceptado*, esto hace que XML sea una Tecnología de Vanguardia.

Intercambio de Información

El intercambio de Información siempre ha sido un grave problema cuando se utilizan lenguajes y sistemas operativos incompatibles, esto se ha dado desde los niveles de procesadores de texto hasta la utilización de bases de datos que utilizan variaciones sobre el Structured Query Language o SQL.

Este problema se agudizó con la aparición de internet; donde antes era posible limitar o controlar la utilización de diferentes sistemas para intercambiar información, con internet ya no lo es; inclusive antes de la aparición de internet se crearon mecanismos para lograr el intercambio *fluido* de información entre diferentes sistemas, el primer método fue GML, posteriormente SGML y actualmente *XML*, todos estos mecanismos son llamados *lenguajes de marcación* o *meta-lenguajes*

GML y SGML

GML ("General Markup Language") fue uno de los primeros lenguajes de marcación que fue diseñado para componer estructuras de datos descriptivas, esto es, un *meta-lenguaje*, estructuras de datos describiendo otras estructuras de datos.

GML eventualmente se convirtió en SGML ("Standard Generalized Markup Language") y fue en 1986 que fue adoptado como un *"Estándar Internacional para el Intercambio y Almacenaje de Información"*, identificado por ISO 8879. Aunque SGML fue adoptado y aún es utilizado en varios proyectos por ser un *lenguaje de marcación* muy poderoso, su forma es un tanto *compleja* y por consiguiente *costosa de desarrollar.*

Este lenguaje de marcación (SGML) ha encontrado uso en proyectos gubernamentales, industrias manufactureras, publicista, etc. e inclusive es utilizado todos los días por navegadores como MS-Explorer y Netscape.

Uso de SGML en Internet Explorer y Netscape.

Toda la información que recibe un navegador en aplicaciones cliente generalmente es HTML, el detalle es que HTML *es parte* de SGML. Lo siguiente es parte de un documento HTML:

```
<HTML>
<HEAD>
<TITLE> Documento Básico en HTML
</TITLE>
</HEAD>
<BODY>
<H2> Este es el Titulo </H2>

<!-- Este es un comentario que no
aparecerá desplegado-->

<a href=mailto:daniel@osmosislatina.com>
Enviame un Mail </a>

</BODY>
</HTML>
```

Si se observa detalladamente y como su nombre lo indica HTML *también* es un *lenguaje de marcación*: estructuras de datos describiendo otras estructuras de datos, los parámetros: <TITLE> <H2> <HTML> <HEAD> *describen* otras estructuras de datos: "Documento Básico en HTML" es el título, sin embargo HTML proviene de algo más global que es precisamente SGML.

Todo navegador contiene al menos tres DTD[7] para HTML, éstos son *Strict* el cual incluye todos los elementos que no han sido deprecados, *Transitional* que incluye a los elementos contenidos en el DTD strict más los deprecados y *Frameset* que incluye a los elementos contenidos en el DTD transitional mas los frames. Estos DTD indican como debe ser *desplegada* la información en el navegador, esto es, definen que <TITLE> es el título del documento, <H2> es un encabezado, etc.

Estos **DTD's forman parte primordial de SGML**, y son estos DTD los que *realmente* van conformando el estándar HTML, el estándar HTML 4.0 está compuesto por ciertos DTD. Inclusive muchos navegadores están equipados con DTD's propietarios, lo cual hace que ciertos elementos de HTML sean incompatibles con otros navegadores.

Si se deseara intercambiar *información* con una empresa en Chile y otra en Holanda, seguramente se tendría que acordar un estándar. Para asegurarse que esta *información* fuera reutilizable independientemente del sistema operativo o Aplicación seguramente se utilizaría un estándar internacional como SGML.

[7] Data Type Definition

Sin embargo, SGML es tan complejo de implementar que seguramente en un trabajo pequeño sus costos excederían sus beneficios, por esto en 1996 el " World Wide Web Consortium" inició trabajos sobre XML, un estándar más simplificado.

XML Lenguaje de Marcación

La importancia de *XML* radica en la facilidad para el intercambio de información, ya que es posible utilizar este *lenguaje de marcación* para generar así como distribuir información que sea utilizada por transacciones financieras, aplicaciones inalámbricas, bases de datos, impresoras, etc.

La principal ventaja que presenta este lenguaje es su independencia de sistema operativo y aplicación que será capaz de utilizarlo, esto es, se puede tener un documento escrito en XML y puede ser manipulado en los sistemas operativos: Sun Solaris, Windows, AIX o en un ambiente Java, VBScript, PL/SQL.

Es esta facilidad de intercambio de información por la que ha sido adoptado XML en desarrollos B2B o Business to Business.

Debido a que XML fue desarrollado a partir de SGML eliminando un gran número de funcionalidades que lo hacían extremadamente complejo, XML aun mantiene una similitud con SGML, un ejemplo:

```
<producto>
<nombre> Bocinas </nombre>
<modelo> XJ-246432 </modelo>
<precio> $123.25 </precio>
<disponibilidad> Si </disponibilidad>
</producto>
```

XML también utiliza *tags* como SGML/HTML, sin embargo, a diferencia de HTML que ya posee DTD's específicos, en XML es posible describir *información general* como productos, descripciones, nombres, etc., los cuales son denominados *vocabularios*.

Hoy en día ya han sido definidos varios *vocabularios* (DTD's) que definen este tipo de *tags* basándose en industrias, de manera que de la misma forma en que ya existe un estándar para *tags* de presentación (HTML), varias industrias han empezado a definir sus propios *tags*: Industria química CML "Chemical Markup Language", Industria legal XFDL "eXtensible Forms Description Language" y otras.

Terminología en XML

Ésta es parte de la terminología utilizada en XML.

DTD's (Data Type Definition o Document Type Definition): Definen como serán utilizados e interpretados los elementos de un documento XML, esto es, si se utiliza un *tag* como <nombre> o <apellido>, los DTD's definen entre otras cosas: Que tan extenso puede ser su valor, el tipo de carácter (UTF-8,UTF-16, etc.), reglas que deben cumplirse en la información (ser parte de otro TAG, valores específicos, etc.), referencias a otros DTD's.

A su vez estos DTD's son utilizados al procesar un documento XML (vía DOM, SAX, JDOM) para *validar el contenido del mismo*, esto es, si al procesar el documento se encuentra que éste no coincide con las definiciones del DTD, se debe generar un error por parte del analizador DOM, SAX, JDOM.

DOM (Document Object Model): Es una *especificación* desarrollada por el World Wide Web Consortium que *define* como *procesar* documentos en XML. Se debe hacer énfasis que DOM es *solo una especificación*, esto implica que existen diversas *implementaciones* de DOM.

