Synth Editor

Desarrollo de una herramienta de personalización de interfaces gráficas en Java mediante SynthLookAndFeel

Proyecto de Sistemas Informáticos

Autores: **Nuria Hernández García, Antonio Leiva Gordillo, Miguel Ángel Ocaña Silva**

Profesores Directores: **Gonzalo Rubén Méndez Pozo, Virginia Francisco Gilmartín**

Facultad de Informática

Universidad Complutense de Madrid

Julio 2008

Resumen

Nuestra aplicación permite la creación de apariencias (*look & feels*) personalizadas mediante el empleo de la funcionalidad de Java denominada *Synth Look & Feel*.

Para ello, se requiere un archivo XML con una estructura muy definida en la que se le indican las personalizaciones que deberán mostrarse para cada uno de los componentes de las interfaces gráficas en Java. Con el XML y los archivos de imagen que queramos usar para cada uno, *Synth* construye una apariencia totalmente personalizada y definida para dichos componentes o regiones.

Nuestra aplicación se encarga de facilitar en gran medida la creación de dicho archivo XML, ofreciendo herramientas de uso muy fácil e intuitivo que permiten llevar a cabo todas las modificaciones posibles de cada componente. El usuario ya no necesita aprender ni conocer la estructura de dicho archivo, ni necesita escribirlo, lo que reduce en gran medida el número de errores y la velocidad de creación.

Summary

Our application lets the user create custom look and feels based on the Java functionality known as Synth Look & Feel.

In order to do this, it's required an XML file with a well defined structure where all the customizations that will be shown for each Java component are defined. With this XML and image files wanted for each one, *Synth* builds a totally customized and defined look for these components or regions.

Our application deals with easing as much as possible this XML generation, providing tools which are very easy to use and intuitive, which allows to carry out all the possible modifications for each component. Te user no longer needs to learn the structure of the XML file. They don't need either to write it, what reduces to a great extent the number or mistakes and the generation speed.

Palabras clave

Java, Synth, Look & Feel, apariencia, interfaz gráfica, XML, editor, usabilidad, software libre.

ÍNDICE

1. Introducción

Las interfaces de usuario son el vínculo principal entre la aplicación y el usuario final. La facilidad de manejo de las mismas determina en gran medida el número de usuarios que comprarán y utilizarán el producto, y por ello ha sido una de las mayores preocupaciones para las empresas de desarrollo software en los últimos años.

Las interfaces gráficas de aplicaciones complejas suponen un esfuerzo de desarrollo muy grande, ya que se requiere una estructura muy complicada debido al gran número de funcionalidades que deben permitir. A esto se une la ardua tarea de controlar cada interacción del usuario con la aplicación para no permitir ninguna acción prohibida o entrada de datos incorrecta.

Todas estas complejidades en el diseño de interfaces se han ido agilizando a lo largo de los años gracias a la aparición de nuevas herramientas como los editores de interfaces gráficas, o patrones de diseño específicos para esta parte del desarrollo, que logran limitar la aparición de problemas futuros que no se hayan previsto en el diseño.

Pero además de un manejo fácil, hoy en día los usuarios también demandan una apariencia agradable y vistosa. Es por eso que el tiempo de desarrollo de las interfaces está aumentando considerablemente, y cualquier herramienta que ayude a hacer más rápida esta labor supone un ahorro de tiempo, de recursos y económico para las empresas de desarrollo de software.

Nuestra aplicación centra sus esfuerzos en facilitar en la medida de lo posible la creación de una apariencia novedosa y totalmente personalizada en un tiempo mínimo, hasta ahora imposible de igualar con las herramientas disponibles.

1.1. Desarrollo de interfaces de usuario en Java

Ya desde su nacimiento, Java permitió la creación de interfaces de usuario mediante componentes de alto nivel, lo que supuso una gran revolución en el mundo de la informática, porque, aunque algunas herramientas permitían la creación de interfaces para otros lenguajes, ninguna poseía la potencia y facilidad de manejo que brindó el API de *Abstract Window Tookit* (AWT).

AWT fue la primera gran librería de creación de interfaces en Java, y como tal tenía algunas desventajas. Carecía de muchos componentes vitales en la creación de interfaces, lo que hacía complicada la representación de ciertas tareas.

Otra gran desventaja de AWT era que empleaba la apariencia del sistema operativo sobre el que corría, usaba los componentes nativos del sistema, y por tanto era imposible ningún tipo de personalización gráfica que estuviera fuera del ámbito de representación del sistema operativo. Esto incentivaba el uso de otros lenguajes como C++ en detrimento de Java, ya que C++ permite mucho más control sobre la interfaz de usuario, e incluso sobre la forma de las ventanas.

Por ello en la versión 1.2 de JDK (*Java Development Kit*) se incluye una nueva librería, Swing, que abrió un nuevo abanico de posibilidades.

Por un lado, se añaden varios nuevos componentes, aportando mucha mayor funcionalidad a las interfaces.

Por otro, Swing es completamente independiente del sistema operativo en el que se ejecute, ya que pinta sus propios componentes y no requiere de librerías nativas del sistema. Esta nueva característica aporta el concepto de "Look and Feel", lo que comúnmente se había conocido como "skins" o pieles, que permiten modificar la apariencia gráfica de la interfaz. Entre los *look and feel* que proporciona Java por defecto, siempre encontramos disponible uno que simula la apariencia del sistema operativo en la que corre el programa, además del llamado Metal Look and Feel (apariencia por defecto en Swing) y el Motif Look and Feel. Cabe destacar que en la actualidad se encuentra en fase de implantación otra nueva apariencia, llamada Nimbus, que verá la luz en las próximas actualizaciones de Java 6.0.

Pero las novedades introducidas no acaban ahí, ya que el nuevo API incluye una serie de clases e interfaces que permiten a los programadores crear sus propios l*ook and feel*, haciendo posible así que sus aplicaciones tomen la apariencia que ellos deseen.

El mayor problema en este punto es que crear una apariencia desde cero es una labor difícil, que sólo son capaces de llevar a cabo programadores muy experimentados y con un enorme conocimiento del funcionamiento de Swing, además de que el tiempo de creación del nuevo look and feel sería de meses, por lo que para las empresas no es nada rentable.

1.2. La nueva apariencia de Java: El Synth Look and Feel

En Java 5.0 aparece un nuevo look and feel: el Synth Look and Feel. Esta nueva apariencia suministra una nueva opción de personalización hasta entonces jamás vista en el mundo java.

Mediante un simple archivo XML (*Extensible Markup Language*) podemos detallar cada aspecto gráfico en nuestra aplicación. El XML es un metalenguaje extensible de etiquetas desarrollado por el *World Wide Web Consortium* (W3C).

¿Por qué se eligió el XML como medio? Básicamente porque los encargados de diseñar la apariencia de una aplicación suelen ser diseñadores gráficos, que no tienen por qué saber nada de programación, y un XML es uno de los documentos estructurados más simples que existen, que no requieren de ningún conocimiento de programación para ser aprendidos.

Esto reduce la creación de un look and feel personalizado de meses a semanas.

Aunque las posibilidades ahora son enormes, Synth no ha calado muy hondo entre los desarrolladores de software. Tan sólo hay dos colectivos (l2fprod y Synthetica), aparte del propio Sun, que han sabido sacar rendimiento a tan potente herramienta.

El motivo principal radica en que, aunque sólo es un XML, aún lleva mucho tiempo dominar su empleo.

Además, Sun no ha apostado lo suficiente por su expansión, y prácticamente no se puede encontrar información en la red, sólo algunas introducciones con pequeños ejemplos simples que en ningún caso son comparables a la complejidad que alcanzan los XML de una apariencia medianamente compleja.

El tiempo que hay que dedicar a la investigación no compensa, y prácticamente nadie se ha atrevido a ponerlo en práctica.

1.3. La apuesta de L2fprod: El Synth Builder

L2fprod, los creadores de Skin Look and Feel (un look and feel basado en Synth que permite la carga de skins), introdujeron una idea excelente: la creación de una aplicación que ayude al usuario a crear el XML. El usuario sólo tiene que decidir en la interfaz gráfica cómo serán sus personalizaciones, y es la aplicación la que se encarga de escribir el XML necesario para cargarlo en Synth.

Se trataba de una gran idea. Con esta aplicación, el desarrollo de una interfaz podría reducirse incluso a días, porque un diseñador que aprendiera a usarla no tendría ningún problema en materializar su creación a gran velocidad.

Pero sin ningún motivo aparente, este proyecto se desechó. La última versión disponible para descarga sólo es una interfaz sin ningún tipo de funcionalidad.

Aquí es donde surge nuestra idea de continuar la labor que quedó sin terminar, y que podría abrir muchas puertas al empleo de Synth Look and Feel como el mejor método para personalizar la apariencia de aplicaciones Java.

1.4. Synthetica

Synthetica es una herramienta gratuita para uso no comercial (o con licencia para uso comercial) que explota al máximo todas las ventajas de Synth, y que puede ser descarga de su página web[1].

El proyecto consiste en una serie de subclases que heredan de la jerarquía de *Synth* y que mejoran y añaden funcionalidad a las ya, como los menús traslúcidos, bordes redondeados, etc.

A la apariencia por defecto se suman un número creciente de *skins* que se van creando y mejorando a lo largo del tiempo. Aunque todas ellas son muy similares, varían sobre todo en la gama de colores que emplean.

[1] Sitio web de Synthetica: http://www.javasoft.de/jsf/public/products/synthetica

2. El Synth Look And Feel

2.1. Introducción

Como se comentó en la introducción, Synth apareció por primera vez en el JDK 5.0 para permitir a los programadores la creación de interfaces gráficas totalmente personalizadas a partir de un XML e imágenes.

Synth no está muy extendido entre la comunidad de desarrolladores, tanto por su complejidad, como por la falta de información, lo que posiblemente se vea solucionado en los próximos años.

2.2. Estructura

Sun proporciona una DTD (*Document Type Declaration*) para los documentos XML de Synth. Una DTD es un documento que determina la estructura y la sintaxis de un documento XML.

Para el caso de Synth, esta DTD impide que se carguen archivos XML que no sigan su estructura, o que contengan fallos de sintaxis. Cuando se asigna un XML a la interfaz, se comprueba internamente si dicho documento se ajusta a la DTD y se lanzará una excepción en el caso de que no sea así.

