PROYECTO DE GRADO

PROYECTO GML

Un lenguaje de marcas para generar interfaces gráficas

Informe del proyecto

Marzo 2004

Estudiantes: Lucía Bonifacino, Marcelo Saccone

Tutores: Juan José Prada (INCO), Ricardo Zuasti (Koala Developers)

Instituto de Computación
Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay.

NOTA DE ENTREGA

A continuación se listan los ítems que forman parte de la entrega de este proyecto.

Nº. Item	Nombre	Descripción	Medio
1.	Informe del proyecto	Este documento	Copia impresa y CD
2.	Anexo A – Requerimientos	Documento de requerimientos	CD
3.	Anexo B – Relevamiento del estado del arte	Relevamiento del estado del arte	CD
4.	Anexo C – Decisiones tecnologicas	Decisiones tecnológicas	CD
5.	Anexo D – Alcance	Alcance del sistema	CD
6.	Anexo E – Arquitectura	Arquitectura del sistema	CD
7.	Anexo F – Testing del sistema	Testing del sistema	CD
8.	Anexo G – Manual de usuario	Manual de usuario	Copia impresa y CD
9.	Anexo H – Manual de desarrollo	Manual de desarrollo	Copia impresa y CD
10.	Anexo I – Glosario	Glosario	CD
11.	Anexo J – GMLSchema.xsd	XML Schema con la definición del lenguaje GML	CD
12.	Anexo K – Codigo fuente	Código fuente	CD

RESUMEN DEL PROYECTO

El proyecto consiste en la construcción de una herramienta de desarrollo que permite generar interfaces gráficas de forma sencilla y rápida mediante la escritura de documentos XML que definen el contenido, la estructura y el comportamiento de dichas interfaces.

La motivación de este proyecto tiene su origen en diversos aspectos de la realidad actual. Uno de ellos radica en el hecho de que debido al amplio desarrollo de Internet y de las aplicaciones Web, la mayoría de los programadores tienen conocimientos en programación Web. El objetivo es que GML facilite la creación de aplicaciones de escritorio ya que es un lenguaje de marcas al igual que HTML. Otro de los aspectos que motivó este proyecto, es que hasta el momento la programación utilizando las librerías gráficas actuales no es sencilla ni intuitiva. GML fue diseñado de forma tal de simplificar las complejidades que deben ser enfrentadas al programar una interfaz de aplicación de escritorio.

El lenguaje GML (GUI Markup Language) en el cual se especifican las interfaces gráficas, está definido formalmente mediante XML Schema. Esto permite realizar rigurosas validaciones previo al procesamiento de los documentos.

La herramienta permite definir interfaces gráficas que contengan los componentes más comúnmente utilizados, entre los que están etiquetas, botones, campos de texto, menús, listas y árboles entre otros; manejar los eventos generados por éstos escribiendo código embebido en sus documentos XML (code-inline), o en documentos independientes (code-behind). Este código embebido puede hacer referencia a los componentes existentes en la interfaz logrando que esta sea completamente dinámica.

La herramienta procesa los documentos XML utilizando la API SAX y genera código en lenguaje Java que es independiente de la librería gráfica con la cual se desee ejecutar. Actualmente, se provee soporte para la ejecución utilizando las librerías Swing y SWT de Java. Adicionalmente, la arquitectura de GML fue diseñada de modo tal que posibilita la integración de otras librerías gráficas y de futuros generadores para otros lenguajes.

Como característica no funcional de la herramienta se destaca la posibilidad de ejecución en cualquier plataforma para la cual exista una implementación de la JVM (Java Virtual Machine).

INFORME DEL PROYECTO

Fecha: 01/05/2004

Versión: 0.0.2

Responsables: Lucía Bonifacino, Marcelo Saccone

ÍNDICE

1 INTRODUCCIÓN

1.1 Objetivo

GML es un lenguaje de marcas basado en XML para la especificación de interfaces gráficas (de aquí proviene el nombre GUI Markup Language). GML también es una herramienta de generación de interfaces gráficas Java a partir de documentos escritos en este lenguaje.

Por tanto, el objetivo del proyecto abarca la especificación del lenguaje y la implementación de la herramienta correspondiente.

1.2 Motivación

Inspirado por las metodologías de desarrollo de páginas Web dinámicas, la herramienta permite separar la presentación de la lógica del sistema, y la diagramación de interfaces con *layouts* dinámicos y flexibles.

Otros motivos que impulsaron este proyecto tienen su origen en dos aspectos de la realidad actual. El primero de ellos radica en el hecho de que debido a la amplia difusión de Internet y de las aplicaciones Web, la mayoría de los programadores tienen mas experiencia en programación de interfaces Web, que en interfaces de aplicaciones de escritorio. GML busca facilitar la creación de aplicaciones de escritorio ya que es un lenguaje de marcas al igual que HTML y debiera resultar familiar para los programadores Web.

El segundo aspecto de la realidad que motivó este proyecto, es que hasta el momento, la programación utilizando las librerías gráficas Java de la actualidad tiene aspectos complejos y poco intuitivos. Entre las características mas destacables se encuentran el manejo del layout (para el cual existen diversos modelos y en ninguno se logra satisfacer todas las necesidades), y la dificultad para hacer referencias entre los distintos componentes gráficos que pertenecen a una misma interfaz. Además, el manejo de eventos (basado en el modelo de suscripción) resulta poco intuitivo para comprender en un principio, y en general la asignación de propiedades a componentes, como puede ser la fuente, el color, etc. resulta un poco tediosa ya que implica la creación de objetos que representen estas propiedades y luego la asignación de estos objetos al componente.

Estas propiedades motivaron para que GML ofrezca una forma de crear interfaces gráficas con la misma potencialidad que las librerías convencionales atacando las deficiencias mencionadas.

1.3 Características

El lenguaje GML permite especificar los eventos que se desean manejar para cada componente y escribir el código que se desea ejecutar cuando este es disparado. Este código puede alterar el estado del componente que ocasionó el evento o de cualquier otro que pertenezca a la misma interfaz. La asociación de código puede realizarse en dos modalidades diferentes: escribiendo el código directamente en el documento (*code inline*) o escribiendo el código en otro archivo y luego haciendo referencia a éste desde el documento principal (*code behind*).

Se realizó una definición formal del lenguaje, lo que permite validar la sintaxis y la gramática de los documentos XML previo a la generación.

La herramienta GML está implementada en el lenguaje Java, lo que asegura que funcione en cualquier plataforma que soporte Java. Adicionalmente, para la generación de archivos se tuvieron en cuenta algunas consideraciones con el objetivo de lograr que los archivos sean generados de acuerdo a la plataforma utilizada.

La herramienta fue diseñada con una arquitectura abierta que permite la abstracción de la librería gráfica utilizada, generando código que puede ser ejecutado utilizando diferentes librerías. Asimismo, permite la futura integración de otras librerías gráficas y de generadores para otros lenguajes.

Como características del producto final, se destacan el manejo del layout propuesto, en donde se incursionó en conceptos innovadores para la plataforma Java, y el hecho de que el código generado es independiente de la librería que se utilice en tiempo de ejecución. Actualmente, GML provee soporte para que este código sea ejecutado utilizando las librerías Swing o SWT de Java.

1.4 Estructura del documento

El **capítulo 2 — Conceptos utilizados —** estudia los conceptos cuyo entendimiento es necesario para la comprensión de algunos capítulos presentados en este informe. Lectores con conocimientos en orientación a objetos, modelo de delegación de eventos, patrón de diseño *Factory Method* y *Reflection* pueden omitir la lectura de este capítulo.

El **capítulo 3 — Estado del arte —** repasa brevemente la investigación realizada sobre las tecnologías XML, y temas concernientes a interfaces gráficas.

El **capítulo 4 — Alcance del proyecto —** se concentra en la definición del alcance del proyecto. Se narra la propuesta de layout introducida. Luego, se presenta el alcance del proyecto en términos de componentes y eventos soportados, discriminando que grupo sería implementado en una primera etapa, y cual sería dejado para una etapa posterior. Más adelante se describen las características de la generación y la ejecución.

El **capítulo 5 — Decisiones tecnológicas —** analiza las decisiones tecnológicas de mayor relevancia que se tomaron en el marco del proyecto.

El **capítulo 6 — Definición del lenguaje GML —** describe en forma breve los elementos del lenguaje GML. Lectores con conocimientos avanzados de XML Schema, podrán omitir la lectura de este capítulo, y consultar directamente el archivo GMLSchema.xsd (**[Ref: Anexo J – GMLSchema.xsd]**), que contiene la definición formal del lenguaje.

El **capítulo 7 — Arquitectura —** trata sobre el diseño de la arquitectura del sistema, detallando su estructura y los principales desafíos que se debieron resolver.

El **capítulo 8 — Plan de pruebas del sistema —** presenta la metodología utilizada para realizar el plan de pruebas del sistema, y puntualiza todos los casos de prueba realizados, culminando con un análisis general de los resultados obtenidos.

El **capítulo 9 — Conclusiones y trabajos futuros —** presenta una conclusión general del proyecto, e introduce algunas ideas interesantes para continuar este trabajo.

2 CONCEPTOS UTILIZADOS

En esta sección se realiza una aproximación de alto nivel hacia algunos de los conceptos utilizados en el proyecto, de los cuales es requisito su familiaridad por parte del lector para el entendimiento de secciones posteriores.

2.1 Orientación a objetos

La orientación a objetos es una metodología de programación, basada en una jerarquía de clases y objetos claramente definidos que cooperan entre sí.

Los objetos a los cuales nos referimos, son piezas de software que sirven tanto para modelar objetos de la vida real, o conceptos abstractos como puede serlo una operación matemática. Un objeto contiene un conjunto de variables a través de las cuales mantiene un estado, y un conjunto de métodos que le permiten definir su comportamiento. Un objeto encapsula la información sobre su estado, y ofrece una interfaz para que otros objetos se comuniquen con el. Esto conlleva un alto grado de modularidad, lo que permite que el mantenimiento sobre el código fuente asociado, se realice de manera independiente del resto del sistema.

Como es de suponer, es normal que existan varios objetos del mismo tipo. Una clase es una especie de prototipo de objeto, en donde se definen variables y métodos comunes a todos los objetos de un mismo tipo. Como alternativa de optimización, se pueden definir variables y métodos de clase, utilizados para manejar información concordante a todos los objetos de un mismo tipo.

Al nombrar la jerarquía de clases, es donde se introduce el concepto de herencia. La herencia es una forma de reutilización de software, en la cual se crean nuevas clases a partir de otras ya existentes, mediante la incorporación implícita de sus variables y métodos, y extendiendo éstos con las capacidades que las nuevas clases requieren. Al crear una clase, se puede especificar que debe heredar de otra clase ya definida, denominada clase base. La clase recién creada, se denomina clase derivada, que a su vez, se convierte en candidato potencial a clase base para alguna clase derivada futura. Es así que la herencia forma estructuras jerárquicas de apariencia arborescente.

A su vez, la utilización de herencia entre clases, nos introduce en el concepto de clases abstractas. Como lo indica su nombre, estas clases no pueden ser instanciadas. Las clases abstractas son aquellas cuyo único fin es proporcionar una clase base apropiada, desde la cual puedan heredar otras clases y reutilizar su implementación. Una clase es abstracta cuando uno o mas de sus métodos se declara como abstracto, es decir, no se provee implementación para el método.

Otro de los beneficios de usar la herencia en el diseño, es la posibilidad de utilizar polimorfismo. El polimorfismo es la capacidad de objetos de clases diferentes (relacionados mediante la herencia), a responder de manera diferente ante una misma llamada de algún método de la clase base que los relaciona.

Como último gran recurso del diseño orientado a objetos, existe un tipo de clase que se denomina interfaz. Una interfaz define como un contrato, en el sentido de que contiene un conjunto de métodos, y toda clase que desee implementar esta interfaz, se compromete a brindar una implementación para estos.

2.2 Modelo de delegación de eventos

El nombre de este modelo deriva del hecho de que el manejo de eventos es delegado de la fuente del evento, a una o mas entidades (*listeners*) que esperan por su invocación.

La premisa detrás del modelo es simple: los componentes disparan eventos los cuales pueden ser escuchados y procesados por determinadas entidades. Estas entidades deberán registrarse previamente, manifestando así su interés por ser notificados al momento de generarse algún evento en particular.

Asociado al evento en cuestión, existe un método en alguna interfaz dada, cuya implementación debe proveerse para el manejo del evento. La interfaz posiblemente contenga métodos asociados a otros eventos, que en caso de no existir interés en procesarlos, se debe proveer una implementación vacía para estos. Únicamente, se podrán registrar para recibir notificación sobre un evento, objetos de clases que implementen la interfaz que contiene el método asociado.

Para facilitar la implementación de las interfaces se pueden utilizar clases denominadas *Adapter*, que proveen implementaciones vacías para todos lo métodos de las interfaces, por lo que heredando de esta clase alcanzará con redefinir únicamente los métodos de nuestro interés.