JAXP (Java API for XML Processing): Es una iniciativa de Sun para uniformizar el desarrollo de aplicaciones Java con XML, es muy importante señalar que JAXP *no es un analizador*, sino que JAXP funciona *en conjunción con un analizador*.

Lo que se intenta lograr mediante JAXP es interoperabilidad entre los diferentes analizadores que existen en el mercado.

Namespaces : Mediante Namespaces es posible mezclar diversos elementos (*vocabularios*) que pudieran prestarse a confusión, esto es, suponer que se está generando una aplicación para la industria química y se requiere usar diversos DTD que contienen distintas definiciones para el elemento <*papel*> y requiere utilizar todas éstas, mediante el uso de Namespaces y schemas es posible eliminar la ambigüedad que pueda surgir al referirse al elemento <*papel*> dentro del programa o aplicación.

SAX (Simple API for XML): Como DOM es solo una especificación, pero a diferencia de éste, SAX ofrece mayor sencillez para manipular y procesar información en XML.

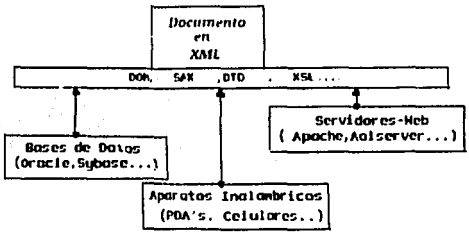
Schemas : Han surgido como una alternativa a los DTD's utilizados para validar información en XML, a diferencia de DTD's el utilizar schemas permite definir los elementos de validación en XML directamente y la utilización de Namespaces.

TrAX (Transformation API for XML): Es una especificación muy reciente que *formará parte* de JAXP (versión 1.1), en si TrAX *extiende* el funcionamiento de JAXP. Esta extensión JAXP surgió como una solución para permitir interoperabilidad en las diversas *implementaciones de analizadores en XML*, la intención de TrAX es permitir la interoperabilidad de las distintas máquinas *XSL*.

XSL | XSLT (Extensible Stylesheet Language): Es un *lenguaje* derivado de XML que permite *transformar y manipular* documentos en XML. Transformar y manipular documentos XML se puede hacer con DOM, sin embargo, XSL lo permite a través de *formatos*, permitiendo manipular documentos de XML a HTML.

XSL funciona con un analizador como DOM, SAX o JDOM, este software comúnmente llamado **XSL engine** *ya incluye* un analizador en su estructura.

Para poder intercambiar cualquier documento XML es necesario que el receptor conozca la definición del *DTD*, basándose en la siguiente gráfica:



El proceso que lleva a cabo el documento es el siguiente:

- Definir el documento y su *DTD* o *schema*.
- Se procesa el documento vía DOM (Document Object Model), SAX (Simple API for XML) o JDOM.
- Se aplica un XSL o eXtensible Stylesheet Language.

Estos tres pasos son los que garantizan que el mismo documento escrito en XML pueda ser llevado de un punto a otro sin ningún conflicto. Las posibilidades que pueden surgir a partir de esto son interminables, a continuación algunas de las más utilizadas.

Desarrollo de Sitios para diferentes Clientes

Es común el desarrollo de aplicaciones donde seguramente aparece algún conflicto con el tipo de navegadores que visitan una página, los ejemplos más claros de estos conflictos están en la utilización de frames en HTML o JavaScript.

Para resolver estos problemas una de las soluciones (si es que puede ser llamada solución) más comunes es recomendar al usuario final cual navegador (Netscape 4.x, Explorer 5.0) utilizar para experimentar el sitio de una manera apropiada.

Otra manera más apropiada es mediante la revisión de las cabeceras enviadas en la requisición HTTP del cliente, esta revisión de cabeceras es realizada por el servidor de páginas y una vez conocido el navegador que está solicitando la información, ésta se envía.

Una desventaja de este método es que deben existir tantas versiones navegadores sean esperados, esto es, si la información generada en la aplicación de servidor (ASP, ADP, JSP, etc.) o aplicación de cliente (JavaScript, Java, etc.) afecta como será desplegada en el cliente (Netscape o MS-Explorer) deben ser definidas diferentes versiones para cada tipo de navegador.

Utilizando XML es posible definir un solo documento global con *TAGS* generales y definir varios *XSL* (Extensible StyLe sheet's) para cada navegador que sea esperado. Aparentemente es la misma solución que fue descrita anteriormente, pero no lo es así.

Debido a que existe un solo documento que define los datos es posible concentrar los esfuerzos de administración y programación en este solo documento maestro (XML), para enviar información a los diferentes navegadores es necesario crear *XSL's* lo cual es una labor mucho más sencilla que aquella definida anteriormente, estos *XSL's* definen que tipos de datos del documento global (XML) deben ser enviados al navegador: Sea Netscape, MS-Explorer, un aparato inalámbrico, PDA u otro, además debe recordarse que como ya posee la información en XML, el día que sea necesario ofrecerle esta información a un proveedor o cliente vía cualquier otro medio, estará disponible de una manera inmediata a diferencia que ésta estuviera en HTML.

DOM,SAX o JDOM

Para ahondar en los conceptos DOM, SAX o JDOM es necesario recordar un poco lo que es manipulación de Datos en un sistema de Información.

Un Archivo de Texto

```
Daniel:Rubio:Ensenada:Mexico:25
Fernanda:Fontes:Macae:Brasil:27
Luis:Arano:Mendoza:Argentina:27
Hrvoje:Galic:Zagreb:Croacia:28
```

El manipular el archivo plano anterior ya sea para actualizarlo o agregar una línea tiene sus limitaciones, esta situación se agrava si desea intercambiar archivos de este tipo a otros sistemas de cómputo o empresas, ya que no existe ninguna indicación sobre el significado de cada elemento, esto es, ¿el tercer elemento es país o ciudad?, ¿la división de elementos siempre será ":"?, etc. La solución es generar un documento *descriptivo*, por tanto, en XML

```
<Amigos>
   <Nombre> Daniel</Nombre>
   <Apellido> Rubio</Apellido>
   <Ciudad> Ensenada</Ciudad>
   <País> México</País>
   <Edad> 25</Edad>

   <Nombre> Fernanda </Nombre>
   <Apellido> Fontes</Apellido>
   <Ciudad> Macae </Ciudad>
   <País> Brasil </País>
   <Edad> 27 </Edad>

   <Nombre> Luis  </Nombre>
   <Apellido> Arano</Apellido>
   <Ciudad> Mendoza </Ciudad>
   <País> Argentina </País>
   <Edad> 27 </Edad>

   <Nombre> Hrvoje  </Nombre>
   <Apellido> Galic </Apellido>
   <Ciudad> Zagreb </Ciudad>
   <País> Croacia </País>
   <Edad> 28 </Edad>
</Amigos>
```

Como ya fue mencionado, XML está compuesto por elementos *descriptivos* (Meta lenguaje) con sus valores correspondientes (Nombre, Apellido, Ciudad, etc.) esto facilita el uso de este archivo a otros sistemas de cómputo o empresas, ahora bien, la pregunta que continuaría es: Ya se tiene la información en formato XML, ¿cómo se utiliza o traslada esta información a un programa en Java o una aplicación de servidor?.