A continuación mostramos los detalles más importantes de dicha DTD, para su mejor comprensión. Es una versión traducida, pero se puede encontrar la original en la página oficial de Sun.[2]

Formato del archivo

El formato del archivo Synth.dtd[3] permite especificar todas las piezas necesarias para crear tu propio look and feel. Un archivo synth es cargado mediante el método SynthLookAndFeel.load[4]. El siguiente ejemplo usa el método “load” para configurar un SynthLookAndFeel y lo establece como el look and feel actual:

```
SynthLookAndFeel laf = new SynthLookAndFeel();
laf.load(MyClass.class.getResourceAsStream("laf.xml",
         MyClass.class));
UIManager.setLookAndFeel(laf);
```

Se realizará una validación sintáctica del XML mediante la DTD de Synth. El análisis fallará si un atributo necesario no es especificado, o de un tipo erróneo.

El elemento synth

```
<!ELEMENT synth ((%beansPersistance;) | style | bind | font | color |
                 imagePainter | imageIcon)*>
<!ATTLIST synth
          version              CDATA          #IMPLIED
```

[2] http://java.sun.com/j2se/1.5.0/docs/api/javax/swing/plaf/synth/doc-files/synthFileFormat.html
[3] http://java.sun.com/j2se/1.5.0/docs/api/javax/swing/plaf/synth/doc-files/synth.dtd
[4] http://java.sun.com/j2se/1.5.0/docs/javax/swing/plaf/synth/SynthLookAndFeel.html#load%28java.io.InputStream,%20java.lang.Class%29

```
>
```

Definición de atributos:

`version` Versión del formato de archive, debería ser 1

El elemento *synth* contiene el resto de elementos que conforman una definición del *SynthLookAndFeel*

El elemento style

```
<!ELEMENT style (property | defaultsProperty | state | font | painter
| imagePainter |backgroundImage | opaque | (%beansPersistance;) |
imageIcon)*>

<!ATTLIST style
          id              ID          #IMPLIED
          clone           IDREF       #IMPLIED
>
```

Definición de atributos:

`id` Identificador único para el estilo

`clone` Identificador de un estilo anteriormente definido que es copiado y usado para el nuevo estilo. Esto proporciona un cómodo mecanismo para sobreescribir sólo una porción de un estilo existente.

Un elemento “style” corresponde a un “SynthStyle”, cuyos elementos hijo especifican, o bien propiedades que se aplican a todos los estados, o elementos “state” que contienen propiedades específicas de un estado particular.

El siguiente ejemplo crea un estilo opaco con el identificador “button”, márgenes de 4, 4, 4, 4 y una fuente Dialog 12.

```
<style id="button">
  <opaque value="true"/>
  <insets top="4" left="4" right="4" bottom="4"/>
  <font name="Dialog" size="12"/>
</style>
```

El siguiente ejemplo crea un nuevo estilo con el identificador *clonedButton* que es una copia del estilo con identificador *button* y tiene una fuente Dialog 14. El estilo resultante sera *opaque*, tendrá márgenes 4, 4, 4, 4 y una fuente Dialog 14.

```
<style id="clonedButton" clone="button">
  <font name="Dialog" size="14"/>
</style>
```

El elemento state

```
<!ELEMENT state (color | font | painter | imagePainter |
(%beansPersistance;))*>
<!ATTLIST state
          id              ID              #IMPLIED
          clone           IDREF           #IMPLIED
          value           CDATA           #IMPLIED
          idref           IDREF           #IMPLIED
>
```

Definición de atributos:

`id` Identificador único para el estado.

`clone` Identificador de un estado previamente definido que es copiado y usado en el nuevo estado.

`value` Identifica el estado del componente para el que serán aplicadas las propiedades. Es una lista de: ENABLED, MOUSE_OVER, PRESSED, DISABLED, FOCUSED, SELECTED or DEFAULT. Los estados múltiples deberían ser separados por 'and'. Si no se especifica un valor, los contenidos se aplican a todos los estados.

`idref` Indica que este estado debería ser igual que otro estado definido previamente. Esto es útil cuando se desea que múltiples estilos compartan las mismas propiedades visuales para un estado particular.

El elemento *state* especifica las propiedades visuales que deben ser usadas para un estado particular de un componente. Por ejemplo, se podría especificar el color de fondo cuando el componente está desactivado.

Cabe destacar que no todos los componentes soportan todos los estados. Por ejemplo, un "Panel" sólo soporta los estados ENABLED y DISABLED.

El siguiente ejemplo crea un estado con fondo rojo que será usado cuando el componente esté en un estado seleccionado y pulsado:

```
<state value="SELECTED AND PRESSED">
  <color value="RED" type="BACKGROUND"/>
</state>
```

El estado con el mayor número de coincidencias individuales será el elegido. Por, ejemplo, lo siguiente define dos estados:

```
<state value="SELECTED and PRESSED" id="one">
  <color value="RED" type="BACKGROUND"/>
</state>

<state value="SELECTED" id="two">
  <color value="RED" type="BACKGROUND"/>
</state>
```

El estado "one" es usado cuando el componente está seleccionado y pulsado, y el estado "two" cuando el componente está seleccionado.

Si el estado del componente contiene al menos SELECTED and PRESSED, el estado "one" será seleccionado. Por otro lado, si el estado es SELECTED, pero no contiene PRESSED, el estado "two" será usado.

El elemento font

```
<!ELEMENT font EMPTY>
<!ATTLIST font
          id              ID              #IMPLIED
          clone           IDREF           #IMPLIED
          name            CDATA           #IMPLIED
          style           CDATA           #IMPLIED
          size            CDATA           #IMPLIED
>
```

Definición de atributos:

`id`	Identificador único para font
`idref`	Identificador de un font anteriormente definido.
`name`	Nombre del font
`style`	Estilo del font. Es una lista de estilos definidos por Font separados mediante espacios: PLAIN (simple), BOLD (negrita) o ITALIC (cursiva). Si no se especifica, se usará PLAIN.
`size`	Tamaño de la fuente, en píxeles.

El elemento Font define la fuente del *state* o *style* actual. Se deben especificar un *idref* o un *name*, y un *size*.

El siguiente ejemplo crea un estilo con una fuente Dialog 12 negrita.

```
<style id="test">
  <font name="DIALOG" size="12" style="BOLD"/>
</style>
```

El siguiente ejemplo crea un estilo con una fuente Dialog 12 negrita que será usada si el componente está ENABLED. En cualquier otro caso, usará una Dialog 12 cursiva.

```
<style id="test">
  <font name="DIALOG" size="12" style="ITALIC"/>
  <state value="ENABLED">
    <font name="DIALOG" size="12" style="BOLD"/>
  </state>
</style>
```

Mientras que se pueden suministrar distintas fuentes por estado, en general los componentes NO revalidan cuando el estado cambia, por lo que se puede incurrir en problemas de tamaño si se intenta usar una fuente con un tamaño significativamente diferente para distintos estados.

El elemento color

```
<!ELEMENT color EMPTY>
<!ATTLIST color
          id              ID              #IMPLIED
          idref           IDREF           #IMPLIED
          type            CDATA           #IMPLIED
          value           CDATA           #IMPLIED
>
```

Definición de atributos:

`id`	Identificador único para el color.

`idref` Identificador de un color previamente definido

`type` Describe dónde debería usarse el color. Típicamente es una de las constantes definidas por ColorType: FOREGROUND, BACKGROUND, TEXT_FOREGROUND, TEXT_BACKGROUND o FOCUS. Alternativamente se puede especificar una ruta completa a una clase y un campo, por ejemplo javax.swing.plaf.synth.ColorType.FOREGROUND. Esto es útil para subclases de Synth que definen tipos de color adicionales.

`value` Valor para el color. Tiene tres formas posibles:

- El nombre de una constante de la clase Color, por ejemplo RED.
- Un valor hexadecimal de la forma #RRGGBB donde RR especifica la componente roja, GG la verde y BB la azul.
- Un valor hexadecimal de la forma #RRGGBBAA, que es igual que #RRGGBB más un componente alfa (transparencia).

El elemento *Color* define un color y a qué parte del componente debería ser aplicado.

El siguiente ejemplo usará un color de fondo rojo cuando el componente esté activo

```
<state value="ENABLED">
  <color value="RED" type="BACKGROUND"/>
</state>
```

El siguiente ejemplo tendrá un fondo rojo cuando el componente esté activo, y si no será azul.

```
<style id="test">
  <state value="ENABLED">
    <color value="RED" type="BACKGROUND"/>
  </state>
  <state>
    <color value="#00FF00" type="BACKGROUND"/>
  </state>
</style>
```

El elemento property

```
<!ELEMENT property EMPTY>
<!ATTLIST property
          key   IDREF
#REQUIRED
          type  (idref|boolean|dimension|insets|integer)
"idref"
          value CDATA
#REQUIRED
>
```

Definición de atributos:

`key` Nombre de la propiedad

`type` Indica el tipo de propiedad

`value` Valor para la propiedad. Para propiedades booleanas será true o false, para propiedades enteras será un entero válido, para dimensiones será el ancho y el alto separado por un espacio, para márgenes serán valores numéricos para el superior, izquierdo, inferior y derecho separados por un espacio, y para propiedades idref será el identificador único de un objeto previamente definido.

Los elementos *Property* se usan para añadir pares clave-valor a un SynthStyle que puede ser accedido mediante un método get. Muchos componentes usan los pares clave-valor para configurar su apariencia visual.

Es recomendable consultar la *property table*[5] para ver la lista de propiedades que soporta cada componente.

A continuación se crean las propiedades ScrollBar.allowsAbsolutePositioning, OptionPane.minimumSize, ScrollPane.viewportBorderInsets, Tree.rowHeight y foreground con valores a falso, dimensiones de 262x90, márgenes 5, 5, 5, 5, el entero 20 y una instancia de la clase ArrowButtonPainter.

```
<style id="test">
      <property key="ScrollBar.allowsAbsolutePositioning"
      type="boolean" value="false"/>
      <property key="OptionPane.minimumSize" type="dimension"
      value="262 90"/>
      <property   key="ScrollPane.viewportBorderInsets"   type="insets"
      value="5 5 5 5"/>
      <property key="Tree.rowHeight" type="integer" value="20"/>
      <object class="ArrowButtonPainter" id="ArrowButtonPainter"/>
      <property key="foreground" type="idref"
      value="ArrowButtonPainter"/>
</style>
```

El elemento defaultsProperty

```
<!ELEMENT defaultsProperty EMPTY>
<!ATTLIST defaultsProperty
          key                                                    IDREF
#REQUIRED
          type                (idref|boolean|dimension|insets|integer)
"idref"
          value                                                  CDATA
#REQUIRED
>
```

[5] http://java.sun.com/j2se/1.5.0/docs/api/javax/swing/plaf/synth/doc-files/componentProperties.html

Definición de atributos:

`key`	Nombre de la propiedad.
`type`	Indica el tipo de propiedad.
`value`	Valor para la propiedad. Para propiedades booleanas será true o false, para propiedades enteras será un entero válido, para dimensiones será el ancho y el alto separado por un espacio, para márgenes serán el superior, izquierdo, inferior y derecho separados por un espacio y para propiedades idref será el identificador único de un objeto previamente definido.