2.3 Patrón de diseño *Factory Method*

El propósito de este patrón de diseño es definir una interfaz para crear un objeto, pero delegar a las subclases su instanciación. También es conocido como Constructor Virtual.

El patrón de diseño *Factory Method* puede ser usado bajo las siguientes circunstancias:

- Cuando una clase no puede anticipar la clase de los objetos que debe crear.
- Cuando una clase desea que sus subclases sean quienes especifiquen el objeto a crear.
- Cuando una clase delega la responsabilidad a una de varias clases auxiliares, y se desea centralizar el conocimiento de cual subclase es la delegada.

Existe una variación del patrón denominada *Parametrized Factory Method*, que permite que el método que crea los objetos, pueda crear múltiples tipos de objetos. En este caso el método toma como parámetro de entrada el identificador del tipo de objeto que debe ser creado, y según su valor instancia un objeto de la clase correspondiente.

2.4 *Reflection*

Es un concepto de la programación orientada a objetos que permite representar o reflejar clases, interfaces y objetos.

Utilizando *Reflection* se puede:

- Determinar la clase de un objeto.
- Obtener información sobre una clase: campos, métodos, constructores y superclases.
- Determinar las declaraciones de constantes y métodos que pertenecen a una interfaz.
- Crear una instancia de una clase cuyo nombre es determinado recién en tiempo de ejecución.
- Obtener y asignar el valor del campo de un objeto, aún cuando el nombre del campo es determinado recién en tiempo de ejecución.
- Invocar un método en un objeto, aún cuando el método es determinado recién en tiempo de ejecución.
- Crear un nuevo vector, cuyo tamaño y tipo de componentes es determinado recién en tiempo de ejecución, y luego modificar sus componentes.

3 ESTADO DEL ARTE

Esta sección contiene una breve reseña de la investigación realizada durante la primera etapa del proyecto, donde se cubrieron temas fuertemente relacionados con el proyecto, obteniendo como resultado de esta etapa un documento en el que se plasmó la información recavada (**[Ref. Anexo B - Relevamiento del estado del arte]**).

En primera instancia se estudiaron los conceptos básicos de XML, se analizaron algunos de los lenguajes existentes para la especificación y validación de documentos XML, como DTD y XML Schema, y se realizó una comparación entre éstos, analizando cuales características resultarían beneficiosas en el marco del proyecto.

Posteriormente, se analizaron dos de las APIs mas difundidas para el manejo de documentos XML: XML DOM y SAX. Del mismo modo, se compararon ventajas y desventajas que ofrecen ambas APIs.

Esta primera etapa tuvo como objetivo introducir y analizar las tecnologías que se sabía podrían ser de utilidad en el marco del proyecto.

Luego se investigaron herramientas generadoras de interfaces gráficas a partir de archivos XML, con el objetivo de conocer herramientas similares a la que se deseaba construir. La investigación se focalizó en analizar las facilidades que ofrecen, los componentes gráficos que soportan, las librerías gráficas que utilizan y también las deficiencias que presentan.

Por último, se estudiaron las librerías gráficas Java más conocidas y utilizadas en la actualidad: AWT, Swing y SWT, en donde se realizó nuevamente un análisis que abarcó componentes y eventos que ofrece cada librería, y los distintos modelos de *layout* que soportan.

En los siguientes párrafos se presenta un resumen de los principales conceptos estudiados, que son base sustancial de la teoría y la tecnología en la cual se basa este proyecto. Para profundizar esta información referirse al documento correspondiente (**[Ref. Anexo B - Relevamiento del estado del arte]**).

3.1 XML

XML (E**X**tensible **M**arkup **L**anguage) es un lenguaje de marcas diseñado para la descripción de datos. No existen *tags* predefinidos, sino que estos deben ser definidos por el desarrollador junto con la estructura del documento. XML utiliza un DTD (**[Ref: 3.2]**) (**D**ocument **T**ype **D**efinition) o un XML-Schema (**[Ref: 3.3]**) para la descripción de los datos, lo que permite su uso para la creación de nuevos lenguajes.

Básicamente, un documento XML se compone de elementos relacionados entre si. Dos elementos XML se relacionan como padre e hijo cuando uno contiene al otro, y como hermanos cuando ambos se encuentran dentro del mismo nivel. Además de contener a otros elementos, un elemento XML puede contener texto (contenido simple), texto y otros elementos (contenido mixto), o contenido vacío cuando no lleva ningún tipo de información adicional (no contiene texto ni otros elementos). Por último, un elemento XML puede tener atributos, los cuales en general se usan para proveer información que no es parte de los datos (metadata).

Para que un documento XML se considere válido, debe ser sintácticamente correcto (para lo cual debe cumplir con las reglas descriptas en el párrafo anterior), y además respetar un conjunto de reglas definido en un archivo con este fin (DTD o XML Schema).

Los documentos XML son extensibles, es decir, se pueden agregar elementos al documento XML y seguir procesándolo de la misma forma.

Actualmente existe una especificación (que se ha convertido en un estándar), de que es lo que debe realizar y como debe hacerlo, un programa que valide documentos XML. Dos de las interfaces más conocidas

que cumplen con esta especificación son XML DOM y SAX. Un programa que implemente una interfaz que cumpla la especificación mencionada se denomina *parser*.