Esto se hace mediante un analizador *DOM, SAX o JDOM*

DOM

DOM ("Document Object Model") es solamente una *especificación* definida por el "World Wide Web Consortium", muy similar al J2EE de Sun, ya que permite a diversas compañías u organizaciones definir analizadores alrededor de esta especificación. La especificación DOM más reciente es 2.0 y la gran mayoría de los analizadores DOM disponibles ya cumplen con ella.

Entonces es inmediata la pregunta ¿Cual es la diferencia entre manipular información con DOM o simplemente abriendo el archivo con un manejador de texto?.

DOM genera un árbol jerárquico en memoria del documento o información en XML, basándose en el documento anterior cada elemento <Amigos>, <Nombre>, <Apellido>, etc. es considerado un nodo dentro del árbol.

Este árbol jerárquico de información en memoria permite que a través del analizador sea manipulada la información, las ventajas son las siguientes:

- Puede ser agregado un *nodo* (información) en cualquier punto del árbol.
- Puede ser eliminada información de un *nodo* en cualquier punto del árbol.
- Lo anterior se ejecuta *sin* las limitaciones de manipular un archivo de alguna otra manera.

Debido a que DOM es solo una *especificación* existen diversos analizadores que lo implementan:

- Xerces
- DOM de Oracle
- DOM para MS-Explorer (el cual permite procesar un documento XML directamente en MS-Explorer, tal y como si fuera una aplicación cliente).

Un detalle notorio de cualquier analizador es que la mayoría están escritos en Java, ésta no es ninguna coincidencia ya que Java es uno de los lenguajes que permite mayor portabilidad entre sistemas operativos.

Ahora bien, a pesar de esta portabilidad en Java, DOM es solo una especificación y por ende existen diversas *implementaciones*, esto lleva a otra pregunta importante ¿si se desarrolla una aplicación que utilice el analizador Xerces, este programa puede ser utilizado posteriormente con un DOM desarrollado en Oracle ?, la respuesta es NO.

El utilizar un analizador implica aprender a utilizar sus clases | funciones o API ("Application Programming Interface"), como el API de Xerces, sin embargo, *ciertas* clases | funciones difieren un poco en los diversos analizadores, por eso se recomienda (si se utiliza Java) que los programas | aplicaciones que requieran de las funcionalidades de un analizador sean diseñadas alrededor de JAXP "Java API for XML Processing".

SAX

SAX ("Simple API for XML") procesa el documento o información en XML de una manera muy diferente a DOM, SAX procesa la información *por eventos*. A diferencia de DOM que genera un *árbol jerárquico en memoria* , SAX procesa la información en XML conforme ésta sea presentada (*evento por evento*), efectivamente manipulando cada elemento a un determinado tiempo, sin incurrir en uso excesivo de memoria. Por lo tanto tiene las siguientes características:

- SAX es un analizador ideal para manipular archivos de gran tamaño, ya que no genera un *árbol en memoria* como es requerido en DOM.

- Es más rápido y sencillo que utilizar DOM

- La sencillez antes mencionada tiene su precio, debido a que SAX funciona *por eventos* no es posible manipular información una vez procesada, en DOM no existe esta limitación ya que se genera el *árbol jerárquico en memoria* y es posible regresar a modificar nodos.

La especificación más reciente de SAX es 2.0, y al igual que DOM 2.0 ésta se incluye en casi todos los analizadores disponibles en el mercado:
- Xerces
- DOM de Oracle
- Parser XP

Coinciden con los analizadores DOM debido a que casi todos incluyen tanto una implementación para DOM como para SAX y dependiendo de su situación tiene la flexibilidad de utilizar DOM o SAX. La frase *"es capaz de utilizar"* hace resurgir un tema ya mencionado: ¿Es posible migrar o adaptar un programa | aplicación a diferentes analizadores como Xerces, Oracle o XP? Si se utiliza Java SI, para esto surgió JAXP.

JDOM

JDOM ("Java DOM") es una representación en Java de un documento XML. JDOM provee una manera de representar un documento dado para su lectura, escritura y manipulación de forma fácil y eficiente. Tiene una API ligera y rápida y orientada al programador en Java. Representa una alternativa a DOM y SAX, aunque se integra bien con ambos.

Bibliografía utilizada en este capítulo

Professional XML
Didier Martin, Mark Birbeck, Michael Kay, Brian Loesgen
Wrox

Capítulo V.- Cliente universal para requerir información vía internet

Con obligatoriedad por ley y por periodos generalmente establecidos, las instituciones gubernamentales de nuestro país requieren información a la ciudadanía. Tal es el caso por ejemplo de:

- Declaraciones Patrimoniales
- Declaraciones de Ingresos
- Declaraciones de Impuestos

La información requerida generalmente debe concentrarse en forma central, quizá previo paso por algunas oficinas regionales que validen o preprocesen ciertas información específica. Siempre la información debe hacerse llegar desde quien la provee hasta quien la solicita. Quien la provee puede ser cualquier persona y quien la solicita comúnmente es una institución que por lo general debe centralizar esta información para su uso.

Los esquemas para recolección incluyen situaciones donde se requiere autenticación, en este caso el individuo requerido acude a un lugar central, o bien situaciones donde el envío de datos se hace en medio magnético, esto último a veces con el apoyo de algunas facilidades de la red global que se conoce como internet.

Enseguida se describirá brevemente la situación actual de los métodos de recolección de información por medio magnético y su principal problemática vigente.

Posteriormente se detallarán las características de la solución propuesta, una aplicación o sistema portable que permita un ***uso efectivo de Internet como vehículo para la requisición masiva de información***.

Las herramientas tecnológicas que soportan la solución que se planteará son las presentadas en los capítulos anteriores.

En el CD anexo se encuentra el sistema para uso abierto esperando sea de utilidad en la solución de situaciones similares.

Método de recolección común

El método que se utiliza comúnmente para requerir información tiene muchas variantes, algunas utilizando internet como vehículo para el envío de los datos.

Existen opciones que mantienen en línea la captura de cuestionarios, asimismo existen opciones que mantienen la misma captura fuera de línea. El criterio utilizado para decidir si una aplicación es en línea o fuera de línea comúnmente es el tiempo que el usuario dedicará al llenado de una forma, es decir, si se habla de información que demandará al usuario al menos un par de horas de

su tiempo muy seguramente convendrá ofrecerle un esquema fuera de línea, pero si solo le tomará cinco minutos lo ideal es un esquema en línea.