Los elementos *DefaultsProperty* se usan para definir propiedades que serán insertadas en la tabla *UIDefaults* que *SynthLookAndFeel* proporciona al *UIManager*. El siguiente código asigna el color rojo al valor Table.*focusCellForeground*.

```
<style id="test">
  <object class="javax.swing.plaf.ColorUIResource" id="color">
    <int>255</int>
    <int>0</int>
    <int>0</int>
  </object>
     <defaultsProperty key="Table.focusCellForeground" type="idref"
     value="color"/>
</style>
```

Este valor podría ser pedido de la forma :
`UIManager.get("Table.focusCellForeground").`

El elemento graphicsUtils

```
<!ELEMENT graphicsUtils EMPTY>
<!ATTLIST graphicsUtils
          idref           IDREF             #REQUIRED
>
```

Definición de atributos:

`idref`	Identificador de un objeto SynthGraphicsUtils previamente definido que va a ser usado como el SynthGraphicsUtils para el *style* actual.

Los elementos *GraphicsUtils* son usados para definir el SynthGraphicUtils que el *style* actual usará. El siguiente ejemplo crea un estilo con una instacia de CustomGraphicsUtils para el SynthGraphicsUtils.

```
<style id="test">
  <object class="CustomGraphicsUtils" id="graphics"/>
  <graphicsUtils idref="graphics"/>
</style>
```

The insets element

```
<!ELEMENT insets EMPTY>
<!ATTLIST insets
          id            ID              #IMPLIED
          idref         IDREF           #IMPLIED
```

```
          top          CDATA          #IMPLIED
          bottom       CDATA          #IMPLIED
          left         CDATA          #IMPLIED
          right        CDATA          #IMPLIED
>
```

Definición de atributos:

`id`	Identificador único para Insets
`idref`	Identificador de unos Insets previamente definidos.
`top`	Componente superior de los márgenes
`bottom`	Componente inferior de los márgenes
`left`	Componente izquierdo de los márgenes
`right`	Componente derecho de los márgenes

Los elementos *Insets* son usados para definir los márgenes del *style* actual. Los márgenes serán establecidos para cualquier componente que esté asociado con dicho *style*. El siguiente ejemplo crea un estilo con insets 1, 2, 3, 0.

```
<style id="test">
  <insets top="1" bottom="2" left="3"/>
</style>
```

El elemento bind

```
<!ELEMENT bind EMPTY>
<!ATTLIST bind
          style        IDREF             #REQUIRED
          type         (name|region)     #REQUIRED
          key          CDATA             #REQUIRED
>
```

Definición de atributos:

`style`	Identificador único para un estilo previamente definido
`type`	Un nombre o una región. Para el tipo nombre se usa component.getName(). En el otro caso, se usa el nombre de la región.
`key`	Expresión regular aplicada al nombre del component, o el nombre de la región, dependiendo del valor de *type*.

Los elementos *bind* especifican qué regiones de un estilo deben ser usadas. El siguiente ejemplo aplica el estilo test a cualquier componente cuyo nombre comience por test.

```
<style id="test">
  <insets top="1" bottom="2" left="3"/>
</style>
<bind style="test" type="name" key="test.*"/>
```

Se pueden aplicar numerosos estilos a una *region*, en cuyo caso cada uno de los estilos que coincidan son fundidos en un estilo resultante, que será el que se use. Se da precedencia a los estilos definidos después en el archivo. Por ejemplo, el siguiente ejemplo define dos estilos, a y b. El estilo a es aplicado a cualquier componente con un nombre que comience por test, y el estilo b es usado para la región Button.

```
<style id="a">
  <font name="DIALOG" size="12" style="ITALIC"/>
```

```
  <insets top="1" bottom="2" left="3"/>
</style>
<bind style="a" type="name" key="test.*"/>
<style id="b">
  <font name="DIALOG" size="12" style="BOLD"/>
</style>
<bind style="b" type="region" key="button"/>
```

Para un botón con el nombre test, esto es equivalente a:

```
<style>
  <font name="DIALOG" size="12" style="BOLD"/>
  <insets top="1" bottom="2" left="3"/>
</style>
```

La fusión ocurre también para estados de un mismo estilo:

```
<style id="a">
  <font name="DIALOG" size="12" style="ITALIC"/>
  <insets top="1" bottom="2" left="3"/>
  <state value="ENABLED">
    <object id="customPainter" class="CustomPainter"/>
    <painter idref="customPainter"/>
  </state>
</style>
<bind style="a" type="name" key="test.*"/>


<style id="b">
  <font name="DIALOG" size="12" style="BOLD"/>
  <state value="ENABLED">
    <font name="Lucida" size="12" style="ITALIC"/>
  </state>
</style>
<bind style="b" type="region" key="button"/>
```

Para un botón con el nombre test, esto es equivalente a:

```
<style>
  <font name="DIALOG" size="12" style="BOLD"/>
  <insets top="1" bottom="2" left="3"/>
  <state value="ENABLED">
    <object id="customPainter" class="CustomPainter"/>
    <painter idref="customPainter"/>
    <font name="Lucida" size="12" style="ITALIC"/>
  </state>
</style>
```

El elemento painter

```
<!ELEMENT painter EMPTY>
<!ATTLIST painter
          idref                  IDREF          #IMPLIED
          method                 CDATA          #IMPLIED
          direction
(north|south|east|west|horizontal|vertical|horizontal_split|vertical_s
plit)          #IMPLIED
>
```

Definición de atributos:

`idref`	Identificador de un SynthPainter anteriormente definido.
`method`	Identifica el método SynthPainter que debe ser usado. El nombre corresponde al nombre de un método en SynthPainter con el prefijo paint eliminado y la siguiente letra en minúscula. Por ejemplo SynthPainter.paintButtonBackground está identificado por 'buttonBackground'. Si no se especifica, el painter es usado para todos los métodos que no tengan un painter específico.
`direction`	Identifica la dirección u orientación para la que se debe usar este painter. Sólo es útil para los métodos de SynthPainter que emplean una dirección u orientación. Si no se especifica, se usa para todas las direcciones.

El elemento Painter define un SynthPanter para el estado del estilo actual. El siguiente ejemplo asigna una instancia de la clase MyPainter que debe ser un SynthPainter del estilo test.

```
<style id="test">
  <object class="MyPainter" id="MyPainter"/>
  <painter idref="MyPainter"/>
</style>
```

El painter usado para un método y un estado particular es determinado de la siguiente manera:

1. Painter especificado para el estado y el método actual.
2. Painter especificado para estado actual.
3. Painter especificado para el estilo, método y dirección.
4. Painter especificado para el estilo y método.
5. Painter especificado para el estilo.

Considera lo siguiente:

```
<style id="test">
  <painter idref="fallbackPainter"/>
  <painter idref="styleButtonBackgroundPainter"
method="buttonBackground"/>
  <state value="SELECTED">
    <painter idref="stateFallbackPainter"/>
    <painter idref="stateButtonBackgroundPainter"
method="buttonBackground"/>
  </state>
</style>
```

La siguiente tabla resume qué painter será usado para qué método de SynthPainter:

Estado	Método	Painter
SELECTED	paintButtonBackground	stateButtonBackgroundPainter
SELECTED	Cualquiera excepto paintButtonBackground	stateFallbackPainter
Caulquiera excepto SELECTED	paintButtonBackground	styleButtonBackgroundPainter
Cualquiera excepto SELECTED	Cualquiera excepto paintButtonBackground	fallbackPainter

El elemento imagePainter

```
<!ELEMENT imagePainter EMPTY>
<!ATTLIST imagePainter
          id                    ID              #IMPLIED
          method                CDATA           #IMPLIED
          direction
(north|south|east|west|horizontal|vertical|horizontal_split|vertical_s
plit)          #IMPLIED
          path                  CDATA           #REQUIRED
          sourceInsets          CDATA           #REQUIRED
          destinationInsets     CDATA           #IMPLIED
          paintCenter           (true|false)    "true"
          stretch               (true|false)    "true"
>
```

Definición de atributos:

`id` identificador único para el imagePainter.

`method` Identifica el método SynthPainter que debe ser usado.
El nombre corresponde al nombre de un método en SynthPainter con el prefijo paint eliminado y la siguiente letra en minúscula.
Por ejemplo SynthPainter.paintButtonBackground está identificado por 'buttonBackground'.
Si no se especifica, el painter es usado para todos los métodos que no tengan un painter específico.

`direction` Identifica la dirección u orientación para la que se debe usar este painter.
Sólo es útil para los métodos de SynthPainter que emplean una dirección u orientación.
Si no se especifica, se usa para todas las direcciones.

`path` Ruta de la imagen.
Se usa el método getResource de Class para resolver la ruta, de la Class suministrada al SynthLookAndFeel.load.

`sourceInsets` Márgenes de la imagen fuente.
Son: superior, izquierdo, inferior, derecho, con cada componente separado por un espacio.

`destinationInsets` Márgenes de la imagen de destino.
Son: superior, izquierdo, inferior, derecho, con cada componente separado por un espacio.
Si no se especifican, se usan los *sourceInsets*.

`paintCenter` Si se debe dibujar o no el centro de la imagen.

`stretch` Si los componentes norte, sur, este y oeste de la imagen resultante deberían ser escalados (true) o embaldosados (false).

El elemento *ImagePainter* define un painter para el estilo o estado actual que renderizará una imagen dada.

El siguiente ejemplo establece una imagen que será usada para pintar cualquier componente que use el estilo test:

```
<style id="test">
  <imagePainter path="resources/myImage.png"
                sourceInsets="2 2 2 2"/>
</style>
```

Es recomendable consultar la descripción de *painter* para más detalles sobre la selección de éste.

El elemento imageIcon

```
<!ELEMENT imageIcon EMPTY>
<!ATTLIST imageIcon
          id                    ID              #REQUIRED
          path                  CDATA           #REQUIRED
>
```

Definición de atributos:

`id` Identificador único para el imageIcon.
`path` Ruta de la imagen. Se usa el método getResource de Class para resolver la ruta, de la Class suministrada al SynthLookAndFeel.load.