Ejemplo de un archivo XML:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<nota fecha = ”10/07/1998”>
        <!-- esta es la nota que le deje a Maria -->
        <para>Maria</para>
        <de>Juan</de>
        <titulo>Recordatorio</titulo>
        <cuerpo>No te olvides de pasarme a buscar!</cuerpo>
</nota>
```

Para ilustrar el rol que XML esta tomando en la realidad actual, a continuación mencionamos algunas de las tecnologías que surgieron a partir de este.

- **XHTML** (e**X**tensible **HTML**): es la reformulación de HTML 4.01 en XML.
- **CSS** (**C**ascade **S**tyle **S**heets): proveedor de formato para documentos XML.
- **XSL** (e**X**tensible **S**tylesheet **L**anguage): compuesto por tres partes, transformación de documentos XML (XSLT), una sintaxis para la concordancia de patrones (*pattern matching*) (Xpath) y un intérprete para formatear objetos.
- **XSLT** (XML Transformation): más poderoso que CSS, puede ser usado para transformar documentos XML en distintos formatos de salida.
- **XPath** (XML Pattern Matching): es un lenguaje para el direccionamiento de partes de documentos XML.
- **XLink** (XML Linking Language): permite la inserción de elementos en documentos XML para crear asociaciones entre recursos XML.
- **XPointer** (XML Pointer Language): provee direccionamiento dentro de las estructuras internas de los documentos XML.
- **DTD** (**D**ocument **T**ype **D**efinition): (**[Ref: 3.2]**)
- **Namespaces**: especifican un método para la definición de elementos y nombres de atributos XML mediante la asociación de estos a referencias URI.
- **XSD** (XML Schema): alternativa más poderosa que los DTD. Son escritos en XML, y soportan *namespaces* y tipos de datos. (**[Ref: 3.3]**)
- **XDR** (**X**ML **D**ata **R**educed): versión reducida de XML Schema, introducida con IE 5.0 cuando XML Schema no estaba suficientemente maduro.
- **XQL** (**X**ML **Q**uery **L**anguage): provee funcionalidades de consulta para extraer datos de los documentos XML.
- **DOM** (**D**ocument **O**bject **M**odel): definición de interfaces, propiedades y métodos para manipular documentos XML. (**[Ref: 3.4]**)
- **SAX** (**S**imple **A**PI for **X**ML): interfaz para leer y manipular documentos XML. (**[Ref: 3.5]**)

3.2 DTD

El objetivo de un DTD (**D**ocument **T**ype **D**efinition) es definir la estructura de documentos XML partir de una lista de elementos permitidos. Un DTD puede ser declarado embebido en el documento XML, o como una referencia externa.

Desde el punto de vista del DTD, un documento XML esta compuesto por los siguientes bloques:

- Elementos: son los bloques principales, pueden tener texto, otros elementos o ser vacíos (Ejemplos: `body, table, hr, br`).
- Tags: utilizados para delimitar elementos.
- Atributos: proveen información extra sobre los elementos. Se ubican en el *tag* de inicio del elemento.
- Entidades: son variables usadas para definir texto (Ejemplos: ` , <, &`).
- PCDATA: **P**arsed **C**haracter **Data**, texto que será analizado por el *parser*.

- CDATA: **C**haracter **Data**, texto que no será analizado por el *parser*.

La sintaxis de DTD esta basada en SGML (Standard Generalized Markup Language), por lo cual es más difícil de *parsear* que si estuviera basada en XML. Al igual que XML, SGML es un lenguaje de marcas genérico para la representación de documentos.

DTD permite definir restricciones de cardinalidad (especificación de que un elemento se puede repetir cero o una vez, cero o mas veces, una o mas veces), soporta tipos de datos textuales y definición de enumeraciones (especificación de un valor a un conjunto restringido de valores).

Desde un punto de vista práctico, DTD es sencillo de utilizar, lo que lleva sumado a sus varios años de existencia, que tenga un uso y soporte muy extendido.

3.3 XML Schema

XML Schema es una alternativa al DTD basada en XML, y al igual que éste se utiliza para describir la estructura de un documento XML. La denominación formal es **X**ML **S**chema **D**efinition (**XSD**)

El propósito de un XML Schema es definir la construcción de bloques legales de un documento XML. Un XML Schema permite definir:

- Elementos que pueden estar presentes en un documento.
- Atributos que pueden estar presentes en un elemento.
- Relaciones entre elementos.
- Restricciones de cardinalidad (cero o una vez, cero o mas veces, una o mas veces, o un número determinado de veces).
- Tipo de contenido de un elemento (vacío o puede incluir texto).
- Tipos de datos para elementos y atributos.
- Valores por defecto y fijos para elementos y atributos.

XML Schema se presenta como el sucesor de DTD, y se estima que muy pronto será usado en la mayoría de las aplicaciones Web en su sustitución.

Las principales razones que dan lugar a esta afirmación son:

- Es más potente que DTD.
- La sintaxis está basada en XML.
- Soporta los tipos de datos predefinidos mas comunes (string, integer, float, boolean, etc.), y tipos de datos simples y complejos definidos por el usuario.
- Soporta validaciones basadas en el contenido mediante la definición de expresiones regulares.
- Soporta herencia. Permite extender otros documentos, posibilitando la reutilización y la refinación.
- Soporta espacios de nombres (*namespaces*), evitando conflictos de nombres.

Es importante recalcar que la potencialidad que XML Schema ofrece para definir documentos XML, trae aparejada una complejidad adicional que resulta en mayor dificultad para el usuario.

3.4 XML DOM

XML DOM (**D**ocument **O**bject **M**odel) es una interfaz estándar de programación de aplicaciones (API – Application Programming Interface) para manejar la estructura de documentos XML. El objetivo es facilitar la tarea de los programadores para acceder a los componentes de estos documentos, agregar y/o modificar su contenido. En conclusión, DOM ofrece a los programadores una interfaz de programación para el manejo de documentos XML, independiente de la plataforma y del lenguaje de utilizado.

XML DOM, permite crear un documento XML, recorrer su estructura, y agregar, modificar o eliminar elementos. Todo esto se provee a través de la representación del documento XML en su propio modelo de objetos, basado en árboles.

XML DOM necesita *parsear* el documento antes de comenzar el procesamiento. Luego de esto carga en su árbol de nodos el documento completo. Es tarea del programador escribir el código necesario para recorrer la estructura del árbol, pero como contraparte es posible acceder a cualquier sección del documento, en cualquier momento y cuantas veces sea necesario, e inclusive alterar su contenido, ya que XML DOM ofrece una API altamente funcional. Como desventaja tiene una mayor sobrecarga de memoria.

3.5 SAX

SAX (**S**imple **A**PI for **X**ML) define un mecanismo de acceso serial para la manipulación de documentos XML. Es el protocolo mas rápido, y también el mas eficiente, hablando en términos de memoria, que existe actualmente para el manejo de documentos XML. Esto en parte se debe a que el procesamiento se realiza en paralelo con el *parsing*.

SAX sigue un modelo basado en el manejo de eventos, en donde el usuario/programador implementa los métodos que los manejan (especificados en la interfaz de SAX), y el *parser* los invoca a medida que corresponda durante el procesamiento del documento. Por la forma de operar que presenta, no es posible volver a posiciones anteriores (ya procesadas) del documento, o avanzar posiciones, salteando partes del documento. Presenta como desventaja en contraposición a DOM, que requiere de un esfuerzo mayor en programación y es necesario definir un modelo de objetos propio para representar los datos de nuestros documentos XML.

El *parser* "dispara" (invoca) eventos, a medida que encuentra alguno de los siguientes bloques en el documento:

- *Tag* que abre un elemento.
- *Tag* que cierra un elemento.
- Secciones de texto tipo #PCDATA o CData.
- Instrucciones de procesamiento, comentarios o declaración de entidades.

En general, convendrá utilizar SAX cuando en nuestra aplicación exista una necesidad importante de procesamiento rápido y eficiente de los documentos XML, o nos encontremos ante una baja disponibilidad de memoria en el sistema.

Existen otras circunstancias donde el uso de SAX resulta más adecuado, cuando por ejemplo el acceso a los documentos XML es únicamente en modo lectura, o cuando la aplicación no requiere de un procesamiento complejo de los documentos XML, y este se realiza de manera secuencial y únicamente hacia adelante ("forward only").

3.6 Herramientas de generación de interfaces gráficas

En esta sección se realizó una investigación con su correspondiente análisis, sobre el estado del arte en herramientas de generación de interfaces gráficas a partir de archivos de especificación basados en XML.

Las herramientas que se analizaron relacionadas al tema, se pueden categorizar de acuerdo a algunas características básicas, y muy bien diferenciadas.

Existe un primer grupo de herramientas que se comportan como "browsers XML", es decir que dado un archivo XML, despliegan la interfaz especificada por éste en un *browser*. En este grupo se encuentran Mozilla XUL (y las diferentes herramientas XUL existentes: Luxor XUL, jXUL, etc.), XSmiles, y Enode.

Un segundo grupo de herramientas entre las que están Java Swing GUI, SwiXML y Glade, generan el código de la interfaz gráfica de aplicación correspondiente a partir de un archivo XML. Estas tres

herramientas se caracterizan por no proveer un mecanismo para la definición del comportamiento de la interfaz. En particular Java Swing GUI y SwiXML generan código Java únicamente para la librería gráfica Swing. A pesar de que no permiten la definición de comportamiento, ambas permiten la modificación de los archivos .java generados para agregarlo. Por su parte, Glade genera implementaciones vacías de los manejadores de eventos correspondientes a la interfaz especificada.

Por último, y como tercer grupo, destacamos la herramienta XWT que mediante una combinación de XML y ECMAScript permite desarrollar aplicaciones que ejecuten de forma remota, y "proyecten" la interfaz en la computadora del usuario, accesible desde Internet. Esto lo logra mediante la utilización de un control ActiveX, o un *applet* de Java, lo que le provee soporte para la mayoría de las plataformas.

Como conclusión de la investigación realizada sobre herramientas similares a la que se buscaba construir, podemos decir que ninguna reúne todas las características que se deseaba para GML. Estas son:

- especificar la interfaz gráfica en un archivo XML,
- permitir definir el comportamiento de la aplicación,
- permitir la elección de la librería gráfica con la que se quiere construir la interfaz, y
- que pueda ser ejecutado en diversas plataformas.

3.7 Librerías gráficas JAVA

Parte esencial de la investigación que era necesaria realizar, consistía en relevar las librerías gráficas Java mas difundidas. Conocer los componentes que soportan, los diferentes eventos para cada uno de ellos, y las posibilidades para el manejo del layout que proponen.

Se tuvieron en cuenta las librerías gráficas AWT, Swing y SWT, donde para cada una de éstas se realizó un detalle exhaustivo de las características mencionadas.

- **AWT** (**A**bstract **W**indows **T**oolkit) fue el primer paquete de Java para construir interfaces gráficas. Originalmente consistió en componentes gráficos desarrollados en Java, los cuales delegaban la funcionalidad a otros componentes desarrollados en lenguaje C. La política aplicada era del estilo "mínimo común denominador", lo que tiene por significado que únicamente se proveen los componentes soportados por todas las plataformas.