La opción fuera de línea incluye la distribución o entrega de una aplicación cliente para ser ejecutada a fin de captar la información y prepararla para su envío posterior.

Esta entrega del sistema es personal o por medio de la red, en este último caso el individuo requerido se obliga a obtener de internet o de una oficina el sistema que le corresponda.

Estos sistemas capturan la información y la preparan para poder ser almacenada en medio magnético o enviada vía internet. Lo anterior es lo fundamental del esquema propuesto, en la siguiente sección se aborda la problemática que existe con esta solución.

Es frecuente encontrarse con formularios muy complicados de llenar o que demandan mucha información, un buen ejemplo son los formularios de declaración de impuestos. La complejidad en el llenado o el volumen de información solicitada deriva en tiempos largos de uso de una aplicación, por consiguiente es impensable que la captura se haga en línea, hoy día ningún esquema razonable puede considerar estar en línea en la red para la captura de formularios complejos, o que piden miles de campos a capturar y ejecutan cálculos aritméticos en función de ciertas reglas de negocio embebidas en el código.

La arquitectura de la solución propuesta en esta tesis se inclina por mantener la captura fuera de línea, se considera inviable mantener la captura en línea si se esperan cuestionarios complejos o extensos.

Problemática actual en el uso de la red como vehículo para requerir información

Se enfocará la atención únicamente en la requisición de declaraciones, ya sea patrimoniales, de impuestos o alguna de este tipo.

Como se citó previamente generalmente se dispone de sistemas fuera de línea para la captura de la información, los cuales al final de su proceso preparan los datos para envío a la institución que los requiera. Se prepara en medio magnético para entrega en persona o archivo para viajar por internet, con elementos de seguridad como el cifrado o la firma de los datos.

Existen problemas serios con esta práctica, que hacen que el uso de internet sea pobre y se ocasione un sentimiento de estorbo mas que de ayuda. Las principales deficiencias en esta práctica se describen en las secciones posteriores a continuación.

Múltiples versiones

No existe un único sistema para la captura de la información, deben existir tantas versiones del mismo sistema como ambientes de trabajo existen. Es

decir versiones de MS-Windows y su idioma, software de trabajo (como Office), UNIX, Linux, etc.

Aquí el principal problema es para las áreas de desarrollo y mantenimiento de sistemas pues se deben mantener muchas versiones del mismo sistema. Mucho se hace si se cubren el ambiente MS-Windows y sus variantes y combinaciones, dejando fuera otros sistemas operativos importantes. Aunque en el mayor de los casos ni siquiera las variantes básicas de Windows son cubiertas.

Además, al usuario de los sistemas se le obliga a conocer las características de su ambiente y a bajar de internet la versión que le funcionará, provocando con ello molestias adicionales al usuario, a quién de por sí le resulta molesto el cumplimiento de obligaciones.

Una variedad de este problema se encuentra al momento en que formularios diferentes pero de la misma institución requirente se atienden con sistemas diferentes, seguramente las interfaces no serán consistentes y el usuario final debe lidiar con interfaces de usuario distintas, manuales distintos, estilos de programación distintos, etc.

Saturación de sitios

Es usual que los formularios sufran con frecuencia modificaciones, entonces el sistema debe cambiar y el usuario debe bajar de internet la nueva versión, es común por ejemplo que las requisiciones obligatorias tengan fechas límite, entonces previo a tales fechas los usuarios saturan los anchos de banda de la red para disponer del sistema necesario que se encuentra en algún sitio. Aunque es un problema generado por nuestra cultura finalmente es una molestia al individuo requerido.

Base de datos

Los sistemas de captura generalmente suponen la existencia de cierta base de datos comercial como MS-Access como repositorio de la información de manera local, esto obliga a dotar al sistema de aditamentos del constructor de esta base de datos (librería de acceso dinámico de Microsoft en el ejemplo) para el correcto funcionamiento del sistema.

Como consecuencia de lo anterior el tamaño de las aplicaciones es comúnmente de más de 10 megabytes. El uso del ancho de banda de la institución se degrada aun más y el usuario no está satisfecho al tener que bajar un archivo de tal tamaño para actualizar su sistema.

Actualización del sistema

Por errores o por cambio en los formularios se vuelve necesaria la redistribución de la aplicación. También ocurre frecuentemente que al usuario que no usa internet se le obliga a acudir por la actualización de sistema a determinado lugar, y esto es cada vez que se hay una actualización. Es muy frecuente la actualización provocada por cambios en los formularios, mas aún en los referentes a impuestos.

En resumen, actualmente para resolver este problema se está usando internet y en general la tecnología en forma caótica y deficiente. Esto es cierto al menos en algunas instituciones importantes como la encargada de la recaudación de impuestos en el país donde internet únicamente es un puente ineficiente.

Uso óptimo de la red para requerir información

Un sistema que involucre internet en su ambiente de trabajo y que pretenda recabar masivamente información y que se proponga ser una opción viable deberá tener las siguientes características principales:

Parametrizable

Las reglas de negocio de los formularios deben ser independientes a la aplicación, por lo que cualquier cambio en las mismas debe mantener al sistema sin modificación.

Ligero

El sistema deberá ser de tamaño pequeño, pensar en decenas o cientos de kilobytes y nunca en megabytes. Solo así es una opción viable en la red.

Multiplataforma

Un solo sistema para cualquier ambiente. El contar con esta característica resuelve gran parte de la problemática antes citada. Al corto plazo otorga una interfaz única y para siempre al usuario.

Autosuficiente

El sistema no debe tener dependencias de software adicional, como una base de datos comercial para almacenar localmente la información.

Seguro

Tanto en la calidad de los datos, a través de validaciones y analizadores de fórmulas aritméticas complejas, como en la confidencialidad y autenticidad de los mismos.

Múltiples formatos de salida

Debe soportar diferentes formas de salida, como una impresión, o medio magnético para entrega vía internet, más aún debe soportar diferentes estándares de datos, como archivos planos cifrados o archivos en formato XML.

Compatible para formularios en diferentes versiones del aplicativo

Debe mantener la compatibilidad en las estructuras de datos entre diferentes versiones para así evitar al usuario la doble captura de datos constantes y reduciendo con ello la probabilidad de errores imputables a la captura de la información.

Un sistema con estas siete características inevitablemente debe contar de dos partes, una para construcción de formularios a manera de componentes y otra para su interpretación. El constructor de formularios deberá trabajar en forma centralizada por un área responsable de la generación de los mismos, experta en el negocio y en el uso de la aplicación, pero no en programación, y el intérprete en manos del usuario requerido, experto en nada. El sistema deberá ser distribuido por internet para trabajar fuera de línea y preparar la información que posteriormente viajará.