El elemento *ImageIcon* se usa para asignar una implementación de Icon que está encapsulando una Imagen a un identificador único. Esto se usa típicamente para propiedades que requieren un Icon.

El siguiente ejemplo asigna un ImageIcon a la propiedad RadioButton.icon.

```
<style id="test">
  <imageIcon id="icon" path="resources/myImage.png"/>
  <property key="RadioButton.icon" value="icon"/>
</style>
```

El elemento opaque

```
<!ELEMENT opaque EMPTY>
<!ATTLIST opaque
          value              (true|false)   "true"
>
```

Definición de atributos:

`value` Si el estilo debería ser opaco o no. Si no se especifica, el estilo será opaco.

El elemento *opaque* indica si un component al que el estilo está asociado debe ser opaco o no. Al painter se le pedirá pintar sin importar la opacidad del elemento asociado.

El siguiente ejemplo crea un estilo que no es opaco:

```
<style id="test">
  <opaque value="FALSE">
  <painter idref="painter"/>
</style>
```

La entidad beansPersistance

beansPersistance puede ser usado para incrustar cualquier objeto Java tipo *Object*. Esto se usa normalmente para añadir tus propios *Painters*, pero puede utilizarse para cualquier otro objeto que se desee.

3. Objetivos de trabajo

3.2. Usabilidad

La usabilidad es un aspecto muy importante en las aplicaciones de usuario de hoy en día. En la era actual cualquier persona tiene acceso a un ordenador, tenga muchos o pocos conocimientos de informática, y por tanto las aplicaciones deben estar preparadas para ello. Hay que tener en cuenta que es muy posible que una persona con bajos conocimientos en el manejo de ordenadores puede utilizar nuestro software, y por tanto hay que programarlo en consecuencia.

Las aplicaciones deben ser muy intuitivas, robustas, con lenguaje claro, conciso y útil para el usuario, y que permita aclarar cualquier duda en cualquier momento de manera rápida y sencilla. Además, se hace indispensable que la curva de aprendizaje del manejo de las mismas sea lo más corta y rápida posible.

Es por ello que el desarrollo de la interfaz gráfica es uno de los puntos más importantes y problemáticos de la actualidad en cuanto a aplicaciones de escritorio se refiere.

En nuestro caso particular, lo más probable es que la persona que lo utilice tenga al menos unos conocimientos medios de manejo de ordenadores y de teoría de interfaces gráficas. Lo que no está tan claro es que vaya a conocer todas las propiedades modificables de los componentes de las interfaces gráficas. Las primeras horas de uso de la aplicación requerirán un esfuerzo enorme para el usuario, y es por ello que hemos hecho todo lo posible para facilitar ese primer contacto.

Por eso decidimos investigar métodos para crear interfaces gráficas usables. Existe un proyecto conocido como Status desarrollado en la Universidad Politécnica de Madrid que ha definido una gran cantidad de patrones de diseño enfocados a la usabilidad, y es de ellos de donde tomaremos varias ideas[6]:

- **Diferentes idiomas:** nuestra aplicación permitirá la traducción de la misma de forma fácil y rápida. Para ello basta con tomar como ejemplo cualquier fichero de un idioma ya creado, traducir sus cadenas de texto y guardarlo con el nombre del idioma correspondiente. De esta forma, al ser un proyecto de software libre, cualquier persona interesada puede traducir la aplicación mediante sencillos pasos y sin necesidad de tener ningún conocimiento de programación.
- **Indicación del estado:** permite al usuario en todo momento saber el estado de la aplicación. Se suele representar mediante barras de estado, como por ejemplo en Word la barra que indica la página en la que se encuentra el documento actualmente. En nuestra aplicación también se mostrará información sobre el estado. Esto le indica al usuario en todo momento a qué componente y estado del componente se corresponden las propiedades que se están modificando. Esta información no es sólo útil, sino indispensable para que el usuario sepa en todo momento sobre qué propiedades está trabajando, ya que es fácil despistarse debido al gran número de posibilidades que abarca la aplicación.
- **Accesos rápidos:** permiten a los usuarios experimentados avanzar más rápido en su trabajo. Se han creado accesos rápidos para prácticamente todas las acciones posibles de

[6] Sitio web del proyecto Status: http://is.ls.fi.upm.es/status/

la interfaz, por lo que un usuario experimentado podrá realizar ediciones y accesos a la ayuda sin más que pulsar las teclas adecuadas. Esto permitirá ahorrar mucho tiempo a medida que se vaya cogiendo soltura con el manejo de la aplicación.

- **Validaciones de campo y formulario:** cuando un usuario introduce información, ésta se puede validar campo a campo o una vez que ha rellenado todo el formulario y pulsa aceptar. En nuestra aplicación se usan ambas validaciones dependiendo de la operación que esté realizando. Por ejemplo, cuando un usuario quiere crear un nuevo proyecto o cuando quiere abrir una imagen para utilizarla en un componente, la información no se valida hasta que el usuario no pulsa el botón Aceptar, por lo que se está realizando validación de formulario. Sin embargo, cuando se están introduciendo los valores de los márgenes de una imagen, los campos realizan validación uno a uno, y autocorrigen la entrada del usuario a lo más aproximado posible que sea un dato válido.
- **Deshacer y rehacer:** hoy en día son dos operaciones de edición muy comunes e indispensables en cualquier aplicación, y en la nuestra también cobra una importancia enorme. Puesto que la forma de trabajo de nuestra herramienta requiere hacer muchas prueba para comprobar la apariencia antes de decantarse por una solución definitiva, será muy normal realizar cambios para ver su resultado y posteriormente deshacerlos si no estamos conformes. Es por ello que ha sido uno de los patrones de usabilidad a los que más tiempo hemos dedicado, por su importancia y dificultad de unificación. Debe existir un mismo deshacer que cumpla distintas funciones dependiendo de cuál sea la operación que tengamos que deshacer, lo que requiere un alto poder de abstracción y una complejidad elevada.
- **Ayuda sensible al contexto:** La aplicación está provista de una ayuda rápida. Esta ayuda puede ser pulsada en cualquier momento y permite al usuario obtener información instantánea sobre la tarea que esté realizando en ese momento. Si por ejemplo el usuario desconoce la forma de uso de una propiedad para un componente específico, al pulsar el botón de ayuda se le muestra la información sobre dicha propiedad, lo que ayuda a encontrar de forma rápida y muy sencilla la información que queremos, ahorrando mucho tiempo de navegación sobre la ayuda estándar.
- **Ayuda estándar:** es la típica ayuda general que ofrecen prácticamente todas las aplicaciones de escritorio. El usuario podrá en todo momento acceder a dicha ayuda y obtener información sobre temas generales de uso de la aplicación.
- **Reutilización de información:** Nuestra aplicación permite operaciones como copiar, cortar y pegar, que permiten reutilizar datos de unos componentes y estados en otros aún por definir. De esta manera no se tendrán que repetir tareas que puedan llevar más tiempo, permitiendo al usuario avanzar más rápido en el proceso de creación de su apariencia personalizada.

3.3. Objetivos concretos

Al comienzo del proyecto nos marcamos varios objetivos que creíamos que serían vitales para tener una aplicación completa para la función que queríamos que desempañara. Estos objetivos son los siguientes:

- Creación de una aplicación que permita la personalización de todos los componentes de una interfaz gráfica de usuario realizada con Swing, mediante el empleo de la funcionalidad Synth Look and Feel que aporta el API de Java.

- Creación y mantenimiento de una web para unificar la poca información que existe en la red sobre Synth y añadir nueva información a partir de nuestras investigaciones. Esta web se podrá consultar en dos idiomas: inglés y español.
- La aplicación será un plugin de Eclipse, para poder unificar la creación de la aplicación y la apariencia de la interfaz gráfica personalizada.
- Se dotará de todas las funcionalidades de usabilidad que sea posible aplicar, como herramientas de edición, de deshacer, etc.
- La aplicación será provista de una ayuda inteligente, que permita en cualquier momento conocer información muy concreta de la tarea que estemos realizando, permitiendo así al usuario avanzar a mayor rapidez.
- Dejar un código abierto y preparado para que sea fácil su modificación y añadir nuevas funcionalidades. Para ello, se debe realizar un código limpio, bien estructurado, organizado y comentado.

4. Diseño

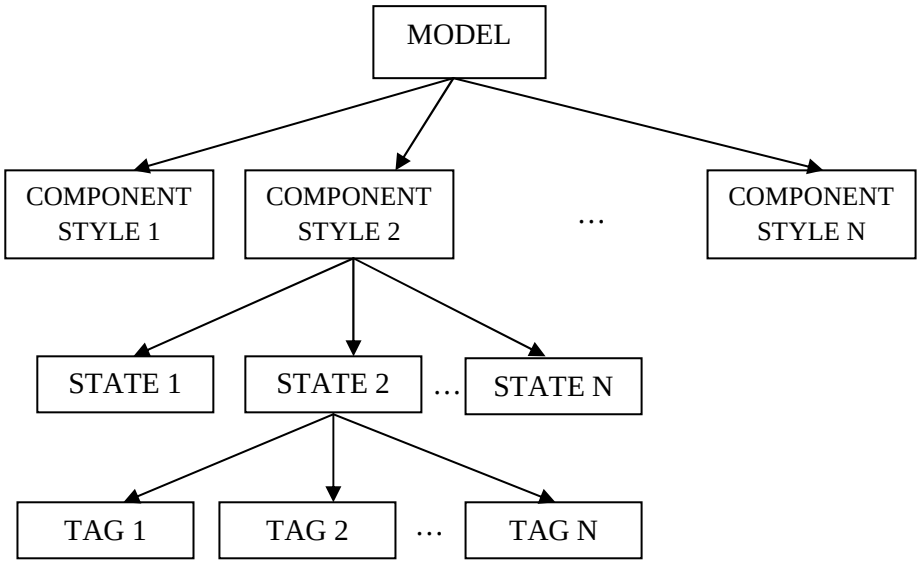
4.2. Estructura

La estructura de la aplicación está basada en el Modelo-Vista-Controlador, aunque con algunas particularidades. En ella, el controlador actúa de intermediario para manejar la comunicación entre el modelo de datos y su presentación en la vista.