  AWT ha estado presente en Java desde la versión 1.0, y en la versión 1.1 se introdujeron cambios muy importantes, sobre todo en lo que respecta al modelo de eventos adoptado.

  Básicamente, para generar interfaces con AWT se requiere de un contenedor (ventana donde situar los componentes), un conjunto de componentes que constituyen la interfaz (menús, botones, etc.), manejar los eventos asociados para definir su comportamiento, y optar entre alguno de los modelos de layout que provee para estructurar y organizar la presentación.

- Swing surgió como el sucesor de AWT. Provee un conjunto de componentes "livianos" (todos en lenguaje Java) que al máximo grado posible, se comportan de igual forma sobre todas las plataformas. Swing adoptó una política diferente que AWT, brindando versiones emuladas para todos los componentes de alto nivel. Lo que utiliza son imágenes pre diseñadas y funciones de dibujo para crear sus propios componentes. Esto solucionó el problema de la funcionalidad, pero implicó una degradación en la performance, haciendo difícil desarrollar aplicaciones que compitan exitosamente con otras desarrolladas para una plataforma especifica.

  Swing provee todos los componentes de una interfaz gráfica típica, como botones, listas, menús y áreas de texto, que se pueden combinar para crear la interfaz del programa. También incluye contenedores como ventanas y barras de herramientas.

  Los componentes Swing son parte de Java Foundation Classes (JFC), y pueden ser usados con JDK 1.1 o la plataforma Java 2.

- SWT (**S**tandard **W**idget **T**oolkit) es una librería gráfica para Java, cuya principal virtud es la innovación en el diseño de su arquitectura, que permite utilizar componentes gráficos nativos del sistema operativo.

  Otro de los principales atractivos de esta librería es la diferencia de velocidad en comparación con Swing, debido a que para las construcción de las interfaces, aprovecha y reutiliza los componentes gráficos del sistema operativo.

  Por otro lado, este diseño generó cierto grado de controversia, debido a su relación de dependencia con los componentes nativos del sistema operativo utilizado como plataforma, lo que esta en contra de una de las principales características de Java, la portabilidad.

  La principal desventaja que presenta es que requiere instalar una biblioteca adicional, para cada plataforma en la que se desee ejecutar. Existe una versión específica del código binario de la librería SWT, y al menos una librería nativa para cada plataforma. En Windows esta librería será un .dll, en Linux un .so, pero si ésta no se encuentra disponible para determinado sistema operativo, entonces SWT no podrá ser utilizado en él.

  El hecho de que se utilicen los componentes nativos de la plataforma para la construcción de la interfaz, tiene como ventajas una presentación más familiar, y un uso de memoria significativamente menor.

  La política del "mínimo común denominador" se enfrentó tomando las siguientes decisiones:

  - SWT provee una imitación para aquellas características de uso muy extendido, pero que no están disponibles en todas las plataformas.
  - Pueden no ser provistas aquellas características que no están disponibles en todas las plataformas, y cuyo uso no es muy común.
  - Para las características que son específicas de una plataforma, se provee un paquete aparte únicamente para dicha plataforma.

4 ALCANCE DEL PROYECTO

En este capítulo se describe el alcance del proyecto. En primera instancia se explica la propuesta de *layout* introducida. Luego, se presenta el alcance del proyecto en términos de componentes y eventos soportados, discriminando que grupo sería implementado en una primera etapa, y cual sería dejado para una etapa posterior. Más adelante se describen las características de la generación y la ejecución.

4.1 Administración del layout

Como ya fue mencionado en la introducción, las librerías gráficas actuales disponen de varios modelos de *layout*, que a nuestro entender no son intuitivos y además no logran satisfacer las necesidades de los programadores al momento de crear una interfaz. Debido a esto, se resolvió no utilizar ninguno de los *layouts* provistos por las librerías gráficas, y darle la libertad al usuario de decidir la ubicación y el tamaño de los componentes y ofrecerle una forma sencilla de poder especificarlos.

Para cada componente que el usuario desee definir en la interfaz, deberá especificar el tamaño en términos de ancho y altura, y su ubicación horizontal y vertical respecto al componente que lo contiene, tomando como referencia el vértice superior izquierdo de dicho componente.

Adicionalmente, el usuario podrá indicar el comportamiento que desea que cada componente tenga al redimensionar el componente que lo contiene. Para cada borde del componente, el usuario podrá establecer mediante el atributo de nombre anchor una relación con el borde correspondiente del componente contenedor, que deberá ser alguno de los siguientes valores:

- fixed: la distancia entre los bordes correspondientes debe mantenerse constante.
- proportional: la distancia entre los bordes correspondientes se modifica proporcionalmente a la variación de tamaño del componente contenedor
- free: no se aplica ninguna restricción entre los bordes.

El usuario bien puede no especificar este atributo, en cuyo caso se utilizará el comportamiento por defecto, que mantiene el tamaño y la ubicación originales del componente.

Como consecuencia de las relaciones que se definan para cada componente, puede ocurrir que el tamaño de éste varíe, lo que en general es el comportamiento buscado.

A continuación presentamos tres ejemplos de comportamiento de los componentes al redimensionar el componente que los contiene. En los tres casos partimos de un *frame* con un área de texto y un botón, por lo que tendrán el mismo aspecto inicial, pero cada uno presentará comportamientos distintos al redimensionar el *frame*.

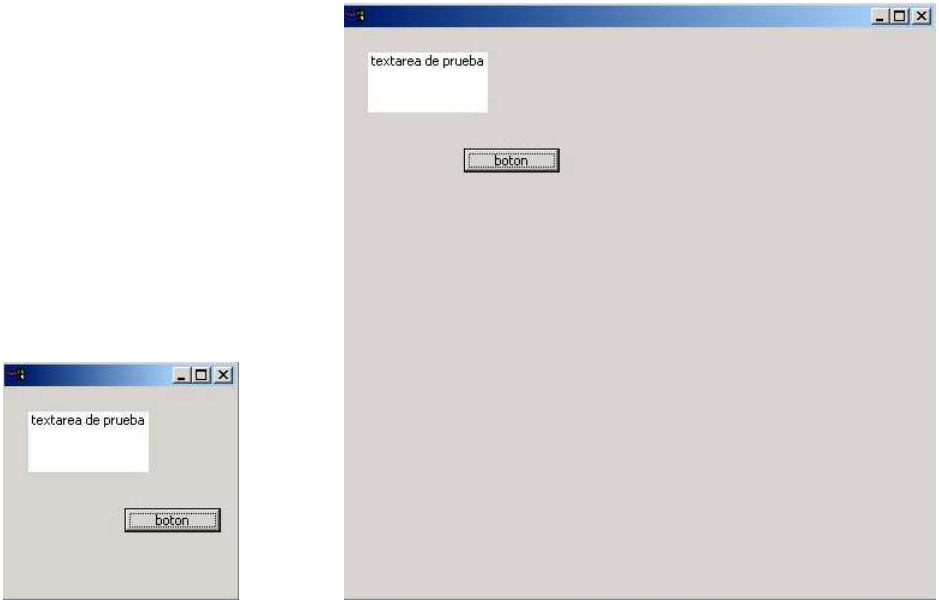
Inicialmente ubicamos al área de texto en la posición (20, 20), y con un tamaño asociado de (100,50). Al botón lo ubicamos en la posición (100,100), y le asignamos un tamaño de (80,20).

Ejemplo 1:

Deseamos que el área de texto y el botón mantengan su posición absoluta respecto a los bordes superior e izquierdo del *frame*, y no modifiquen su tamaño al redimensionar el *frame*.

Para esto especificamos los siguientes valores del atributo anchor para ambos componentes: top="fixed", left="fixed", bottom="free", right="free"

Al aumentar el tamaño del *frame*, se puede observar que se obtiene el comportamiento especificado.

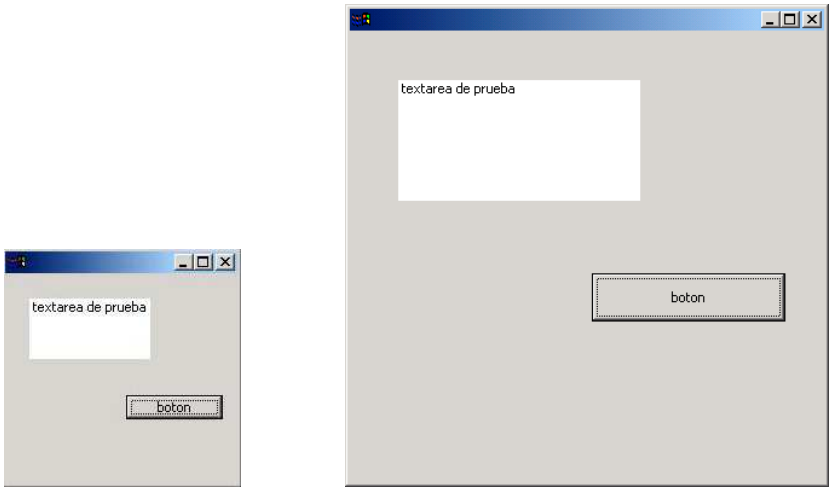


Ejemplo 2:

Deseamos que el área de texto y el botón modifiquen su distancia respecto a los bordes del *frame* y su tamaño, proporcionalmente a los cambios de tamaño del *frame*.

Para esto especificamos los siguientes valores del atributo anchor para ambos componentes: top="proportional", left=" proportional", bottom=" proportional", right=" proportional"

Al aumentar el tamaño del *frame*, se observa el siguiente cambio en la interfaz:

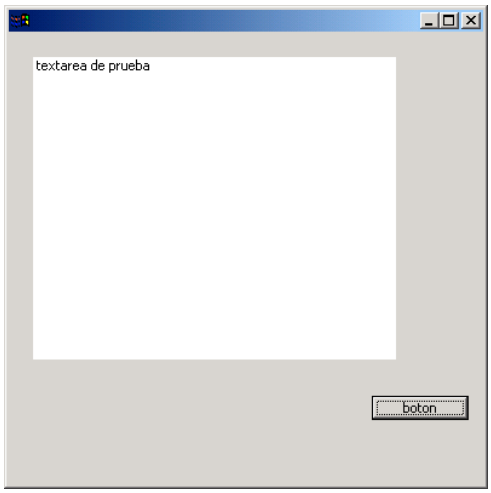


Ejemplo 3:

Deseamos que el área de texto mantenga su posición absoluta respecto a los cuatro bordes del *frame* y modifique su tamaño en caso que el del *frame* sea modificado, y que el botón mantenga su posición absoluta respecto a los bordes inferior y derecho del *frame*, modificando su ubicación respecto al *frame* y manteniendo su tamaño.

Para esto especificamos los siguientes valores del atributo anchor para el área de texto: top="fixed", left=" fixed", bottom=" fixed", right=" fixed", y los valores top="free", left=" free", bottom=" fixed", right="fixed" para el botón.

Al aumentar el tamaño del *frame*, se observa el siguiente cambio en la interfaz:



4.2 Atributos, eventos y componentes que puede contener cada componente

El criterio adoptado para la elección de los componentes a soportar, fue proveer a los usuarios de los componentes mas frecuentemente utilizados en las interfaces gráficas. En cuanto a los eventos, se decidió en base a nuestra experiencia en programación de interfaces gráficas, soportar los que resultan de mayor interés en su manipulación.

4.2.1 Frame

Atributos: name, visible, selected, enabled, tooltiptext, image, text, color, size, location, anchor, scroll
Eventos: frameClosed, frameOpened, frameClosing, focusGained, focusLost, mouseClick, mouseDoubleClick
Componentes que puede contener: frame, label, button, textfield, combo, checkbox, radiobutton, textarea, menu, popupmenu.

4.2.2 Label

Atributos: name, visible, tooltiptext, image, text, font, color, size, alignment, location, anchor
Eventos: mouseClick, mouseDoubleClick
Componentes que puede contener: popupmenu.

4.2.3 Button

Atributos: name, visible, enabled, tooltiptext, image, text, font, color, size, alignment, location, anchor
Eventos: widgetSelected
Componentes que puede contener: popupmenu.

4.2.4 TextField

Atributos: name, visible, editable, enabled, echoChar, tooltiptext, text, font, color, size, alignment, location, anchor
Eventos: focusGained, focusLost, textValueChanged, widgetDefaultSelected
Componentes que puede contener: popupmenu

4.2.5 Combo

Atributos: name, visible, editable, enabled, tooltiptext, font, color, size, location, anchor
Eventos: focusGained, focusLost, textValueChanged, widgetSelected, widgetDefaultSelected
Componentes que puede contener: popupmenu

4.2.6 Checkbox

Atributos: name, visible, enabled, tooltiptext, selected, image, text, font, color, size, alignment, location, anchor
Eventos: widgetSelected
Componentes que puede contener: popupmenu

4.2.7 RadioButton

Atributos: name, visible, enabled, tooltiptext, selected, font, color, size, location, anchor
Eventos: widgetSelected
Componentes que puede contener: popupmenu

4.2.8 TextArea

Atributos: name, visible, editable, enabled, tooltiptext, text, font, color, size, alignment, location, anchor, scroll
Eventos: focusGained, focusLost, textValueChanged, widgetDefaultSelected
Componentes que puede contener: popupmenu

4.2.9 Menu

Atributos: name, enabled
Eventos: ninguno
Componentes que puede contener: menuItem, separator

4.2.10 MenuItem

Atributos: name, text, image, enabled, shortcut
Eventos: widgetSelected
Componentes que puede contener: menu

4.2.11 PopUpMenu

Atributos: name
Eventos: ninguno
Componentes que puede contener: menuItem, separator

4.2.12 Separator

Atributos: ninguno
Eventos: ninguno
Componentes que puede contener: ninguno

4.2.13 Componentes del alcance secundario.

Existe un conjunto de componentes que fue incluido en la definición del lenguaje, pero que se le asignó un orden de prioridad menor para la implementación, denominado alcance secundario. En este grupo se encuentran los siguientes componentes: *list*, *tabbedpane*, *tree*, *toolbar*, *scale*, *slider* y *progressbar*. En el documento de Alcance (**[Ref: Anexo D – Alcance - 5]**) se describe en detalle cada uno de estos.

4.3 Generador

El generador que forma parte de la herramienta, es un traductor en el sentido estricto del término, ya que traduce de un lenguaje fuente (GML) a un lenguaje objeto (Java). No se puede decir que es un compilador, debido a que no contiene todos los componentes que constituyen uno.

La decisión de que el generador tenga un diseño sencillo, se basó en que no era necesario agregarle todas las funcionalidades presentes en un compilador tipo, principalmente porque el análisis lexicográfico y sintáctico es realizado por el *parser* de XML, que valida el archivo de entrada en base a la especificación proporcionada del lenguaje.