No es tarea fácil obtener estas siete características, además de las básicas de todo sistema, como su facilidad de uso por ejemplo. La dificultad es clara y para enfrentarla se cuenta con el apoyo de los conceptos y herramientas expuestos en los capítulos I al IV.

En las secciones siguientes de detalla un sistema portable cuya función es la recolección masiva de Información.

Cómo recabar Información masivamente con apoyo de la computadora

Se llamará **formulario** a todo formato que sea utilizado para recabar información, que contiene incluso todas aquellas reglas de validación o vinculación sin importar el número de páginas incluidas, es decir puede ser de longitud dinámica.

Existen diversas reglas de validación o vinculación, las más comunes son:

- *Tipo de dato, como números, caracteres, etc.*
- *Validar dato contra catálogo, por ejemplo códigos postales o marcas de vehículos.*
- *Datos que no se capturan, son el resultado de operar aritméticamente con otros que se capturan.*
- *Rangos que se validan, edades mayor a cero por ejemplo o tasa que no deben sobrepasar 10%.*
- *Alineación de los datos en pantalla.*
- *Obtención de totales a partir de información repetitiva.*
- *Etcétera.*

Entonces, dado un formulario y sus reglas de validación la solución típica para la captura vía computadora es por medio de una aplicación que contenga un pantalla de captura que soporte las reglas del formulario. Este enfoque no es correcto para lograr una aplicación con las características antes descritas, las dos razones principales son:

- Relación uno a uno entre aplicación de captura y formulario
- El formulario cambia y la aplicación cambia, esto para internet es inviable por los problemas de distribución y compatibilidad de versiones que representa.

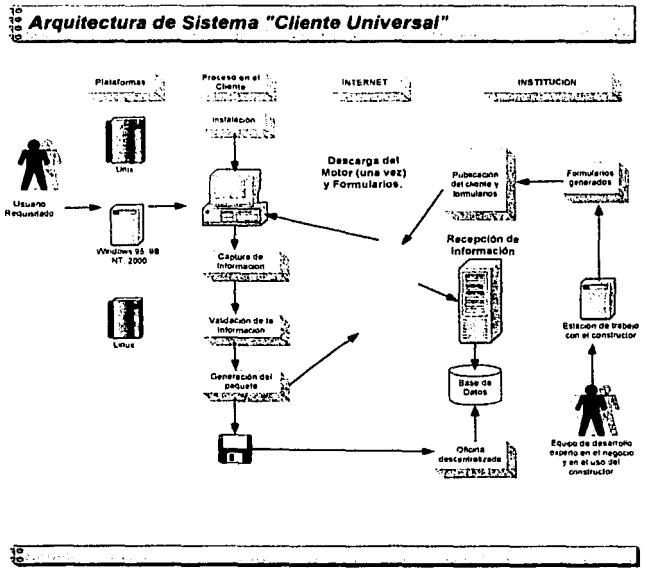
El enfoque de solución que soporta una aplicación como la que se busca es descrito en la sección siguiente.

Diseñador y motor de formularios.

La solución propuesta elimina la existencia de una aplicación o sistema de captura, a cambio de esto considera la existencia de un *Diseñador* de formularios en el sentido amplio de incluir todo su comportamiento. Entonces un *formulario* no es una pantalla de captura, es un *componente* que deberá ser interpretado por un *Motor* de formularios.

El motor de formularios debe generar entonces la pantalla de captura que soporte el comportamiento de cada formulario. El motor nunca cambia debido a cambios en los formularios, salvo que éstos exijan una funcionalidad no soportada por el mismo, pero esto estará en función de la habilidad del equipo de análisis y diseño a cargo.

Así entonces tenemos dos módulos que vinculan su funcionamiento por medio de los formularios. El *diseñador* trabajando centralmente, creando formularios y publicando éstos en algún sitio de internet y el *motor* trabajando como *Cliente Universal* en cualquier PC o equipo. Por conexión a internet se obtendrán los formularios que se desee. Un esquema de la arquitectura de solución se muestra en la siguiente figura.



Especificaciones del diseñador y del motor

El diseñador de formularios trabaja en forma centralizada, comúnmente deberá ser manejado por el usuario de negocio, no por las áreas de sistemas, esto de por sí trae la gran ventaja de descargar a las áreas de desarrollo de trabajo repetitivo y pesado, liberándolos así para la búsqueda de mejoras en otros procesos que los demanden. El motor es una aplicación que se encuentra disponible en internet y que primero obtiene los formularios de la red para después interpretarlos y así generar la captura de los datos. El motor debe preparar al final los datos para su envío, así como garantizar su autenticidad y su confidencialidad.

Diseñador

Es una aplicación de enfoque administrativo y centralizada, que es responsable de crear y modificar formularios, almacenarlos en un formato que llamaremos **componente** y que posteriormente el motor maneja para generar entonces la captura de la información. Los formularios serán los componentes independientes. Aquí el uso de XML es fundamental.

Un **formulario** está constituido por:

Secciones
Páginas
Componentes de Detalle

La creación de un formulario debe permitir:

- Establecer un fondo del formulario, comúnmente logotipos y diseño, esto se logra con la incorporación de imágenes asociadas, las cuales eliminan el uso específico de coordenadas para posicionar los objetos.
- Manejar una o varias páginas por formulario.
- Manejo de catálogos para información que lo demande como son marcas de vehículos, entidades federativas, etc.
- Reglas de negocio por dato o componente de detalle.
- Tipo de dato para las entradas del formulario, más aún información específica como alineación, fuente, rango de datos, tipografía, etc.
- Manejo de páginas con repetición en número indeterminado.
- Posibilidad de cálculos sobre datos en páginas con repetición.

Asimismo un **formulario** debe poseer en sus propiedades la siguiente información:

- Nombre de Identificación
- Vigencia
- Versión
- Fecha de llenado, el mismo formulario puede servir para ocasiones diferentes.

- Propietario
- Situación o estado (en proceso, ya validada la captura, etc.)

Los **componentes de detalle** del último nivel del formulario son en sí los datos y tienen asociadas una serie de propiedades que se agrupan en tres rubros, las propiedades comunes, las avanzadas y las de edición. El componente de detalle en forma gráfica se presenta en el diagrama siguiente.