Por tanto, dividiremos la estructura de nuestro proyecto en estas tres partes, que pasamos a comentar a continuación en detalle:

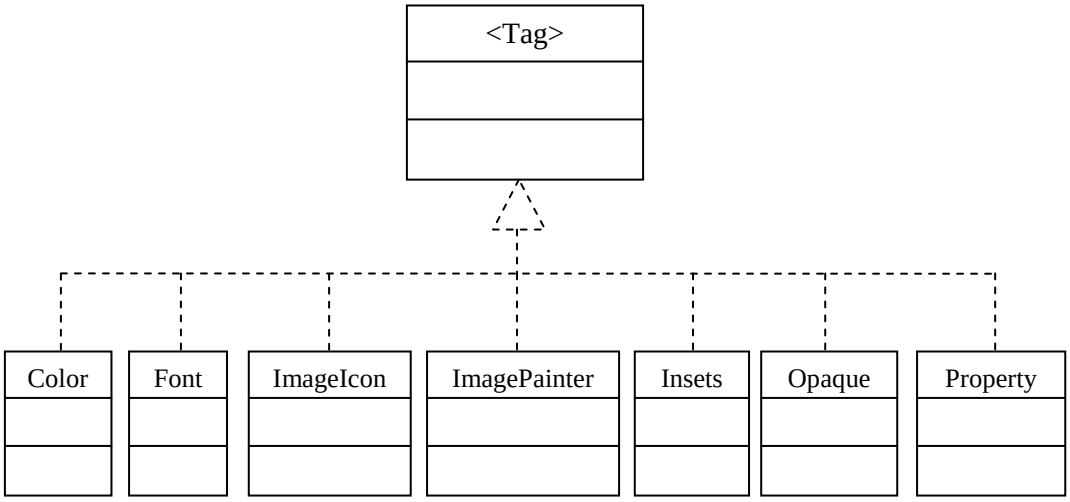
MODELO

El modelo (*Model*) almacena la estructura que permitirá generar el XML que contendrá la información necesaria sobre nuestro Look and Feel personalizado, se podría decir que es la representación del archivo UML.



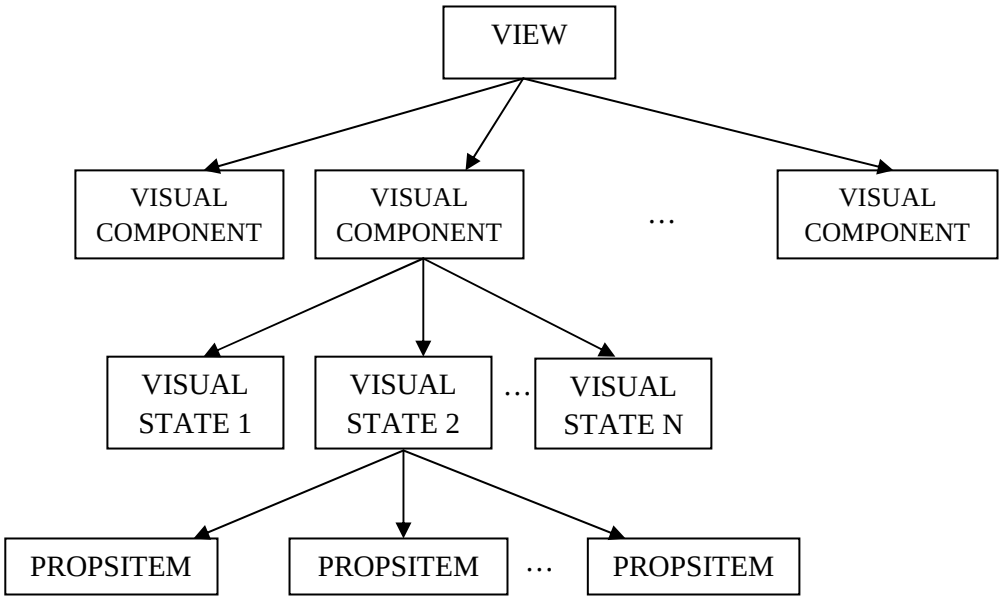
Está formado por un conjunto de estilos para componentes (*ComponentStyle*), que se corresponden con cada una de las posibles regiones de las que consta una interfaz gráfica (Botones, menús, listas, etc). Éstos a su vez mantienen una lista de los posibles estados (*State*) para un mismo componente (por ejemplo para el botón: por defecto, pulsado, enfocado, etc). A su vez cada estado está compuesto por una lista de tags (*Tag*) que se corresponden con las etiquetas XML, y que por lo tanto contienen la información que ha modificado el usuario para el estilo que está creando, y que debe ser reflejada en el archivo XML generado.

Tag es una interfaz que implementan todas las clases que modelan las etiquetas que están permitidas por la DTD de Synth.



VISTA

La vista está formada por una estructura compleja dividida en varios subpaquetes. Básicamente la vista maneja una estructura similar a la del modelo, pero basándose en una visión más intuitiva y dirigida al usuario final. También existe un árbol manteniendo la información que se muestra en la vista, pero organizado en propiedades en lugar de tags, ocultando al usuario que modifique una propiedad todos los cambios que se producirán en forma de Tag en el modelo (y por lo tanto en el XML).



En el primer nivel del árbol generado por la Vista, se encuentran los *VisualComponent*. Por cada región o componente Swing, existe una clase que hereda de *VisualComponent* y que se encarga de crear sus estados (*VisualState*).

También hay clases que heredan de *VisualState* para cada región, y que conocen las propiedades de su componente para recrearlas en la vista. Finalmente, cada VisualState contiene una lista de propiedades que están representadas por la clase *PropsItem*, de la cual nace una compleja estructura de herencia para que cada propiedad sepa cómo comportarse ante distintas llamadas a métodos.

El paquete view.*cellEdition* se encarga de la edición de propiedades. Cada propiedad tiene un tipo de celda, que, en la interfaz gráfica de usuario, es la que condiciona al árbol de presentación a mostrar un tipo de manejador u otro para modificar dicha propiedad. Existen, entre otros, manejadores que permiten seleccionar un archivo, modificar la cadena de texto, seleccionar una fuente y su color, etc.

El paquete *view.menu* coordina todos los menús y la barra de herramientas de la vista, permitiendo así que todos usen el mismo evento para una misma acción, o que todos desactiven y activen a la vez las opciones que en cada momento puedan ser utilizadas.

View.language simplemente almacena los ficheros de properties correspondientes a los idiomas en que se encuentra traducida la aplicación (en principio, solamente español e inglés). Cada componente de la vista accede al fichero de properties activo en cada momento para mostrar el texto correspondiente.

XML

Este paquete se encarga de generar el archivo XML que contiene la configuración Synth creada por el usuario.

Ésto se realiza a partir de las opciones y decisiones tomadas por el usuario en la interfaz. Ha sido implementado de tal manera que funcione como otra vista. Al acceder al modelo y presentar los datos (aunque en vez de por pantalla, en forma de fichero), hemos considerado que presentaba todas las cualidades para ser tratado como otra vista alternativa de los datos. El modelo es el mismo y el controlador se encarga de entregarles los datos. Mientras que la vista presenta los datos en un árbol, el paquete XML los presenta en un fichero de XML, necesario para poder cargar nuestra apariencia personalizada mediante el API de Synth.

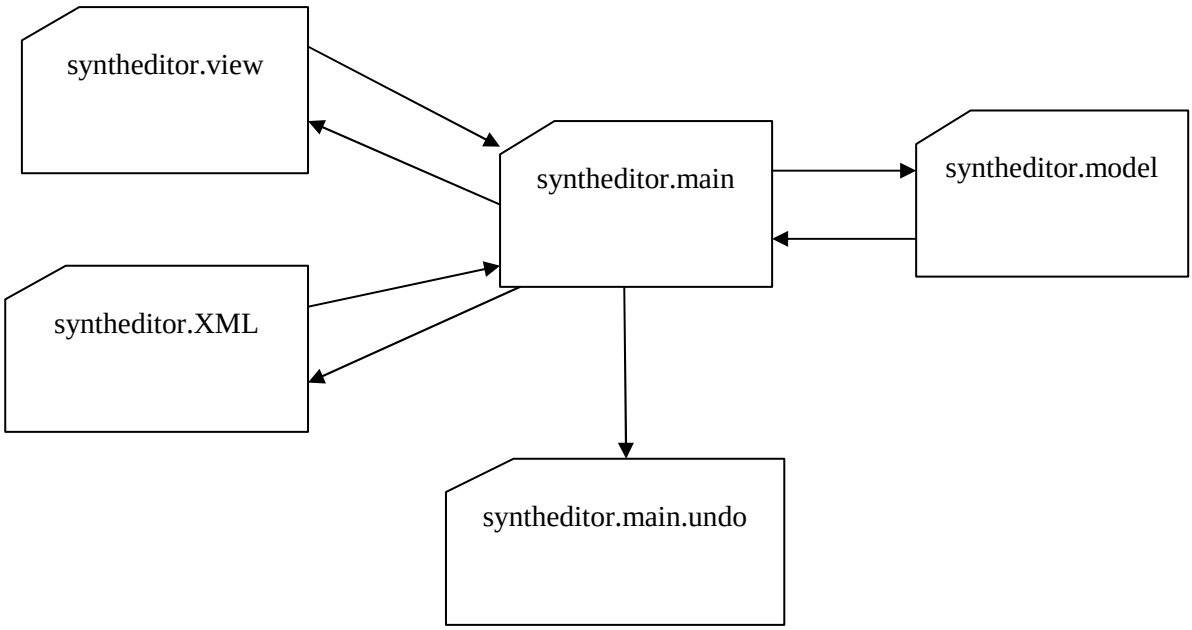
Este paquete, a su vez, se encarga de tomar los datos del fichero de texto y cargarlos en el modelo cuando el usuario emplea la operación “Abrir”.

CONTROLADOR

El controlador es el punto intermedio entre la vista y el modelo. Es el enlace entre ambas partes, por lo que los datos que la vista requiere del modelo deben ser pedidos al controlador, que es quien se encargará de comunicarse con el modelo.

Pero en nuestra aplicación también realiza otra función de vital importancia. Es el encargado de permitir las operaciones de deshacer y rehacer. El paquete *main.undo* básicamente mantiene la pila de operaciones que son susceptibles de ser deshechas. Cada modificación que es enviada desde la vista hacia el modelo es almacenada por el controlador en esta pila. La vista sólo tiene que llamar al método undo() cada vez que el usuario necesita deshacer una operación.