Como complemento se incluyó un componente que permite agregar validaciones que no son contempladas por el *parser* de XML, y que puede ser extendido sin modificar su arquitectura, agregando las validaciones que se consideren necesarias. A modo de ejemplo, se implementó en éste una tabla de símbolos para evitar que se repitan nombres de variables. Otra validación posible sería controlar que los nombres de los componentes no coincidan con palabras reservadas. Este componente está diseñado de manera tal que los posibles errores encontrados por el *parser*, son integrados a los errores que resulten de esta última validación y se despliegan al usuario luego de concluido el procesamiento.

4.4 Utilización de la herramienta.

El modo de utilización de la herramienta por parte del usuario se describe en los siguientes pasos:

1. El usuario especifica la interfaz en un documento XML. Eventualmente, especifica el manejo de los eventos en uno o mas archivos de texto plano.
2. Utilizando el generador provisto con la herramienta, genera uno o varios archivos Java.
3. Compila el o los archivos Java obtenidos como salida del paso anterior.
4. Ejecuta la clase principal, que utilizando el entorno de ejecución provisto con la herramienta desplegará la interfaz.

5 DECISIONES TECNOLÓGICAS

5.1 Utilización de DTD o XML Schema para la especificación y validación de los documentos XML

Para tomar la decisión sobre el lenguaje a utilizar para la especificación y validación de los archivos XML, se realizó una comparación exhaustiva y ambientada en el contexto del proyecto, de las características de DTD y XML Schema presentadas en el punto 5 del documento Relevamiento del estado del arte (**[Ref: Anexo B - Relevamiento del estado del arte.doc – 5]**). Esta comparación derivó en varias conclusiones que son presentadas en el documento de Decisiones tecnológicas (**[Ref: Anexo C – Decisiones tecnológicas.doc – 1]**).

A continuación se presenta una síntesis de las conclusiones más relevantes:

1. Las restricciones de cardinalidad ofrecidas por DTD son suficientes para los requerimientos del proyecto. XML Schema ofrece adicionalmente restringir la cardinalidad a un número determinado de ocurrencias, pero esto no nos ofrece ninguna ventaja.
2. XML Schema provee control de tipos de datos básicos (string, integer, float, boolean, etc.), lo que facilita que alguno de los controles que nosotros tengamos que utilizar (sobre todo de enteros y booleanos) sean realizados en la validación del archivo y no en etapas posteriores.
3. Tanto DTD como XML Schema soportan enumeraciones (especificación de un valor a un conjunto restringido de valores).
4. La validación por contenido que soporta XML Schema, y no así DTD, nos aporta mucho porque nos permite restringir los identificadores de las variables, componentes, etc. de modo que estos sean sintácticamente correctos en Java (o cualquiera sea el lenguaje objetivo).
5. XML Schema soporta herencia. Permite extender otros documentos, permitiendo la reutilización y la refinación. Esto sería aprovechable en este proyecto considerando que la definición del lenguaje realizada dentro del mismo, no incluye todos los componentes gráficos existentes, pero podrían ser agregados en etapas posteriores.
6. Si bien es una realidad que XML Schema es de uso y soporte menos extendido, esto puede atribuirse a que es más reciente y a que presenta una complejidad adicional debido a su potencialidad, y no a que DTD aplique mejor en la mayoría de los casos.

Basándonos en estas conclusiones, es que se decidió a favor de la utilización de XML Schema en la validación de documentos XML, considerando el valor agregado que nos aporta.

5.2 Utilización de XML DOM o SAX como API para *parsear* los documentos XML

La decisión de que API utilizar en el procesamiento de los documentos XML, se realizó en base a una comparación de las características de XML DOM y SAX presentadas en el punto 9 del documento Relevamiento del estado del arte (**[Ref: Anexo B - Relevamiento del estado del arte.doc – 9]**), ambientadas en el contexto del proyecto.

El análisis de las características nos derivó en las siguientes conclusiones:

1. La forma de representación del documento XML ya sea mediante un árbol de nodos como lo hace XML DOM o un flujo secuencial de eventos en el caso de SAX, solo influye en la facilidad de uso.
2. En este proyecto el tratamiento que reciben los documentos es estrictamente secuencial y procesando todos los elementos por única vez, por lo tanto la ventaja que brinda XML DOM de acceso randómico y múltiple a los elementos no sería aprovechada.
3. Era necesario un modelo de datos propio que sirva de apoyo al procesamiento, por lo que el modelo provisto por XML DOM no sería aprovechado.
4. Sólo se descartarían los comentarios del archivo, por lo que la ventaja de utilizar SAX para descartar información no sería aprovechada.

5. El tamaño de los documentos manejados no es de relevancia, por lo que en este aspecto no incide utilizar uno u otro *parser*.
6. El archivo de entrada no sería modificado, por lo que la posibilidad de modificarlo que brinda XML DOM no sería aprovechada.
7. Tanto en XML DOM como en SAX era imprescindible escribir un mínimo de código para poder realizar el procesamiento (aunque probablemente SAX insuma mayor cantidad de líneas de código debido a que maneja el documento XML a mas bajo nivel).
8. El hecho de que XML DOM procese todo el documento antes de cargar el árbol, nos asegura que cuando se acceda a sus elementos, el documento ya fue validado. En SAX se puede procesar mientras se realiza el *parsing*, pero también la validación se realiza en paralelo, por lo que puede pasar que durante el procesamiento se llegue a que el documento no es válido y se interrumpa el procesamiento. La ventaja que se le encuentra a XML DOM es que no hay que "deshacer" las acciones realizadas durante el procesamiento, ya que nunca se iniciaría esta etapa si el archivo no es válido.
9. XML DOM tiene una API más funcional que facilita el reconocimiento y el acceso a los diferentes nodos del árbol. Podría ser de utilidad para el proyecto.

Analizando los puntos anteriores se concluyó que para este proyecto en particular, no hay ninguna característica de XML DOM o de SAX que aporte una ventaja significativa para el procesamiento de los documentos XML. En base a esto y a que el contexto del proyecto es de interés académico, se decidió utilizar SAX teniendo en cuenta que es de uso menos difundido. Siendo conscientes de la dificultad adicional que SAX podría significar, es que se estableció una fecha límite para la implementación en base a esta API. En caso de que esta fuera excedida, se utilizaría XML DOM. Felizmente, la implementación con SAX no representó mayores obstáculos, y los plazos se cumplieron exitosamente.

6 DEFINICIÓN DEL LENGUAJE GML

Esta sección esta dedicada a describir en forma breve los elementos del lenguaje GML. En el manual de usuario (**[Ref: Anexo G – Manual de usuario.doc – 4]**) se encuentra una descripción detallada de éstos, y se presentan ejemplos para cada caso. Lectores con conocimientos avanzados de XML Schema, podrán si así lo desean consultar directamente el archivo GMLSchema.xsd (**[Ref: Anexo J – GMLSchema.xsd]**), que contiene la definición formal del lenguaje.

El lenguaje GML permite definir una interfaz gráfica describiendo los componentes que se encuentran dentro de ésta, incluyendo sus propiedades y los eventos asociados.
Para la definición formal del lenguaje, se utilizaron tipos de datos, algunos de los cuales fueron definidos dentro del lenguaje, y otros tomados de XML Schema.

A continuación se presentan en primera instancia los tipos de datos utilizados, y luego se describen como están definidos los componentes GML, los eventos, el código embebido y la API definida por GML para cada componente.

6.1 Tipos básicos predefinidos de XML Schema.

XML Schema provee soporte para ciertos tipos de datos básicos, entre los que se encuentran integer, string y boolean. GML aprovecha la existencia de estos tipos básicos para definir otros tipo de datos mas complejos.

6.2 Tipos básicos GML

Definimos lo que denominamos tipos básicos GML, y son aquellos para los cuales su definición se basa en un tipo de dato básico predefinido de XML Schema (boolean, string, integer). En ocasiones consiste únicamente en el renombrado de un tipo básico predefinido de XML Schema, y en otras, es uno de estos tipos al que se le agregan ciertas restricciones. A modo de ejemplo, se definió un tipo básico GML de nombre sizeType, que define un integer con restricciones en los valores máximo y mínimo.

6.3 Tipos compuestos GML

Se definieron a su vez tipos compuestos GML, los cuales se utilizan para representar propiedades de componentes cuya naturaleza posee mas de una dimensión. Cada una de estas dimensiones se definen mediante un tipo básico predefinido de XML Schema, o un tipo básico GML. A modo de ejemplo, se definió el tipo compuesto GML sizeElementType, que consiste en las propiedades width y height, cada una de las cuales es de tipo sizeType.

6.4 Componentes

Se definieron los componentes soportados por GML. Un componente se define mediante en un conjunto de atributos simples y/o compuestos (explicados a continuación), componentes contenidos en este, y una sección dedicada al manejo de eventos.

Los atributos simples son aquellas propiedades cuyos valores son de algún tipo básico predefinido de XML Schema, o de algún tipo básico GML. A su vez, los atributos compuestos son aquellas propiedades de varias dimensiones que son de algún tipo compuesto GML.

6.5 Eventos

En la sección dedicada al manejo de eventos dentro de los componentes, se debe especificar el o los eventos que se desean procesar y el código que se desea ejecutar en caso de que estos ocurran. Esto se puede realizar mediante alguna de las dos modalidades provistas por GML: *code inline*, *code behind*.

- ***Code inline***: el código Java que manejará el evento en cuestión, es incluido dentro del documento XML.

- ***Code behind***: el código Java se especifica en un archivo independiente, y lo que se incluye dentro del documento XML, es una referencia a dicho archivo.

6.6 Código embebido

El código asociado al manejo de eventos puede ser potenciado mediante la utilización de los siguientes *tags*:

- **imports**: permite declarar paquetes que serán importados en la clase que maneja el evento
- **declarations**: permite declarar atributos en la clase que maneja el evento
- **constructor**: permite agregar código al constructor de la clase que maneja el evento

Esta misma funcionalidad también se ofrece dentro del *tag* raíz de la interfaz (gml), de forma tal que podamos importar paquetes, agregar atributos y agregar código al constructor de la clase principal que es generada.

El contenido de estos *tags* debe utilizar sintaxis Java.

6.7 Interfaz GML

Las interfaces generadas por GML son dinámicas, es decir, es posible alterar la presentación de la interfaz desde los manejadores de eventos. Un ejemplo habitual es habilitar (o deshabilitar) componentes, desplegar (u ocultar) nuevas opciones, en respuesta a determinadas acciones del usuario.

Para hacer posible esto, GML definió una interfaz con un conjunto de métodos para cada componente, que permite alterar el estado de los mismos. En el Manual de usuario se presenta esta interfaz en forma detallada (**[Ref: Anexo G – Manual de usuario.doc – 4.1.7]**).

La manera de hacer referencia a los componentes de la aplicación es mediante la palabra "frame" reservada por GML con este fin. Es decir, escribiendo *frame.miComponente* se accede al componente de nombre *miComponente.*

7 ARQUITECTURA