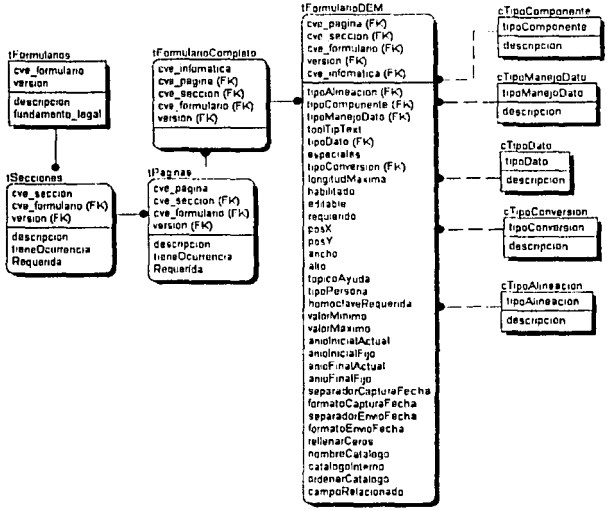
COMPONENTE DE DETALLE
PROPIEDADES BASICAS Posición Ancho Alto Alineación Tipo Longitud máxima Orden de navegación Visibilidad Requerido Habilitado Editable Ayuda en línea Valor mínimo (si el tipo es numérico) Valor máximo (si el tipo es numérico) Ceros a la izquierda (si el tipo es numérico) Ceros a la derecha (si el tipo es numérico) Conversión a mayúsculas o minúsculas (si el tipo es alfabético)
PROPIEDADES AVANZADAS Reglas de Negocio
PROPIEDADES DE EDICIÓN Cortar Copiar Pegar Eliminar

Las propiedades avanzadas permiten manejar la regla de negocio del componente, por ejemplo si su valor se suma a otro componente existente o si se obtiene de operaciones entre otros componentes de detalle.

Existen también operaciones de edición sobre los componentes, el comportamiento de estas operaciones es el conocido de edición, por ejemplo al hacer un "Pegar" ya sea que proviene de un "Cortar" o "Copiar" se heredan las reglas del componente origen.

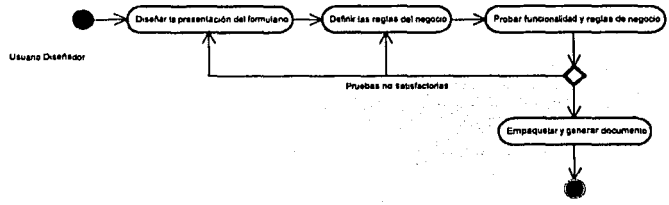
Finalmente, deberá contarse con construcción de expresiones lógicas y aritméticas para otorgar comportamiento a los componentes de detalle. Esto con auxilio de un analizador sintáctico y su contraparte evaluadora en el motor.

La siguiente vista del modelo de datos de la aplicación muestra el diseño básico de un documento del cliente universal. Como puede notarse todas las propiedades de un dato a capturar se encuentran aquí y a partir de este esquema puede comenzar a generarse un modelo más complejo que permita centralizar las propiedades de todos los datos que conformen el negocio de una empresa o institución, aunque son necesarias otras premisas como que los datos sean definidos de manera única, es a partir de esta premisa que puede comenzar a plantearse una evolución de este concepto hacia la especificación básica de un *repositorio central de reglas de negocio*, a partir del cual se alimenten todas las aplicaciones que conformen la vista del usuario final de una empresa o institución de manera dinámica.



Un diagrama de flujo que ilustra la secuencia para la creación o diseño de un formulario se encuentra en la gráfica siguiente.

Flujo de Trabajo en el Diseñador de Formularios



Motor

Es una aplicación o sistema que deberá ejecutarse fuera de línea pero disponible en internet para ser descargado desde algún sitio en la red. Se encarga básicamente de:

- Obtener los formularios desde internet (conexión en línea momentánea)
- Interpretar el formulario generando la interfaz de captura.
- Validar los datos capturados por el usuario
- Garantizar la confidencialidad y autenticidad de la información
- Preparar el paquete de envío por internet.
- Administrar los formularios por propietario y administración de propietarios

Este sistema se ejecuta en cualquier PC en forma de Cliente.

El motor maneja entonces formularios y propietarios de los mismos, es decir puede ser usado en forma personal o colectiva. Permite altas y administración de propietarios y formularios. Una alta de un Propietario es realmente el alta de una persona, física o quizá moral en el ámbito fiscal.

El alta de formularios no es mas que acceder a ellos en un catálogo de los mismos disponible en Internet, descargarlo y registrarlo para su uso en el motor, naturalmente el motor debe contar con las validaciones necesarias para determinar si el formulario es válido o no.

Los datos que identifican a un propietario son los generales de una persona. Estos datos reciben trato especial debido a que deberán poder heredarse como datos constantes dentro de los formularios. En otras palabras, los datos de un propietario pueden ser cargados automáticamente en algunas entradas del formulario, sin capturar nuevamente. Por ejemplo el nombre o el RFC de la persona. Lo anterior trae como beneficio la disminución del riesgo de errores en la captura de datos de identificación.

Un formulario puede ser instanciado para tiempos diferentes, por ejemplo un formulario para manejo de declaración patrimonial puede ser instanciado anualmente, incluso si hay cambios en el formulario, en este caso debe actualizarse el formulario con el mismo motor.

Captura de información con el motor

Un formulario tiene un fondo que generalmente sirve como identificación de la institución que requiere la Información, contiene diseño y logotipos de la institución. Este fondo puede ser una imagen que hará las veces de plantilla de captura.

Contiene campos con alguno de los siguientes comportamientos.

- Capturas directas, con o sin validación
- Campos calculados
- Campos gobernados con catálogo.
- Campos heredados del propietario (constantes)
- Campos auxiliares, los cuales son útiles para cálculos adicionales que no necesariamente son parte de la información requerida.

El motor debe ser capaz de manipular las múltiples páginas de un formulario y permitir la generación ilimitada de páginas de tipo *anexo* o página con repetición, las cuales son utilizadas tantas como el propietario en turno lo requiera.

El motor presenta una página a la vez, si existen referencias entre campos a través de las páginas el motor deberá resolverlo aun sin tener presente las páginas referenciadas. Se recomienda la evaluación de estas referencias al momento de cambiar de página.

Una evaluación o chequeo final es incorporado en el motor, con esto se garantiza que toda la Información es válida y respeta todas las reglas del formulario. Si esta validación devuelve errores entonces el motor está impedido a preparar la Información para envío.

Es muy importante citar que así como el diseñador contiene un constructor de expresiones, el motor cuenta con la contraparte evaluadora de expresiones. Técnicamente es oportuno precisar que no podrá ser utilizado ningún evaluador de terceros (como el de MS-Excel) debido a que dañaría la portabilidad. Es un evaluador propio del motor, que puede construirse con un árbol de decisión o simplemente tokenizando expresiones.

También deberá contarse con herramientas de búsqueda, por página o por datos de propietario.