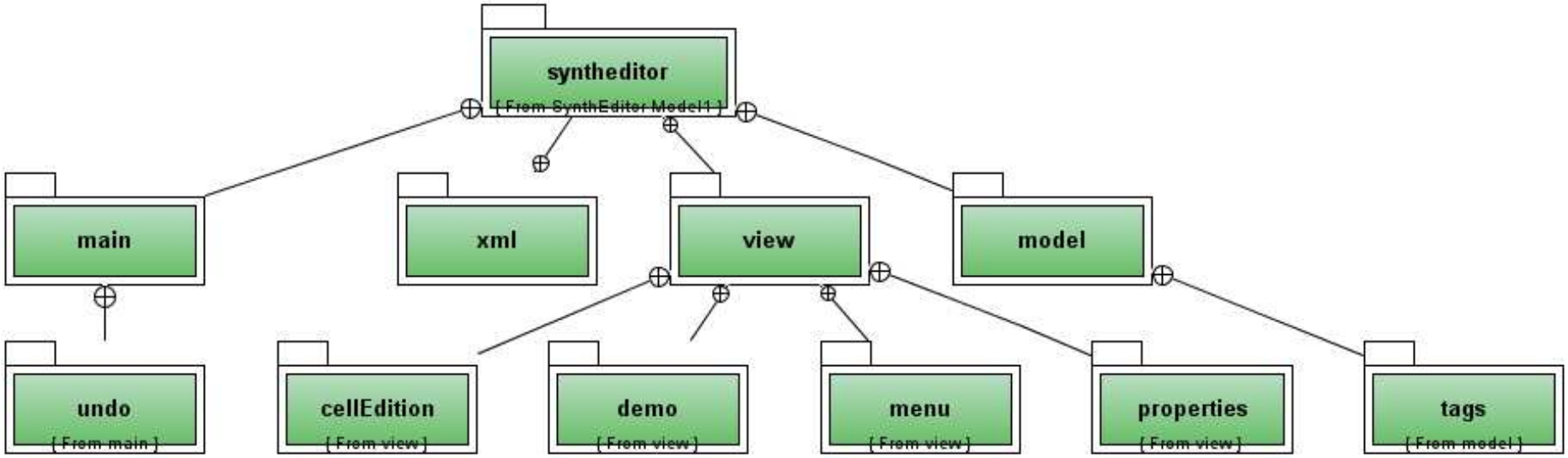
El siguiente diagrama de paquetes da una idea general de la estructura de paquetes que se ha empleado en nuestra aplicación:



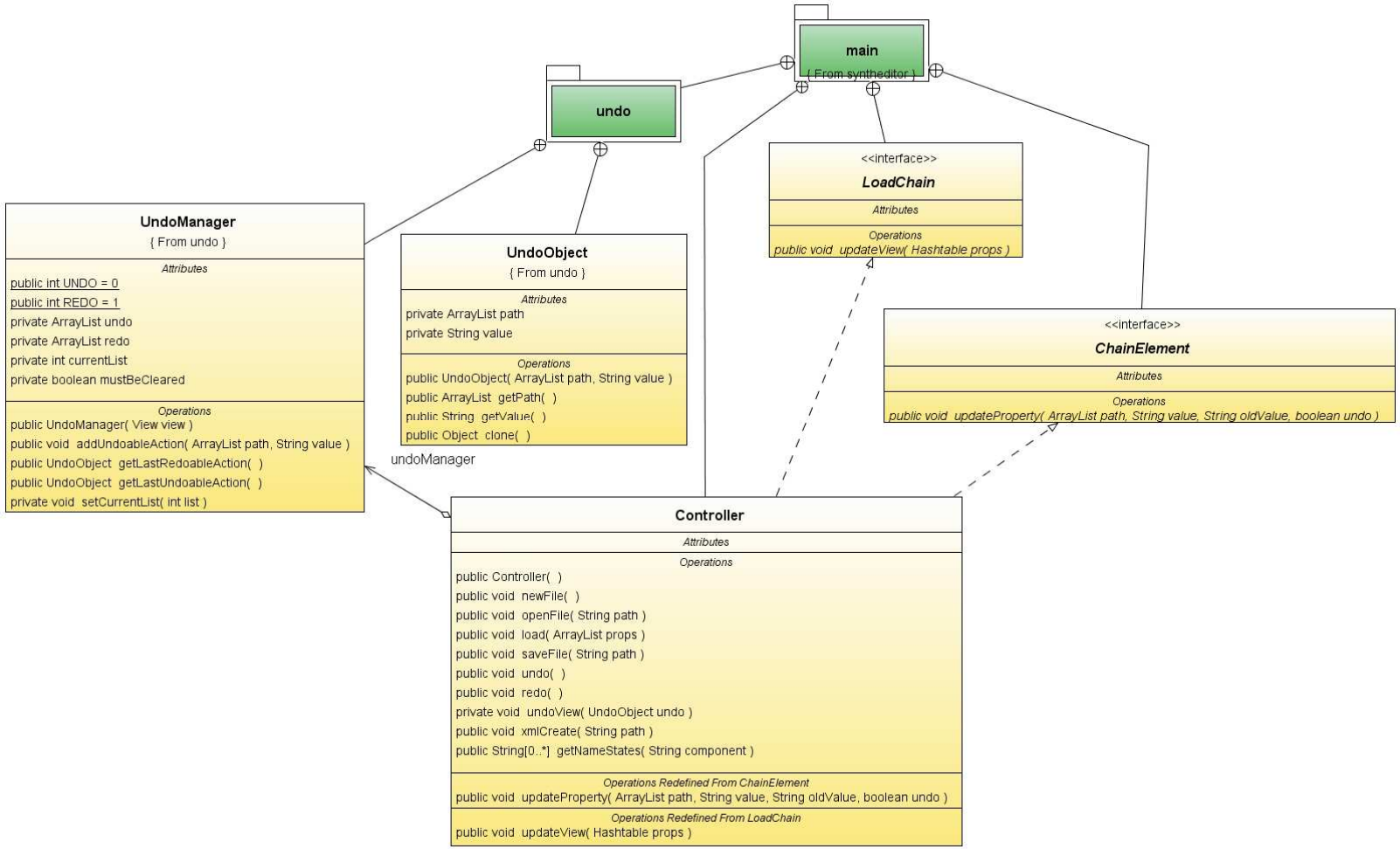
4.3. Diagramas UML

Esquema general:

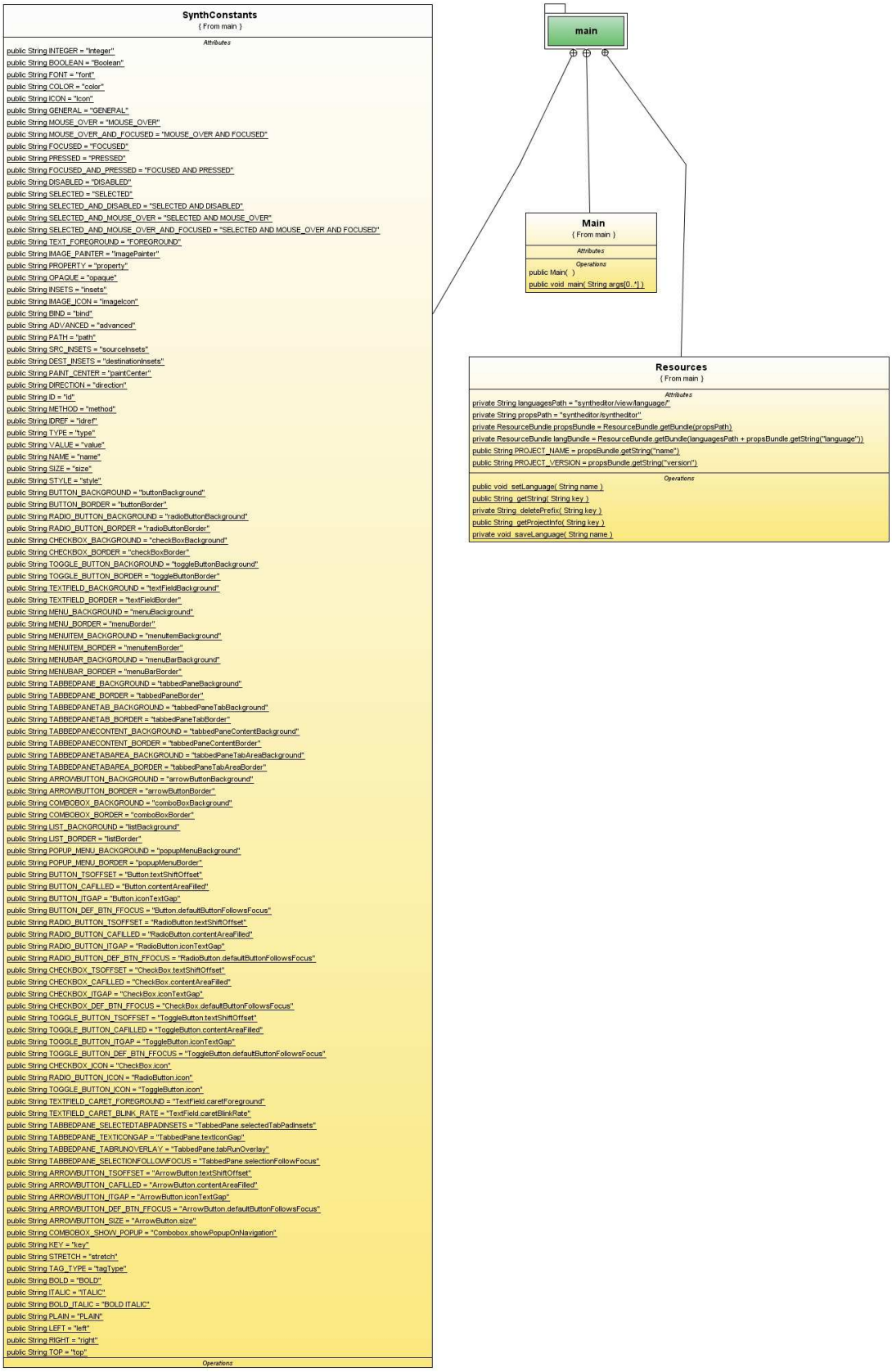
Incluye una vista de los paquetes que conforman la estructura del diseño de nuestra aplicación, qué paquetes son incluidos dentro de otros.