En este capítulo se describe la arquitectura del sistema. Esta fue estructurada tomando en cuenta los principales obstáculos que se debieron resolver. Para cada uno de éstos, se plantearon diversas soluciones, analizando ventajas y desventajas de cada una, concluyendo en la solución más adecuada. Esta discusión se describe completamente en el documento de Arquitectura (**[Ref: Anexo E – Arquitectura.doc – 2]**). Aquí, se presentan únicamente las soluciones elegidas y como estas cooperan para conformar la arquitectura del sistema.

7.1 Estructura general de paquetes

La arquitectura del sistema está compuesta por dos paquetes principales:

- **Generator**: paquete que a partir de un documento XML que contiene la especificación de una interfaz gráfica en lenguaje GML, genera código Java independiente de la librería gráfica.
- **Runtime**: paquete que permite ejecutar el código Java generado por el Generator, desplegando interfaces gráficas utilizando alguna de las librerías gráficas soportadas.

Si bien ocurre primero la generación de código y luego la ejecución, debido a que el Generator genera clases que utilizan la interfaz del Runtime, se explicará primero la estructura y el funcionamiento del Runtime, y posteriormente la del Generator.

7.2 Runtime

El paquete Runtime permite ejecutar el código Java generado, utilizando alguna de las librerías gráficas soportadas. La librería gráfica a utilizar la define el usuario a través de un parámetro al momento de ejecutar. Actualmente soporta las librerías gráficas Swing y SWT, y en caso de que se desee extender el soporte a otras librerías gráficas, puede realizarse siguiendo el instructivo indicado en el manual de desarrollo (**[Ref: Anexo H – Manual de desarrollo.doc – 3]**).

El diseño del Runtime es uno de los aspectos claves de la arquitectura del sistema. En particular, es fundamental la forma de representar en el Runtime los componentes gráficos, los eventos y los manejadores de eventos, de forma tal de que sea viable su implementación en varias librerías.

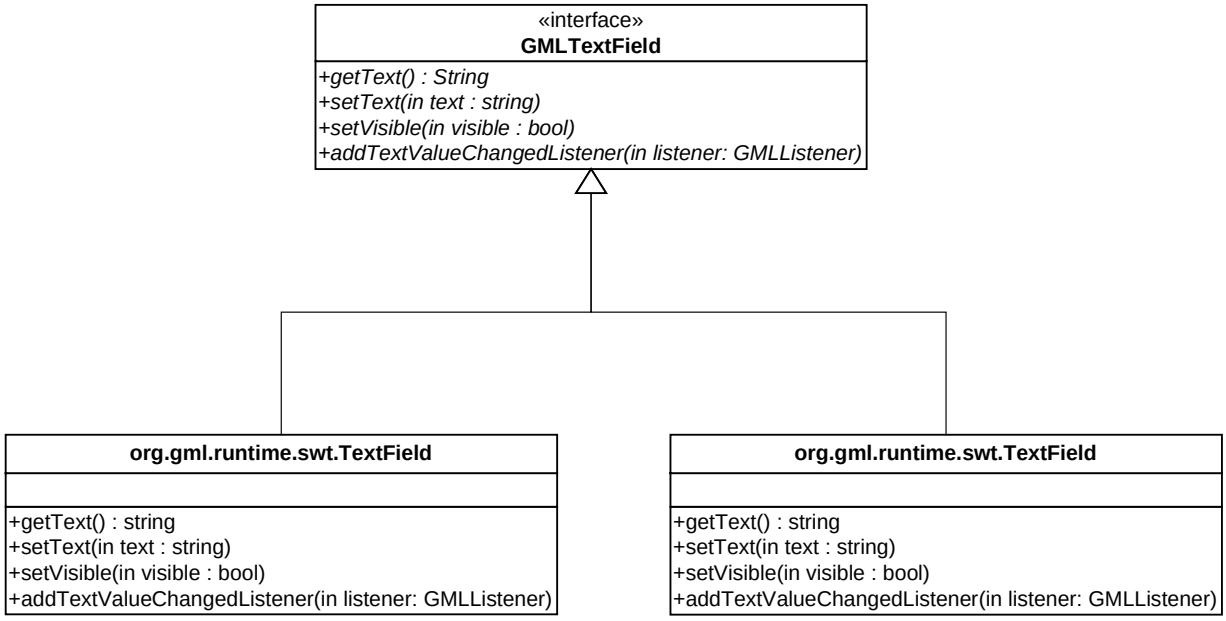
Otro motivo por el cual el Runtime cumple un papel clave en el diseño de la arquitectura, es que el código que escribe el usuario en los manejadores de eventos, eventualmente utiliza la interfaz del Runtime, y será ejecutado sobre él junto con el resto del código generado.

7.2.1 Componentes

Para representar cada componente existe una interfaz cuyo cometido es lograr la abstracción de la librería gráfica utilizada. Estas interfaces tienen los métodos necesarios para la manipulación del componente, lectura y escritura de sus propiedades, y métodos específicos para agregar los manejadores de eventos. Los nombres de estas interfaces deben respetar el siguiente patrón: GML<nombreComponente>, donde <nombreComponente> es el nombre del componente según su definición en el XML Schema (**[Ref: Anexo J – GMLSchema.xsd]**).

Cada librería que desee proveer soporte para un determinado componente, deberá crear una clase de nombre <nombreComponente> que implemente la interfaz asociada, GML<nombreComponente>. Todas las clases que proveen implementación para una misma librería gráfica deben estar agrupadas dentro de un mismo paquete.

A modo de ejemplo, existe una interfaz GMLTextField que representa al componente *TextField*, la cual es implementada por una clase de nombre TextField dentro del paquete org.gml.runtime.swt para la librería SWT, y otra de igual nombre dentro del paquete org.gml.runtime.swing para la librería Swing.



Por otra parte, las clases que corresponden a componentes sobre los que se pueda especificar un *layout* (actualmente todas excepto Frame, Popupmenu, Menu y MenuItem), además de implementar la interfaz correspondiente, deben heredar de la clase base abstracta GMLComponent, la cual les proveerá la implementación de un conjunto de métodos necesarios para el manejo del *layout*, logrando así liberar al usuario programador de esta difícil tarea, y unificar los comportamientos en esta área.

El componente Frame es un caso especial, debido a que es el componente "contenedor" del resto de los componentes, y ofrece un conjunto de métodos para el manejo del *layout* de los componentes contenidos en él. Nuevamente, esto libera al usuario programador de una difícil tarea, y unifica los comportamientos en esta área, independientemente de la librería que se utilice. Es por este motivo que para el componente Frame en lugar de utilizar una interfaz que lo represente, se utiliza una clase abstracta denominada GMLFrame, que tiene algunos métodos implementados y otros abstractos que deberán ser implementados por las clases derivadas. Esta misma situación se repite para todos los componentes que pueden contener a otros componentes, como es el caso de TabbedItem, para el cual se debería crear una clase abstracta GMLTabbedItem, y luego crear clases que hereden de ésta para cada librería.

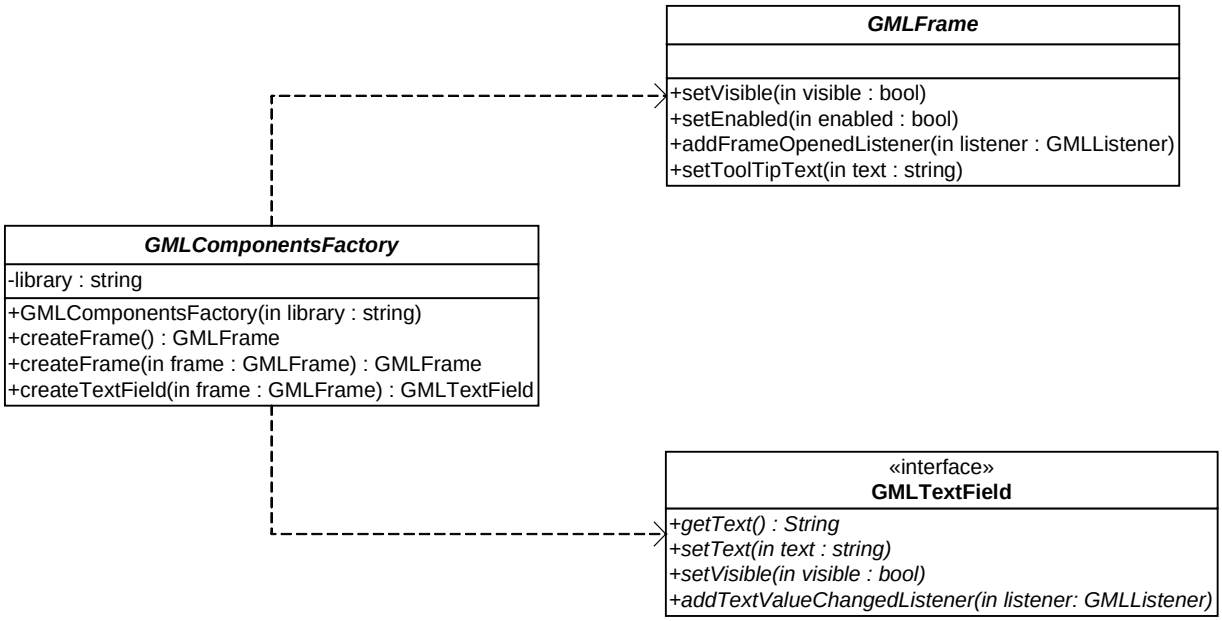
La idea de tener los componentes GML independientes de la librería, y las implementaciones en cada librería cumpliendo la interfaz definida por estos, permite definir la librería gráfica a utilizar en tiempo de ejecución.

La utilización del patrón de diseño *Factory Method* (**[Ref: 2.3]**) hace posible que cuando se crea un componente GML, éste sea instanciado con un objeto de la clase correspondiente a la librería elegida.

Se utiliza una sola clase Factory para crear a todos los componentes (GMLComponentsFactory) y se eligió la variante de dicho patrón que se basa en métodos parametrizados (*Parametrized Factory Method*) para definir la clase a instanciar. Para cada componente GML, esta clase tiene un método create<Component>() que creará por Reflection (**[Ref: 2.4]**) un objeto <Component> de la librería que se indique, y lo devolverá en un objeto del tipo GML<Component>.

Como la librería a utilizar será la misma para todos los componentes una vez definida, se creará un objeto de clase GMLComponentsFactory pasándole como parámetro la librería. Luego cuando se invoquen los métodos create<Component>() del objeto creado, se utilizará esta información para la creación por Reflection.

El uso de *Reflection* para la creación de los objetos correspondientes, hace de GML un sistema completamente extensible para nuevas librerías gráficas, sin necesidad de realizar modificaciones sobre el código base del sistema. Por este motivo es fundamental que para cada librería, en las clases que implementen las interfaces definidas, se respete la nomenclatura descrita y se siga la estructura de paquetes especificada.



7.2.2 Eventos y Manejo de Eventos

En esta sección se analiza la solución provista para permitir al usuario manejar los eventos GML de manera independiente a la librería grafica que se utilice, y referenciar los objetos de la interfaz en el código que escriba dentro del manejador.

7.2.2.1 Forma de referenciar a los componentes de la interfaz

Se analizaron diversas opciones, evaluando ventajas y desventajas que presenta cada una, decidiendo adoptar la siguiente propuesta.

Todos los componentes son definidos como atributos públicos de la clase principal, de forma tal que el usuario puede acceder a ellos utilizando el nombre que le asignó en la declaración. Esto derivó en la necesidad de definir un palabra reservada para hacer referencia a la clase principal. Se eligió la palabra *frame* con este cometido.

Como características de este diseño se destacan:

- Desde el punto de vista del encapsulamiento del paradigma de orientación a objetos, el hecho de que los atributos sean públicos no conlleva ninguna implicancia negativa, ya que esta clase no será utilizada por terceros.
- En la clase de los objetos que manejan los eventos, es necesario mantener una referencia a la clase principal. Esto se solucionó haciendo que el constructor de dicha clase reciba como parámetro esta

referencia. Este parámetro se almacena internamente en un atributo de nombre frame, de forma tal que el código que escribió el usuario en el manejador utilizando la palabra reservada *frame*, estará referenciando a la clase principal de la interfaz.

Del análisis comparativo con el resto de las opciones planteadas, surgieron las siguientes observaciones. Esta solución tiene la ventaja de que los errores que puedan existir en cuanto a los nombres de los componentes referenciados se detectan en tiempo de compilación (compilación Java de los archivos .java generados por GML). Por otro lado, presenta como limitación que no se pueden repetir nombres de componentes, aunque estos estén en diferentes "contenedores" de la interfaz.

7.2.2.2 Implementación de los manejadores de eventos independiente de la librería gráfica.

La idea adoptada para abstraer el manejo de eventos de la librería gráfica, es análoga a la aplicada para los componentes. Sin embargo existe una diferencia sustancial entre ambas, que radica en que para este caso las clases no existen con anterioridad, sino que son creadas por el Generator para cada interfaz en particular, según su especificación.

Para cada evento de un componente que el usuario quiere manejar, se generan las siguientes clases:

- Clase <nombreComponente>GML<nombreEvento>Listener que
  - hereda de la clase base abstracta GMLListener que representa a todas las clases que manejan eventos, y se utiliza como parámetro en los métodos que asocian instancias de estas clases (*listeners*) a los componentes.
  - recibe como parámetro en el constructor la referencia a la clase principal de la interfaz.
  - tiene un método execute() que contiene exactamente el código que escribió el usuario.

y a su vez, para cada librería que GML provea soporte, se genera:

- Clase <nombreComponente><nombreLibrería><nombreEvento>Listener que
  - hereda de <nombreComponente>GML<nombreEvento>Listener.
  - implementa la interfaz de la librería en cuestión asociada al evento.
  - en el constructor recibe como parámetro la referencia a la clase principal de la interfaz gráfica, e invoca al constructor de su clase base pasándole esta referencia.
  - en el método de dicha interfaz que responde al evento deseado, invoca al método execute () definido en la clase base.
  - y para el resto de los métodos de la interfaz brinda una implementación vacía.

Por otro lado, existe una clase abstracta GMLMain que se destina como clase base de la clase principal correspondiente a la interfaz. De esta forma se logra agrupar todas las clases principales bajo un mismo tipo, y se lo puede utilizar como parámetro en los constructores de las clases que manejan los eventos, que como ya se mencionó necesitan una referencia a ésta.

Análogamente al diseño elegido para los componentes, para determinar en tiempo de ejecución las clases a instanciar para el manejo de los eventos, se adoptó el patrón *Factory Method* en su variante *Parametrized Factory Method* (**Ref[2.3]**).

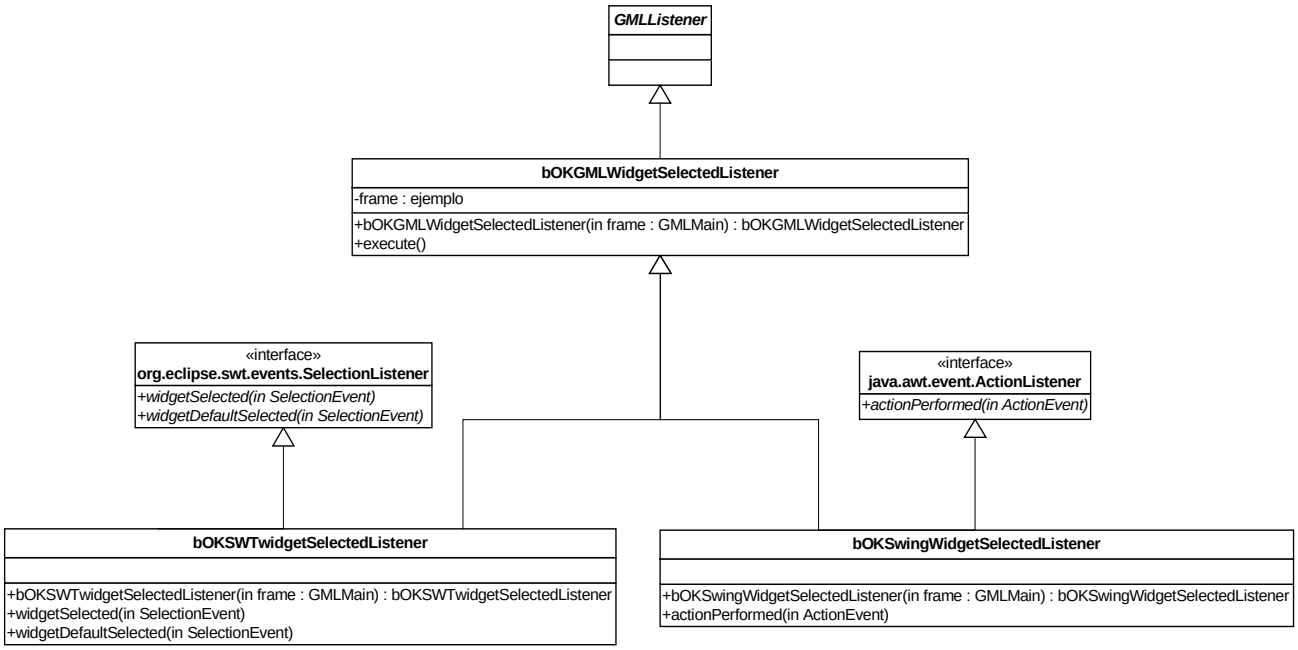
Se utiliza una sola clase Factory para crear todos los *listeners* (GMLListenersFactory). Para cada evento GML, esta clase tiene un método createGML<nombreEvento>Listener() que recibe como parámetro el nombre del componente, y crea mediante *Reflection* (**[Ref: 2.4]**) un objeto <nombreComponente><librería><nombreEvento>Listener de la librería que se indique, devolviéndolo en un objeto del tipo GML<Listener>.

Como la librería a utilizar será la misma para todos los *listeners* una vez definida, se creará un objeto de clase GMLListenersFactory pasándole como parámetro la librería. De igual forma, la clase principal es constante para una misma interfaz por lo que también se pasa una referencia a ésta como parámetro del constructor. Luego cuando se invoquen los métodos createGML<nombreEvento>Listener() del objeto creado, se utilizará esta información para la creación por *Reflection*.