Toda la información que genere el motor debe ser independiente de una base de datos comercial, por lo que se recomienda el uso de archivos planos para grabar información, aunque Java cuenta hoy día con bases de datos relacionales embebidas. Si se manejan archivos planos es necesario manejar longitudes fijas de registro, por ejemplo un registro por página de formulario con

posiciones fijas por campo. Así se optimiza tiempo de acceso a la Información y se independiza por completo de una base de datos comercial.

El formulario

Cada formulario que es instanciado transita por una serie de estados a lo largo de su ciclo de existencia, estos estados se describen a continuación.

En proceso

Desde que se inicia la captura y hasta que se aplica la evaluación final. Durante este estado el usuario puede borrar, modificar y complementar.

Validado

Si el formulario es sometido a evaluación final y la bitácora de errores generada está vacía, entonces alcanza el status de *Validado*. Si en este estado se modifica algún dato el estado vuelve a *En Proceso*. La bitácora de errores debe mostrarse en un formato simple y conciso y debe permitir la navegación e incluso el posicionamiento preciso del error en página y componente.

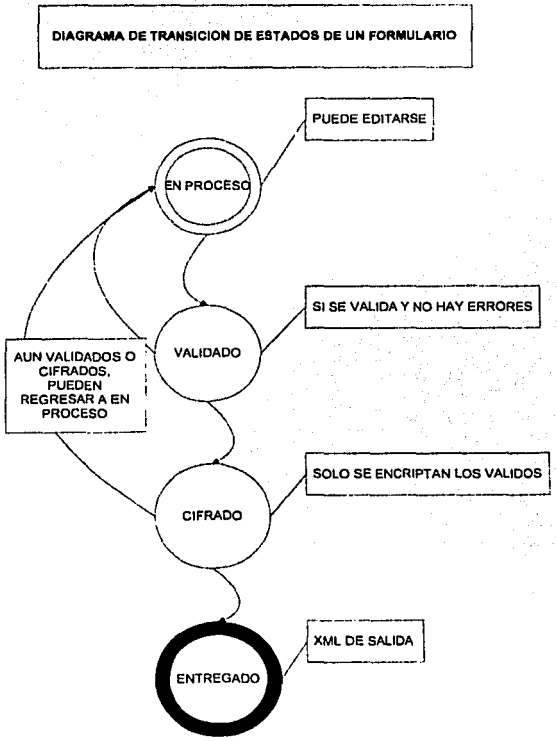
Cifrado

Antes de generar el archivo de salida, el formulario debe ser *Cifrado*, solo se pueden cifrar formularios en estado *Validado*. También puede editarse un formulario Cifrado, regresándolo a estado de *En Proceso*.

Entregado

Una vez que el formulario ha sido cifrado es factible de ser transmitido a la institución requirente, para lo cual debe ser depositado en medio magnético o enviado vía Internet por el motor mismo. El archivo de salida es un XML firmado con encabezado y una serie de detalle. Una vez entregado el XML éste será procesado según las necesidades de la institución requirente. Un formulario con estado de *Entregado* no puede ser modificado, ni eliminado.

La gráfica siguiente ilustra el diagrama de transición de estados del formulario



Finalmente, en la posterior y final sección de este trabajo, se encuentra la plataforma de herramientas utilizada para la construcción completa y una breve reseña de su uso durante la construcción.

Mecanismo de recepción

Para cerrar el ciclo de la operación del Cliente Universal es necesario que la institución requirente cuente con un esquema de recepción de documentos electrónicos, este esquema no debe ser más que una aplicación central con una interfaz a internet y que valide lo que recibe, cabe aclarar que un esquema de recepción escapa a los alcances de este trabajo por lo que sólo se describen sus características básicas. Las validaciones pueden ser del siguiente tipo:

- Validación de versión del cliente que emite el documento.
- Validación de autenticidad del documento.
- Validación de confidencialidad del documento.
- Validación de vigencia del documento recibido.
- Validación de integridad de la información, es decir que lleguen todos los datos que deben llegar.

Aún cuando se puede integrar una validación aritmética para garantizar al máximo la integridad de la información, se recomienda que ésta se deje al cliente y se confíe en él, pues de otra manera el proceso se puede cargar innecesariamente y en períodos pico se puede llegar a tener denegación de servicio.

El sistema receptor debe ser capaz de entregar un acuse de recibo electrónico, el cual a su vez debe ser infalsificable, al usuario que envió el documento, este acuse se puede acompañar o no de un sello digital.

Finalmente debe considerarse una infraestructura de hardware y de comunicaciones lo suficientemente robusta para soportar los volúmenes de información que se espere según la institución donde se busque implementar una solución como ésta.

Software utilizado en la construcción del cliente universal

El cuadro siguiente resume las herramientas utilizadas durante la construcción del Sistema

<table>
<tr><td colspan="10">Software de Construcción</td></tr>
<tr><td colspan="2">PLANEACION</td><td>ANALISIS</td><td>DISEÑO</td><td colspan="2">CONSTRUCCION</td><td>PRUEBAS</td><td>IMPLANTACION</td><td>MANTENIMIENTO</td><td>MANEJO DE ARCHIVOS</td><td>INTERCAMBIO DE INFORMACIÓN</td></tr>
<tr><td colspan="3">HERRAMIENTAS UPPER CASE</td><td colspan="4">HERRAMIENTAS LOWER CASE</td><td rowspan="2"></td><td></td><td rowspan="2"></td></tr>
<tr><td colspan="3" rowspan="2"></td><td colspan="2">APLICACIÓN</td><td>MIDDLEWARE</td><td>BASE DE DATOS</td><td>JDK 2.11</td></tr>
<tr><td colspan="3" rowspan="3">Rational Rose 7.7
Platinum Ewin3 5.2</td><td rowspan="4">JAVA</td><td rowspan="2">Borland JBUILDER PROFESSIONAL 5
JDK 2.11</td><td rowspan="4">N/A</td><td rowspan="4">ARCHIVOS PLANOS</td><td rowspan="4"></td><td rowspan="4"></td><td rowspan="4">• IBM- XML VIEWER 2.01</td></tr>
<tr><td colspan="3"></td></tr>
<tr><td colspan="3"></td><td colspan="2">CONTROL DE VERSIONES
Platinum CCC HARVEST 4.0.0</td><td>• Borland JBUILDER</td></tr>
<tr><td colspan="10">H - Herramientas para documentación y manejo de imágenes</td></tr>
<tr><td colspan="10">JASC Software PAINT SHOP PRO 6.0</td></tr>
<tr><td colspan="10">MS - VISIO 2001</td></tr>
</table>

MS – OFFICE
II - Herr. para Integración de Ambientes
N/A
II - Herr. de Diagnóstico y Reparación
Symantec NORTON UTILITIES de Norton System Works 2001
II - Herr. para Seguridad Informática
ALGORIMOS DE LLAVE PUBLICA (PK) Segurilib
II - Sistemas de Información Administrativos
N/A
Distribución de Software
MS – IIS 4.0
MS – EXCHANGE SERVER 5.0

Rational Rose

Durante la etapa de análisis y diseño se utilizó, como ya se mencionó en el capítulo I, una herramienta CASE enfocada a UML. Esta herramienta es Rational Rose 7.7 de Racional Rose Corporation y brindó la oportunidad de experimentar el estado real de aportación que hacen estas herramientas durante la construcción de una aplicación con un equipo de desarrollo de mediano alcance.