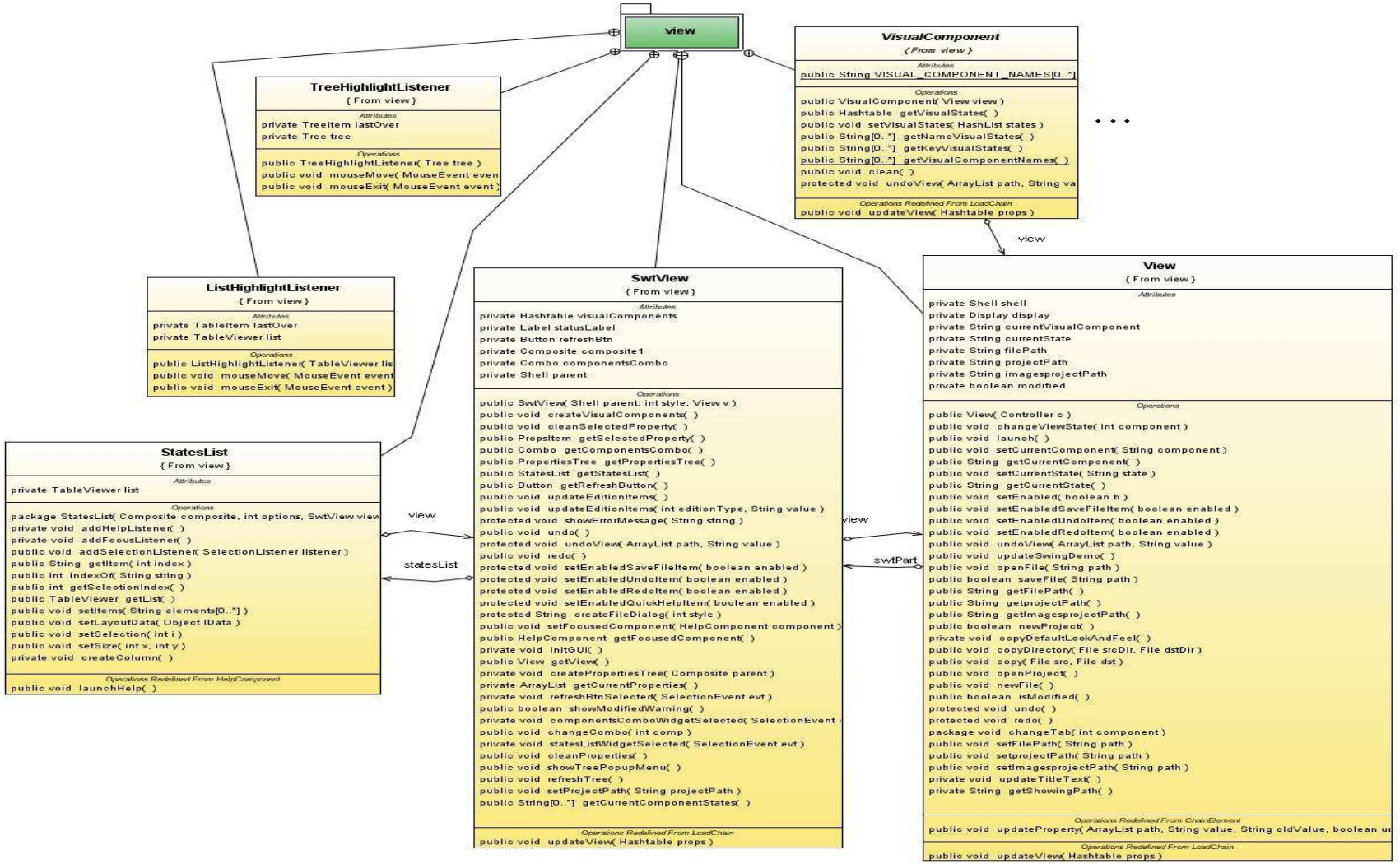
Main (Parte 1): En este diseño se pueden ver las clases pertenecientes a la funcionalidad de deshacer, así como las interfaces del patrón “Cadena de responsabilidad”, que permiten que la información que se carga o se guarda se extienda desde el modelo hasta la vista y viceversa. También se muestra el controlador



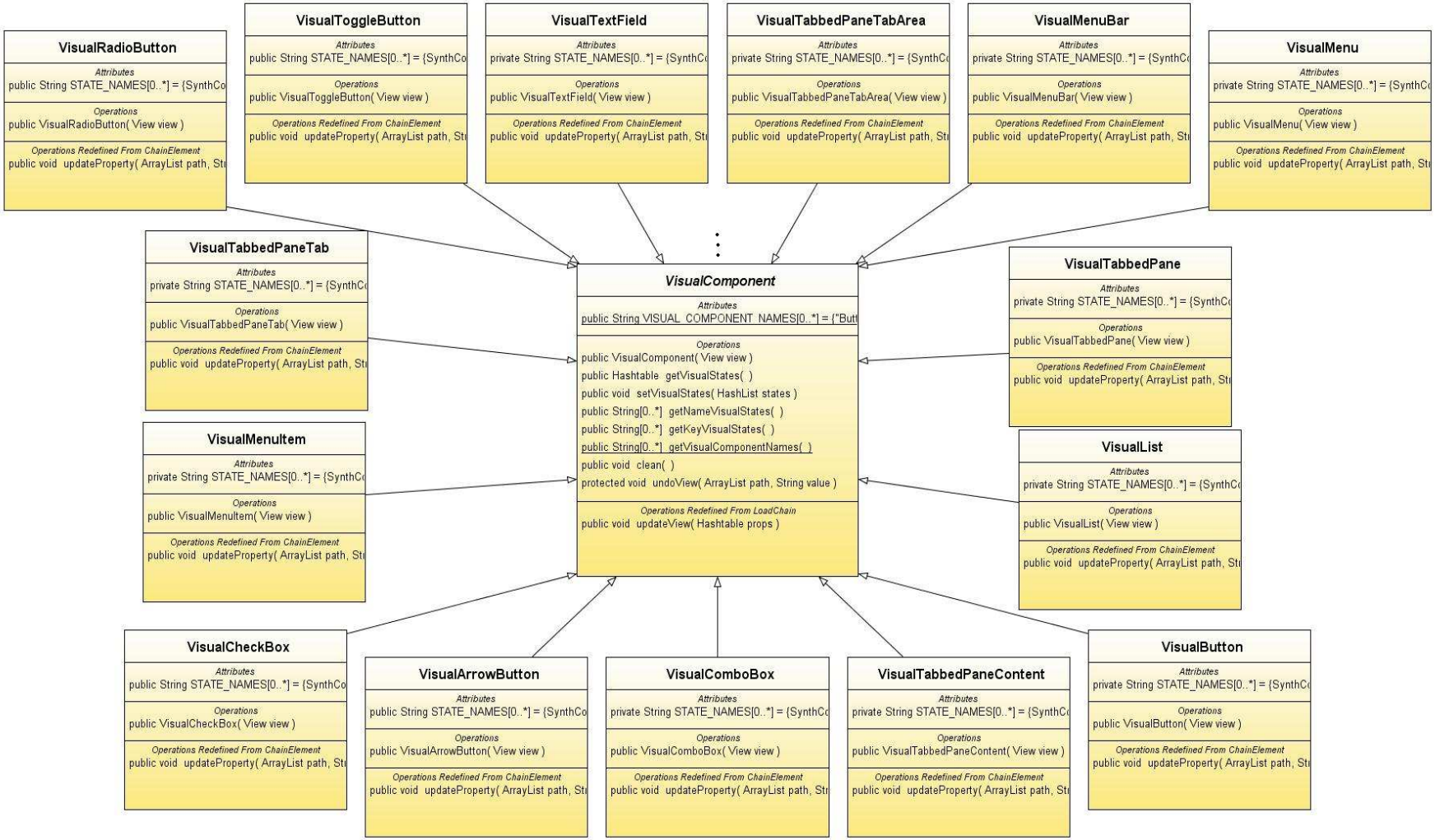
Main (Parte 2): Todas las constantes de la aplicación y la clase Resources, que permite las traducciones y proporciona información del proyecto.



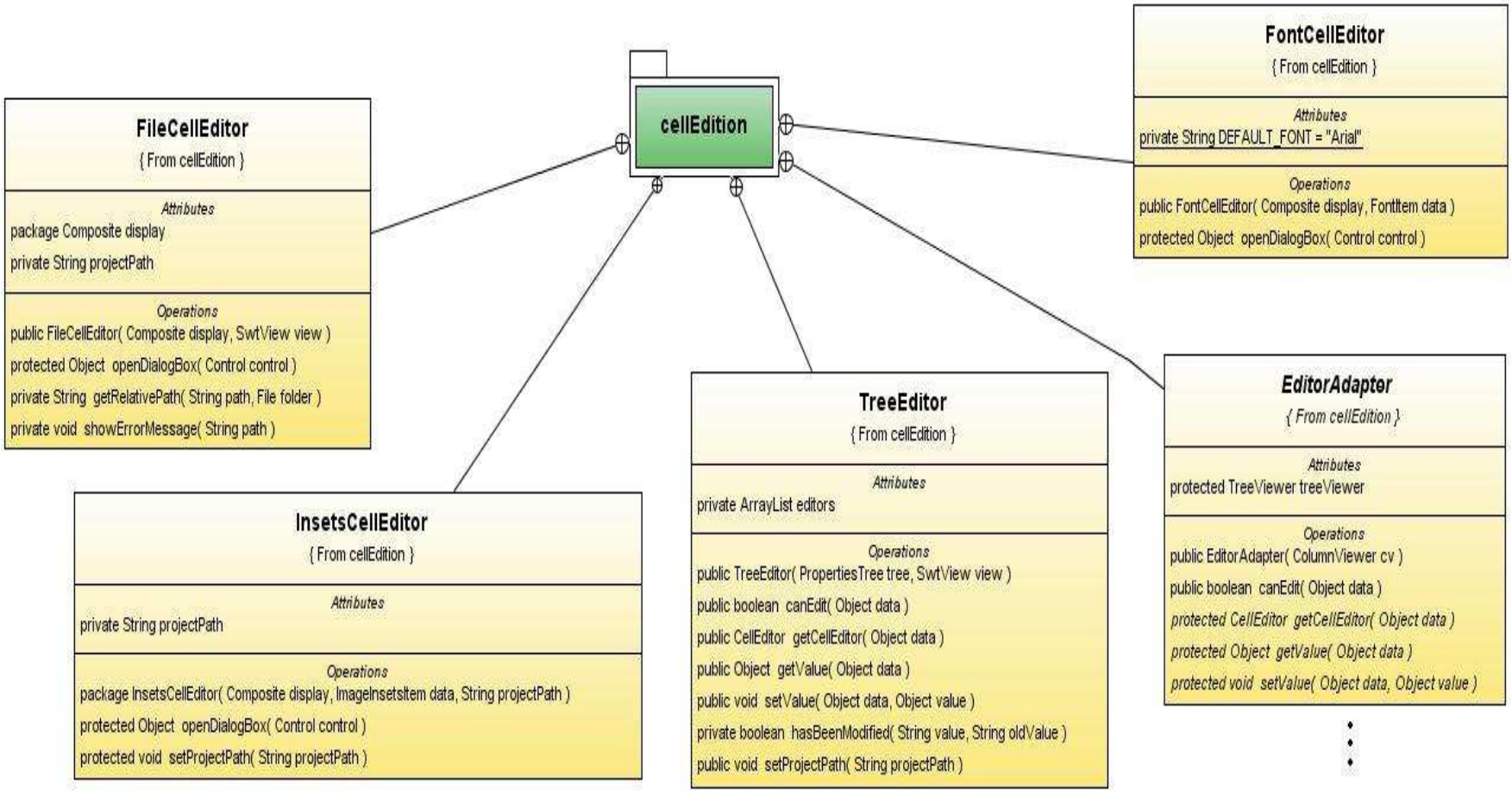
View (Parte 1): Clases de la vista (View) y ventana principal (SWTView). También comprende la lista de estados y algunos oyentes.