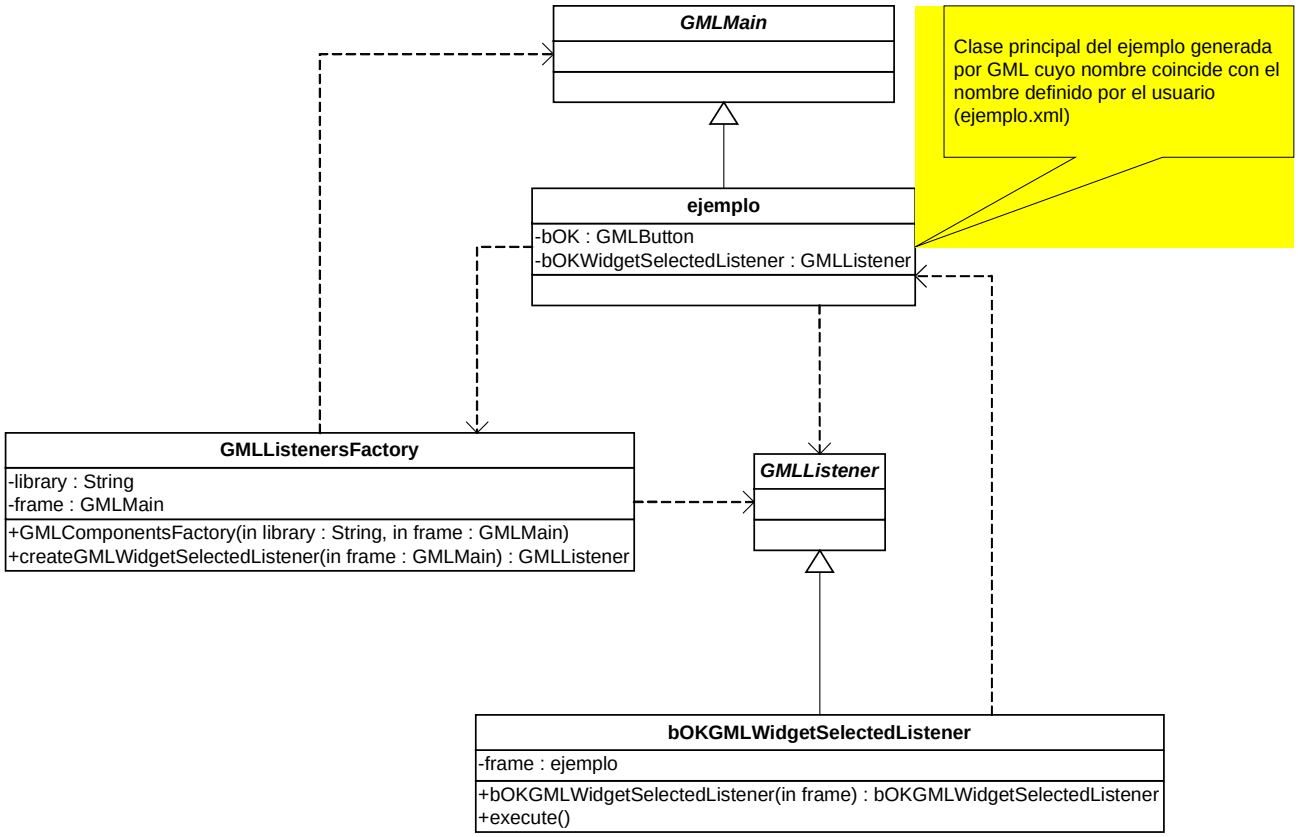
Los constructores de los *listeners* de cada librería (<nombreComponente><librería><nombreEvento> Listener) reciben como parámetro una referencia a la clase principal. Con esta referencia invocan al constructor de la superclase <nombreComponente>GML<nombreEvento>Listener, que lo almacena en un atributo de nombre frame, logrando así que las sentencias que están en el método execute() y usan la palabra reservada "frame" para acceder a los componentes de la interfaz, sean resueltas correctamente.

Por ejemplo, para un archivo de definición de interfaz de nombre ejemplo.xml, se define un botón de nombre "bOK" y se especifica que cada vez que ocurra el evento "widgetSelected" sobre el botón, se ejecute la siguiente sentencia "`System.out.println("Ocurrio un evento");`".

La estructura de clases que se genera para esta situación es la siguiente:



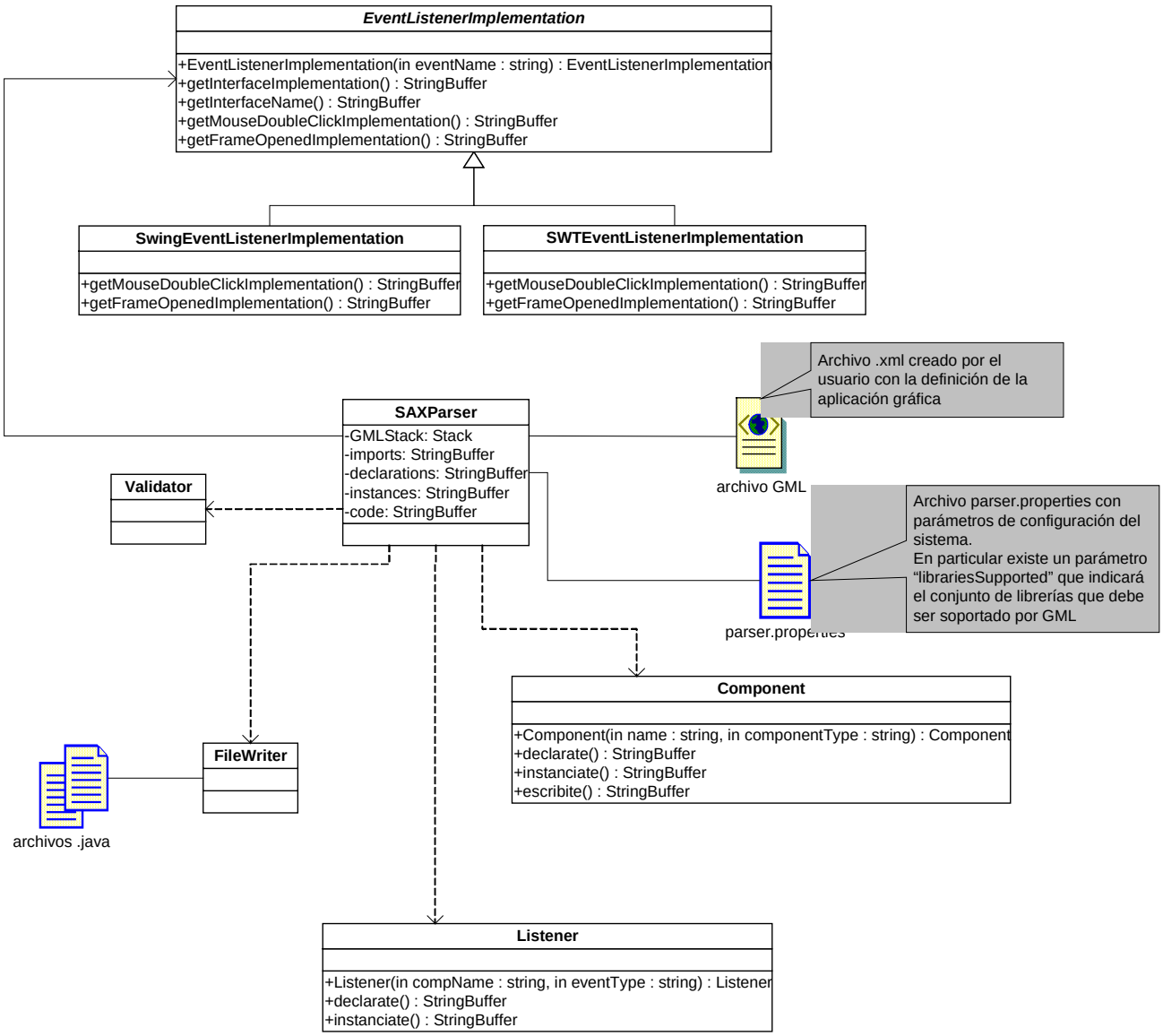
Por último, se presenta la interacción entre la clase principal (ejemplo), la clase *Factory* de *listeners* (GMLListenersFactory), y la clase que representa (abstrayendo de la librería gráfica) a las clases que manejan el evento (bOKGMLWidgetSelectedListener).


GMLMain
Clase principal del ejemplo generada por GML cuyo nombre coincide con el nombre definido por el usuario (ejemplo.xml)
ejemplo
-bOK : GMLButton
-bOKWidgetSelectedListener : GMLListener
GMLListenersFactory
-library : String
-frame : GMLMain
+GMLComponentsFactory(in library : String, in frame : GMLMain)
+createGMLWidgetSelectedListener(in frame : GMLMain) : GMLListener
GMLListener
bOKGMLWidgetSelectedListener
-frame : ejemplo
+bOKGMLWidgetSelectedListener(in frame) : bOKGMLWidgetSelectedListener
+execute()

7.3 Generator

A diferencia del Runtime, el Generator no presentaba dificultades que pudieran ser resueltas mediante el diseño de la arquitectura. Por este motivo, se buscó una arquitectura simple y eficaz, sin perder de vista la extensibilidad.

La dificultad mayor del Generator esta concentrada en el procesamiento de los archivos, es decir, lectura y *parseo* de los archivos XML, y generación de los archivos Java correspondientes.



7.3.1 Descripción de las clases

SAXParser: es el "motor" del generador. En esta clase se procesa el documento XML, se crean y se escriben los archivos .java que correspondan.

FileWriter: clase del paquete Java java.io que provee funcionalidades para la manipulación de archivos.

Validator: clase que permite realizar validaciones sobre los componentes definidos en el documento XML. Se provee para detectar características que no es posible controlar mediante XML Schema y que generarían un error al compilar (por ejemplo, dos componentes con el mismo nombre). Este componente puede ser extendido para agregar las validaciones que se consideren necesarias. En el manual de desarrollo se explican los pasos a seguir para extender esta funcionalidad, acompañado de un ejemplo (**[Ref: Anexo H – Manual de desarrollo.doc – 7]**).

Component: clase en la cual se almacena toda la información correspondiente a un componente, la cual sabe como escribir código ejecutable sobre el Runtime para su declaración e instanciación, y la asignación de sus propiedades.

Listener: clase en la cual se almacena toda la información correspondiente a un *listener*, la cual sabe como escribir código ejecutable sobre el Runtime para su declaración e instanciación.

EventListenerImplementation: esta clase es quien provee la abstracción de la librería gráfica, al momento de construir las clases que manejan los eventos especificados en la interfaz. Para esto define un conjunto de métodos abstractos cuya implementación en la clases derivadas, debe retornar el código completo de la clase que maneja el evento, de acuerdo a la librería que corresponda.

SWTEventListenerImplementation, SwingEventListenerImplementation: extienden de la clase abstracta EventListenerImplementation. Debe existir una clase de estas por cada librería que GML soporte. La convención para el nombre que estas deben tener es <nombreLibrería>EventListenerImplementation. Además, debe agregarse una referencia a éstas en el archivo parser.properties, para que sean tenidas en cuenta. El objetivo de estas clases es proveer la implementación de las interfaces de los *listeners* de la librería que representan.

Como cierre de esta sección, luego de haber descrito los dos paquetes que componen GML, Runtime y Generator, cabe destacar la absoluta independencia entre éstos, lo que permite la eventual sustitución de uno de ellos sin repercutir en el otro.

8 PLAN DE PRUEBAS DEL SISTEMA

El *testing* de la aplicación, no abarca pruebas unitarias ni pruebas de integración, sino que se enfoca directamente en la prueba del sistema.

La validación de los requerimientos descriptos en el documento de especificación de requerimientos (**[Ref: Anexo A - Requerimientos]**), se realiza conjunto a la ejecución de los casos de prueba, indicando según corresponda, el requerimiento que se esté validando. También se verifica el comportamiento de los componentes de acuerdo al *layout* especificado, según lo descrito en el documento de Alcance (**[Ref: Anexo D – Alcance - 6]**).

En el documento de *testing* (**[Ref: Anexo F – Testing del sistema]**) se presentan los casos de pruebas ejecutados, junto a los resultados esperados y los que se obtuvieron en cada ejecución, agregando observaciones en los casos que el resultado obtenido no coincidía con el esperado.

En esta sección se presenta la metodología utilizada, y se puntualizan todos los casos de prueba realizados, culminando con un análisis general de los resultados obtenidos.

8.1 Metodología

La metodología aplicada para realizar el *testeo* del sistema es la siguiente:

- Se establece el requerimiento funcional a *testear*.
- Se define un caso de prueba para dicho requerimiento.
- Se crea un documento XML que se corresponda con el caso de prueba definido.
- Se lleva a cabo el proceso de generación indicando como entrada el documento XML, y el paquete de salida al que pertenecerán las clases generadas.
- Se controlan los posibles mensajes de error y/o excepciones que puedan reportarse durante la generación. En caso de ocurrir se da por terminado el caso de prueba dejando constancia de los errores obtenidos.
- Se compila el paquete generado. En caso de obtener algún error se da por terminado el caso de prueba y se toma nota de los errores obtenidos.
- Se ejecuta la clase principal generada, indicándole como parámetro la librería a utilizar. En caso de obtener errores y/o excepciones durante la ejecución se da por terminado el caso de prueba y se toma nota de los errores obtenidos (*).
- Se controla el normal comportamiento de la interfaz generada (*).

(*) Se repite una vez por cada librería que sea soportada por el sistema.

8.2 Casos de prueba

8.2.1 Requerimientos No Funcionales

- **Generar código independiente de la plataforma:** los casos de prueba son ejecutados en plataforma Linux y Windows.

- **Diseñar una arquitectura que permita la abstracción de la librería gráfica utilizada:** la generación de cada una de las interfaces correspondientes a los casos de prueba descriptos mas adelante, se realiza utilizando las dos librerías que GML soporta actualmente: SWT y Swing.

8.2.2 Requerimientos Funcionales

- **Generar interfaces de usuario:** generar cada uno de los componentes soportados por GML.

1. Generar un *frame* vacío.

2. Generar un *frame* con una *label*.
3. Generar un *frame* con un botón.
4. Generar un *frame* con un *textfield*.
5. Generar un *frame* con un *textarea*.
6. Generar un *frame* con un menú.
7. Generar un *frame* con un *checkbox*.
8. Generar un *frame* con un *radiobutton*.
9. Generar un *frame* con un combo.
10. Generar un *frame* con un *popupmenu*

- **Permitir asociar código a sus documentos XML:** asociar código a los documentos XML en alguna de las dos modalidades provistas.

11. Asociar código *inline*.
12. Asociar código *behind*.

- **Permitir especificar el comportamiento de las interfaces en el documento XML:** procesar todos los eventos que soporta cada uno de los componentes.

13. Generar un *frame* vacío, que responda a los eventos frameOpened, frameClosing y frameClosed, imprimiendo un mensaje en la salida estándar para cada caso.
14. Generar un *frame* con una *label* que responda al evento mouseClick, imprimiendo un mensaje en la salida estándar.
15. Generar un *frame* con un botón que responda al evento widgetSelected, imprimiendo un mensaje en la salida estándar.
16. Generar un *frame* con un *textfield* que responda a los eventos focusGained, focusLost, textValueChanged y widgetDefaultSelected, imprimiendo un mensaje en la salida estándar para cada caso.
17. Generar un *frame* con un *textarea* que responda a los eventos focusGained, focusLost, textValueChanged y widgetDefaultSelected, imprimiendo un mensaje en la salida estándar para cada caso.
18. Generar un *frame* con un menú que contenga un *menuItem* que responda al evento widgetSelected, imprimiendo un mensaje en la salida estándar.
19. Generar un *frame* con un *checkbox* que responda al evento widgetSelected, imprimiendo un mensaje en la salida estándar.
20. Generar un *frame* con un *radiobutton* que responda al evento widgetSelected, imprimiendo un mensaje en la salida estándar.
21. Generar un *frame* con un combo que responda a los eventos focusGained, focusLost, textValueChanged y widgetDefaultSelected, imprimiendo un mensaje en la salida estándar para cada caso.
22. Generar un *frame* con un *popupmenu* que contenga un *menuItem* que responda al evento widgetSelected, imprimiendo un mensaje en la salida estándar.

- **Generar interfaces dinámicas:** en tiempo de ejecución modificar todas las propiedades para cada uno de los componentes.

23. Generar un *frame* con un botón que procese el evento widgetSelected, y modifique todas las propiedades del *frame* accesibles mediante la interfaz que provee GML: color, size, location, toolTipText, image, text, enabled y visible.
24. Generar un *frame* con dos botones, donde uno de ellos procese el evento widgetSelected, y modifique todas las propiedades del otro accesibles mediante la interfaz que provee GML: alignment, anchor, color, enabled, font, size, image, location, size, text, tootlTipText y visible.
25. Generar un *frame* con un botón y una *label*, donde el botón procese el evento widgetSelected, y modifique todas las propiedades de la *label* accesibles mediante la interfaz que provee GML: alignment, anchor, color, font, image, location, size, text, toolTipText, visible.

26. Generar un *frame* con un botón y un *textfield*, donde el botón procese el evento widgetSelected, y modifique todas las propiedades del *textfield* accesibles mediante la interfaz que provee GML: alignment, anchor, color, echochar, editable, enabled, font, location, size, text, toolTipText, visible.
27. Generar un *frame* con un botón y un *textarea*, donde el botón procese el evento widgetSelected, y modifique todas las propiedades del *textarea* accesibles mediante la interfaz que provee GML: alignment, anchor, color, editable, enabled, font, location, size, text, toolTipText, visible.
28. Generar un *frame* con un botón y un menú, donde el botón procese el evento widgetSelected, y modifique todas las propiedades del menú accesibles mediante la interfaz que provee GML: enabled, image, shortCut, text, toolTipText, visible.
29. Generar un *frame* con un botón y un *checkbox*, donde el botón procese el evento widgetSelected, y modifique todas las propiedades del *checkbox* accesibles mediante la interfaz que provee GML: alignment, anchor, color, enabled, editable, font, image, location, size, text, toolTipText, visible.