Si bien Rose permitió el primer y gran avance de código en Java, no hizo nada más. Posteriormente fue imposible lograr sincronización entre modelado y código. En otras palabras, el modelado tomo su camino y el código el suyo y fue necesario un gran esfuerzo final para que el modelado representara de beuna manera lo plasmado en el código.

Java

Como se mencionó en capítulos anteriores se seleccionó Java como lenguaje de desarrollo, el ambiente de desarrollo que lo implementa fue Borland Jbuilder Professional 5, su aplicación permitió lograr lo fundamental, una aplicación cien por ciento portable, al menos en las configuraciones que saturan el mercado, MS-Windows por ejemplo.

La aplicación de este lenguaje tiene sus cuidados, no por aplicarlo se tiene en automático la portabilidad. En el capítulo II se presentó un inventario de situaciones que generan problemas de portabilidad bajo Java.

Para el ámbito internet la seguridad es un aspecto de suma importancia, las bondades de Java a este respecto también son parte fundamental de la aplicación.

XML Viewer

Toda la información de la aplicación que viaja por internet, formularios y su contenido, son manejados con XML. Existe un XML por cada formulario.

La aplicación, gracias a Java, es capaz de leer e interpretar estos archivos XML, a su llegada a un nivel central se puede disponer de los mismos desde otro lenguaje o herramienta, esto gracias a que están bajo el estándar XML.

XML Viewer de IBM ofreció las capacidades suficientes para el diseño y construcción de los archivos XML.

Conclusiones.

Se concluye que la moda creciente de internet también tiene sus desencantos, las frases como "hágalo desde la comodidad de su oficina" en ocasiones no tienen el soporte de sistemas que les otorguen validez.

En el caso especifico de requerir información desde internet se confirma la existencia de un problema serio, siendo comunes situaciones como:

- Ofrecimientos, típicamente de instituciones de gobierno, para llevar a cabo algún trámite vía internet. La desagradable sorpresa viene cuando resulta necesario mantener en línea la captura de formularios complejos o extensos. Una desconexión resulta fatal y desalentadora.

- Uso de internet como medio de distribución de aplicaciones. Aplicaciones cuya ejecución generalmente permite preparar información que posteriormente se retroalimentará también vía internet. Existen aplicaciones que miden al menos diez megabytes y que se ejecutan únicamente en cierta configuración de equipo. Es un proceso de desgaste buscar la versión adecuada al equipo que se usará y después descargar diez megabytes de la red.

Al ejecutar las aplicaciones antes descritas existen serias limitaciones de validación de la información, invalidez de rangos de fechas y de valores, valores derivados redundantes, ausencia de máscaras de captura, ausencia de catálogos. Y en general poca solidez para la validación de información, que al ser enviada con deficiencias no sirve para nada

También se concluye que se está corrigiendo el rumbo, la aparición de lenguajes y herramientas orientadas a la producción de aplicaciones portables da la posibilidad de contrarrestar el problema.

Durante la construcción del *Cliente Universal* se tuvo la oportunidad de constatar puntualmente lo siguiente:

- Si bien el interés de los proyectos de construcción de Software se centra en el código y no en la documentación, de no dar la importancia necesaria a los planos o diagramas de los sistemas, la situación en este campo seguirá siendo caótica y las pérdidas por mantenimiento se mantendrán. El apoyo que ofrecen las herramientas CASE para este fin siguen teniendo deficiencias, la generación de código sigue sin aparecer en la realidad. El uso de un lenguaje de modelado como UML que constituya un estándar en como representar planos de sistemas también sigue siendo muy limitado debido a la poca importancia que se le da a documentar los sistemas.

- Buscando portabilidad aparece el lenguaje Java, que ofrece las facilidades para lograrlo, aunque queda claro que por su sola aplicación no se obtiene la misma. De hecho fue posible presentar un inventario de los errores más comunes al programar en Java y que afectan la portabilidad.

- Para el intercambio de información también es necesario ponerse de acuerdo, XML es una opción bastante sólida para ser considerada en este sentido.

Otra conclusión es que un desarrollo de esta índole debe ser acompañado de un esquema de recepción, que puede impactar en la infraestructura de hardware y comunicaciones de una empresa, aunque si es bien planteado redundará en beneficios al corto plazo.

Finalmente es importante mencionar que el presente trabajo constituye una importante aportación a la industria para la recopilación de información en gran escala a través de formularios, ninguna institución o empresa en México salvo el Servicio de Administración Tributaria cuenta con un esquema como el aquí descrito, lo cual es entendible pues el comprometerse a una empresa como ésta es un paso que representa un gran riesgo y la decisión, de por sí de alto nivel, tiene que contar con el consenso de mucha gente dada la inversión en dinero y tiempo que se ha de hacer.

Sin embargo con este trabajo queda plenamente demostrado que un esquema de esta envergadura y calidad puede desarrollarse en nuestro país y por mexicanos.

Bibliografía

The Java Programming Language (Third Edition)
Ken Arnold, James Gosling, David Holmes
Sun Microsystems

Enterprise Architect for J2EE Technology
Mark Cade, Simon Roberts
Sun Microsystems

UML Distilled: A brief guide to the standard object modeling language (2nd Edition)
Martin Fowler; Kendall Scott
Addison Wesley

UML y Patrones: Introducción al análisis y diseño orientado a objetos.
Craig Larman.
Prentice Hall, 1999.

Java2: The Complete Reference (Third Edition)
Patrick Naughton, Herbert Schildt
Osborne/McGraw-Hill, 2000

1001 tips para programar con Java
Steven W. Griffith
McGraw-Hill. 1997

The Java Tutorial, A short course on the basics
Mary Campione, et al
Addison-Wesley. 2000

Thinking in Java
Bruce Eckel
Prentice Hall

Java I/O
Elliote Rusty Harold
O'Reilly

Database Programming with JDBC and Java
George Reese
O'Reilly

Professional Java Programming
Brett Spell
Wrox

Professional XML
Didier Martin, Mark Birbeck, Michael Kay, Brian Loesgen
Wrox