30. Generar un *frame* con un botón y un *radiobutton*, donde el botón procese el evento widgetSelected, y modifique todas las propiedades del *radiobutton* accesibles mediante la interfaz que provee GML: alignment, anchor, color, enabled, editable, font, image, location, size, text, toolTipText, visible.
31. Generar un *frame* con un botón y un combo, donde el botón procese el evento widgetSelected, y modifique todas las propiedades del combo accesibles mediante la interfaz que provee GML: alignment, anchor, color, enabled, font, image, location, size, text, toolTipText, visible.

- ***Layout* de los componentes:** controlar el correcto funcionamiento de las opciones mas comunes de *layout* para cada uno de los componentes.

32. Generar un *frame* con un *textarea* y un botón que no cambien ni su ubicación ni su tamaño al redimensionar el *frame* utilizando el *anchor* por defecto.
33. Generar un *frame* con un *textarea* y un botón que mantengan la proporción de distancias respecto al *frame* y la proporción de tamaño utilizando el valor de *anchor* proporcional respecto a todos los bordes del *frame*.
34. Generar un *frame* con un *textarea* que aumente su tamaño a medida que el *frame* lo aumenta, manteniendo las distancias constantes respecto a los cuatro bordes del *frame*, y un botón que mantenga su tamaño y mantenga distancia constante respecto a los bordes inferior y derecho del *frame*.
35. Ídem al caso de prueba 34 agregando un *textfield* que aumente su ancho proporcionalmente al ancho del *frame* y un *checkbox* que mantenga su tamaño y mantenga distancia constante respecto a los bordes inferior e izquierdo del *frame*.
36. Ídem al caso de prueba 35 agregando un combo que mantenga su tamaño y mantenga distancia constante respecto a los bordes superior y derecho del *frame*, que aumente su ancho proporcionalmente al ancho del *frame* y un *radiobutton* que aumente su altura proporcionalmente a la altura del *frame*.

8.3 Resultados obtenidos

En esta sección se detallan los resultados finales obtenidos para cada uno de los requerimientos planteados.

Se generaron satisfactoriamente todos los componentes incluidos en el alcance definido. La asociación de código en sus dos modalidades (*code inline* y *code behind*) funcionó como se esperaba. Asimismo, se verificó el correcto funcionamiento de la asociación de código para el manejo de todos los eventos, excepto en los siguientes casos:

- evento frameClosing en la librería SWT
- evento textValueChanged sobre el Combo en la librería Swing
- evento widgetDefaultSelected sobre el TextArea en las librerías SWT y Swing

La generación de interfaces dinámicas resultó exitosa en lo que respecta a la modificación en tiempo de ejecución del estado de los componentes. De todos modos, durante la verificación de este requerimiento se encontró que para algunos componentes, la asignación de algunas de sus propiedades no resultó como se esperaba. Esto en algunos casos se debe a que estas propiedades están incluidas en

el lenguaje GML y no así dentro de la API de las librerías. En otros casos, si están incluidas dentro de la API, pero esta no se comporta como lo indica su especificación, repercutiendo en el comportamiento de los componentes GML.

Los resultados obtenidos para el manejo del *layout* son satisfactorios, pero se debe señalar que los casos implementados no contemplan todas las combinaciones posibles.

En lo que refiere a los requerimientos no funcionales, se verificó que el código generado ejecuta tanto en la plataforma Linux como en Windows, generando interfaces en cualquiera de las dos librerías que se elija. Se presentan algunas diferencias menores (como puede ser el color adoptado por los botones) en distintas plataformas para la librería SWT, debido a que utiliza librerías nativas de la plataforma.

Como conclusión principal de las pruebas realizadas, se obtiene que el camino elegido fue adecuado, la herramienta funciona de acuerdo a lo esperado, y se puede seguir extendiendo y perfeccionando utilizando el modelo aplicado en este proyecto.

Es cierto que aún quedan detalles por corregir, y algunas funcionalidades que no fueron implementadas en su totalidad, como es el caso de la propiedad *anchor* de los componentes en el *layout*, en el que no se cubren exhaustivamente las combinaciones posibles, pero esto no implica una limitación de la herramienta, sino que con agregar mas horas de programación se pueden subsanar.

9 CONCLUSIONES Y TRABAJOS FUTUROS

9.1 Conclusiones

El proyecto se evalúa positivamente considerando que se cumplió el alcance primario definido; se implementaron para las librerías Swing y SWT todos los componentes que forman parte de éste, incluidos todos los atributos y eventos de cada uno. Es importante mencionar que todo el proyecto se realizó cumpliendo con las etapas y el cronograma establecidos.

Como tarea pendiente, queda la implementación del alcance secundario, que no se realizó para cumplir con el cronograma fijado, pero se considera que teniendo la arquitectura definida y los componentes del alcance primario implementados, no debería presentar nuevos obstáculos o dificultades.

Como características del producto final, se destacan el manejo del *layout* propuesto, en donde se incursionó en conceptos innovadores para la plataforma Java, y el hecho de que el código generado es independiente de la librería que se utilice en tiempo de ejecución, sobrepasando los requerimientos planteados al inicio del proyecto.

Desde el punto de vista del equipo de proyecto, se destacan los conocimientos adquiridos en las diversas etapas por las que se transcurrió. Entre ellos, la profundización en XML y sus tecnologías relacionadas, en particular XML Schema y SAX.

9.2 Trabajos futuros

Existen diversas ideas que surgieron durante el transcurso del proyecto, algunas de ellas para crear accesorios para la herramienta, y otras que la potencien o que imiten su modelo para producir herramientas similares. A continuación se mencionan las más importantes.

- Crear *plugins* para IDEs de desarrollo: incorporar la opción de crear documentos GML en IDEs de desarrollo (por ejemplo NetBeans, Eclipse).
- Integración de otras librerías gráficas.
- Construir herramientas similares para otros lenguajes (C#, Phyton, etc.).
- Permitir la utilización de variables en la definición de la interfaz.

10 REFERENCIAS

1. Brigham Young University. 2002. Último acceso: 20/03/2004. *Algorithmic Languages. Compiler textbook - Overview*
   http://topaz.cs.byu.edu/cs431/html/text/Ch1.PDF

2. Dale Green. Último acceso: 20/03/2004. *Trail: The Reflection API.* java.sun.com
   http://java.sun.com/docs/books/tutorial/reflect/

3. DAML (The DARPA Agent Markup Language Homepage). 2002. Último acceso: 15/03/2004. *Language Feature Comparison.* http://www.daml.org/language/features.html#types

4. David M. Geary. 2001. Último acceso: 20/03/2004. *Graphic Java 2, Mastering the JFC: AWT, Volume 1.* java.sun.com
   http://java.sun.com/developer/Books/GJ21AWT/

5. eclipse.org. Último acceso: 20/03/2004. *Eclipse Platform API Specification. Package org.eclipse.swt.widgets*. eclipse.org
   http://www.eclipse.org/documentation/html/plugins/org.eclipse.platform.doc.isv/doc/reference/api/org/eclipse/swt/widgets/package-summary.html

6. eclipse.org. Último acceso: 20/03/2004. *Eclipse Platform API Specification. Package org.eclipse.swt.events*. eclipse.org
   http://www.eclipse.org/documentation/html/plugins/org.eclipse.platform.doc.isv/doc/reference/api/org/eclipse/swt/events/package-summary.html

7. eNode. Último acceso: 20/03/2004. *XML Middleware For Dynamic User Interfaces and Seamless Web – Desktop Integration*
   http://www.enode.com

8. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. 1994. *Design Patterns CD. Elements of Reusable Object Oriented Software*. Addison – Wesley Longman, Inc.

9. Eric van der Vlist. 2001. Último acceso: 15/03/2004. *Using W3C XML Schema.* O'Reilly xml.com.
   http://www.xml.com/pub/a/2000/11/29/schemas/part1.html

10. GLADE. Último acceso: 20/03/2004. *GTK+ User Interface Builder*
    http://glade.gnome.org

11. Ira Pohl. 1993. *Object – Oriented Programming Using C++*. The Benjamin/Cummings Publishing Company, Inc.

12. Java Technology. Último acceso: 20/03/2004. *Creating a GUI with JFC/Swing Lesson: Learning Swing by Example*
    http://java.sun.com/docs/books/tutorial/uiswing/learn/index.html

13. Java Technology. Último acceso: 20/03/2004. *Creating a GUI with JFC/Swing Lesson: Using Swing Components*
    http://java.sun.com/docs/books/tutorial/uiswing/components/components.html

14. Java Technology. Último acceso: 20/03/2004. *Java 2 Platform. Package javax.swing*. java.sun.com
    http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/package-summary.html

15. JP Morgenthal. 2001. Último acceso: 15/03/2004. *A Brief Introduction to XML Schema*. XML.Gov
    http://xml.gov/presentations/xmlsolutions/

16. Mandar Chitnis, Pravin Tiwari, & Lakshmi Ananthamurthy. 2002. Último acceso: 15/03/2004. *Introduction to Web Services Part 3: Understanding XML.* developer.com. http://www.developer.com/services/print.php/10928_1557871_2

17. Monica Pawlan. 1999. Último acceso: 20/03/2004. *Essentials of the Java Programming Language: A Hands-On Guide, Part 2. Lesson 8: Object-Oriented Programming*. java.sun.com http://java.sun.com/developer/onlineTraining/Programming/BasicJava2/oo.html#what

18. Nazmul Idris. 1999. Último acceso: 15/03/2004. *SAX Tutorial.* developerlife.com http://www.developerlife.com/saxtutorial1/default.htm

19. Neil Deakin. 2003. Último acceso: 15/03/2004. XUL Planet. http://www.xulplanet.com/tutorials/xultu/

20. OOP-Reserch. Último acceso: 20/03/2004. *OOP XMLPanelEdit: Java Swing GUI tool / Java Swing GUI builder / Java Swing GUI by XML / XML Editor for Java Swing GUI* http://www.oop-reserch.com/xmlpanel_1_2.html

21. saxproject.org. Último acceso: 15/03/2004. *SAX.* http://www.saxproject.org

22. Simon Ritchie. 2002. Último acceso: 20/03/2004. *SWT – The Standard Widget Toolkit. Tucson Java Users Group* http://www.tucson-jug.org/presentations/SWT.pdf

23. SwiXML. Último acceso: 20/03/2004. *SwiXML 1.1* http://www.swixml.org

24. Universidad Autónoma Metropolitana. Último acceso: 20/03/2004. *Programación Orientada a Objetos con Java Curso SAI Capitulo 5 EL AWT (ABSTRACT WINDOWS TOOLKIT)* http://sai.uam.mx/apoyodidactico/po/Unidad5/poo5.html

25. W3 Schools. Último acceso: 15/03/2004. *XML Tutorial*. http://www.w3schools.com/xml/default.asp

26. W3 Schools. Último acceso: 15/03/2004. *XML DOM Tutorial*. http://www.w3schools.com/dom/default.asp

27. W3 Schools. Último acceso: 15/03/2004. *DTD Tutorial*. http://www.w3schools.com/dtd/default.asp

28. W3 Schools. Último acceso: 15/03/2004. *XML Schema Tutorial*. http://www.w3schools.com/schema/default.asp

29. W3C. Último acceso: 15/03/2004. *Document Object Model (DOM) Activity Statement.* http://www.w3.org/DOM/Activity

30. W3C. 2003. Último acceso: 15/03/2004. *Document Object Model FAQ* http://www.w3.org/DOM/faq.html#SAXandDOM

31. W3C. Último acceso: 15/03/2004. *Namespaces in XML. World Wide Web Consortium 14-January-1999* http://www.w3.org/TR/REC-xml-names/

32. WebReference.com. Último acceso: 15/03/2004. *SAX vs DOM* http://webreference.com/xml/column11/3.html

33. X-Smiles. Último acceso: 20/03/2004. *X-Smiles an open XML browser for exotic devices* http://www.xsmiles.org

34. XWT. Último acceso: 20/03/2004. *XWT – XML Windowing Toolkit* http://www.xwt.org

35. ZVON.org. Último acceso: 15/03/2004. *Mozilla Tutorial.*

http://www.zvon.org/download2.php/MozillaTutorial