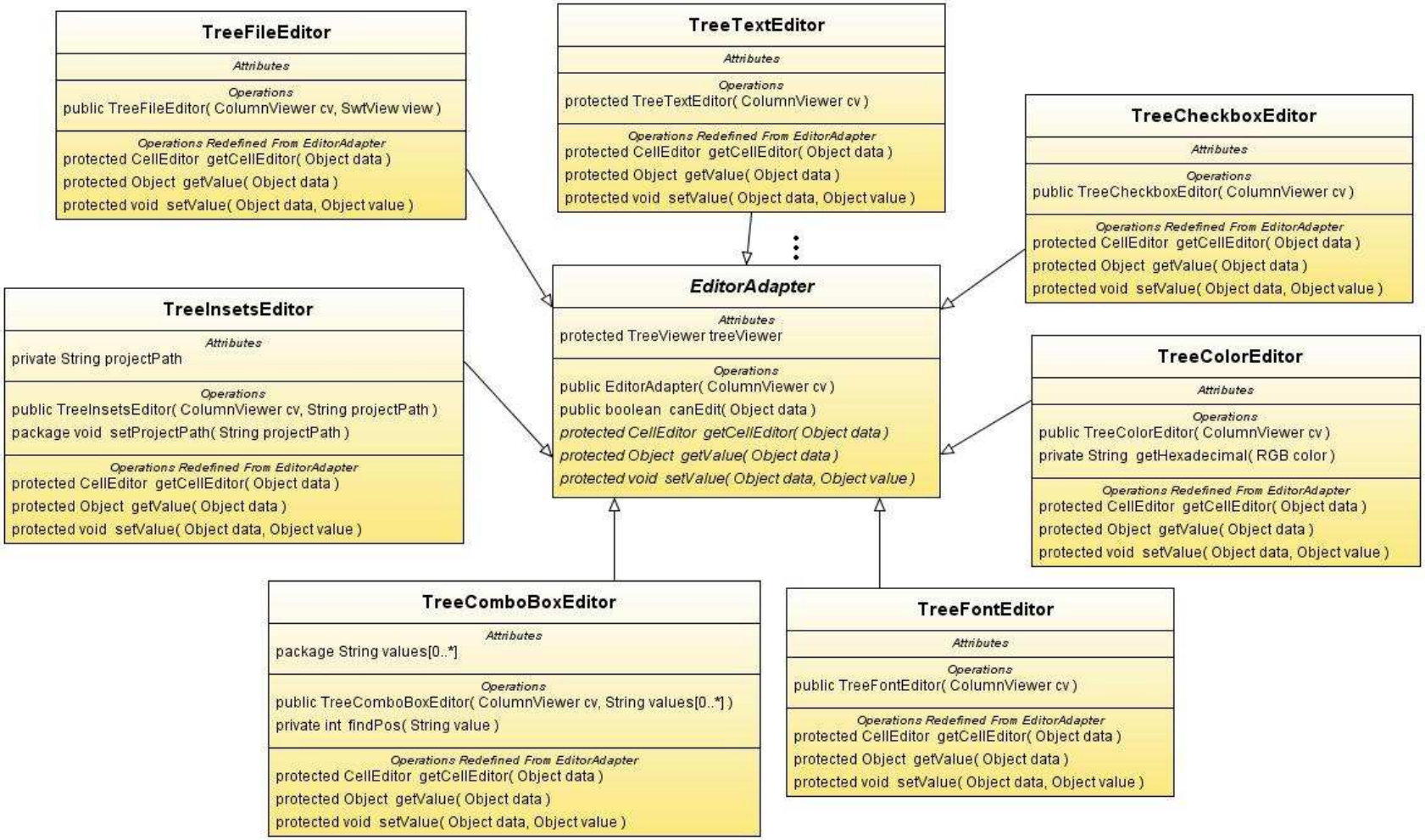
View (Parte2): Estructura jerárquica de los componentes que pueden ser modificados en la vista.



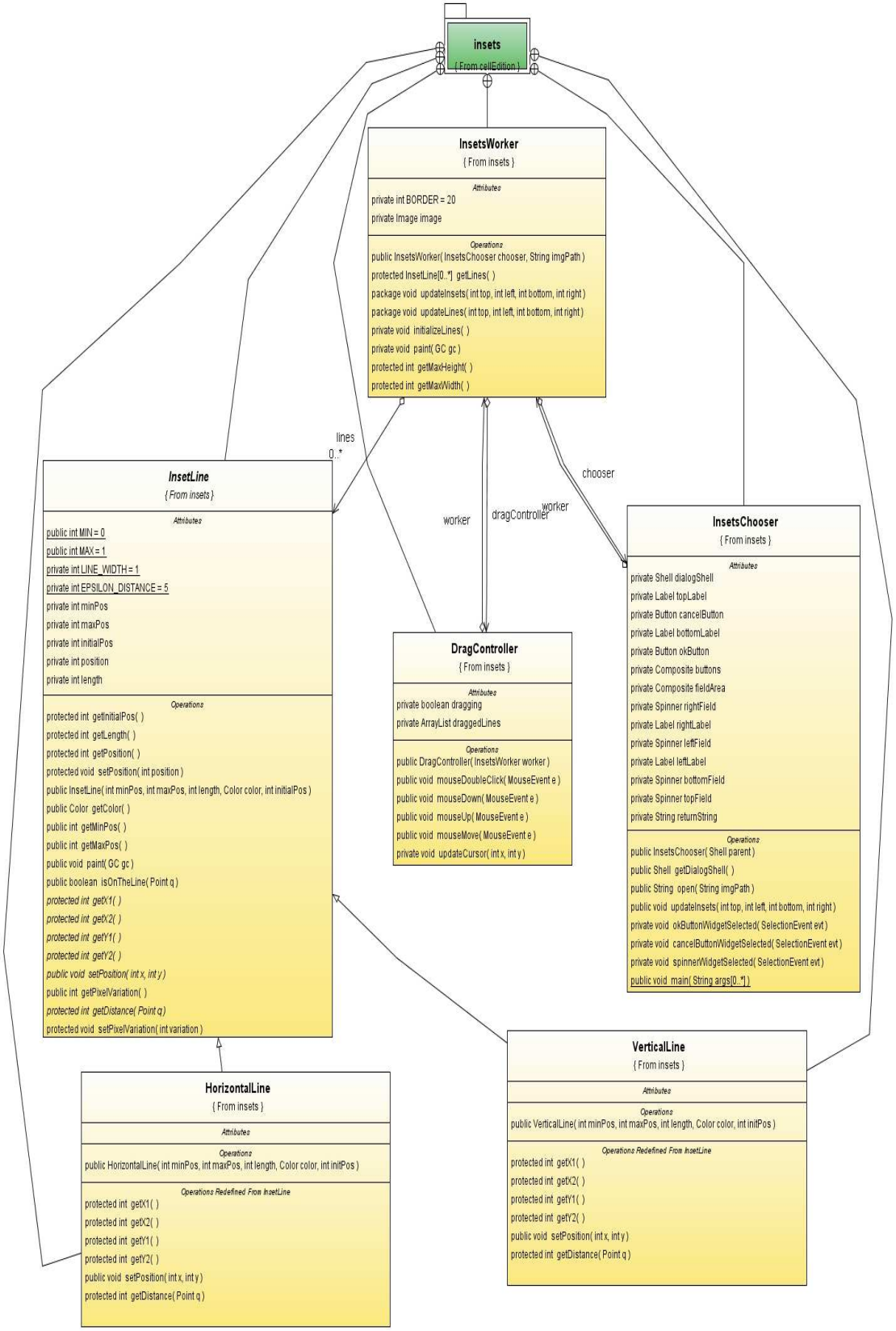
View-CellEdition (Parte 1): Muestra los distintos editores de celdas del árbol de componentes en la interfaz gráfica.



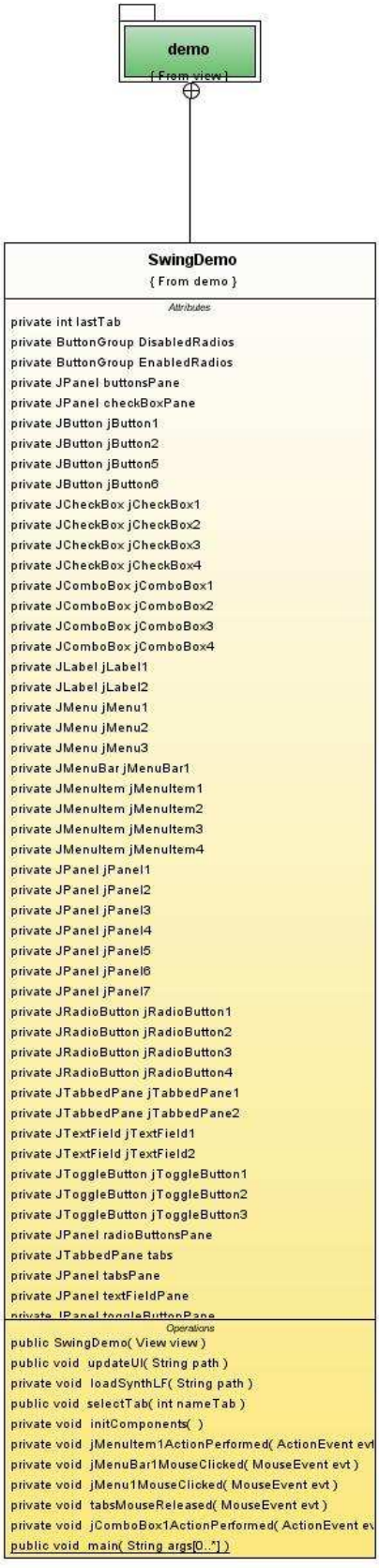
View-CellEdition (Parte 2): Resto de editores de celda.



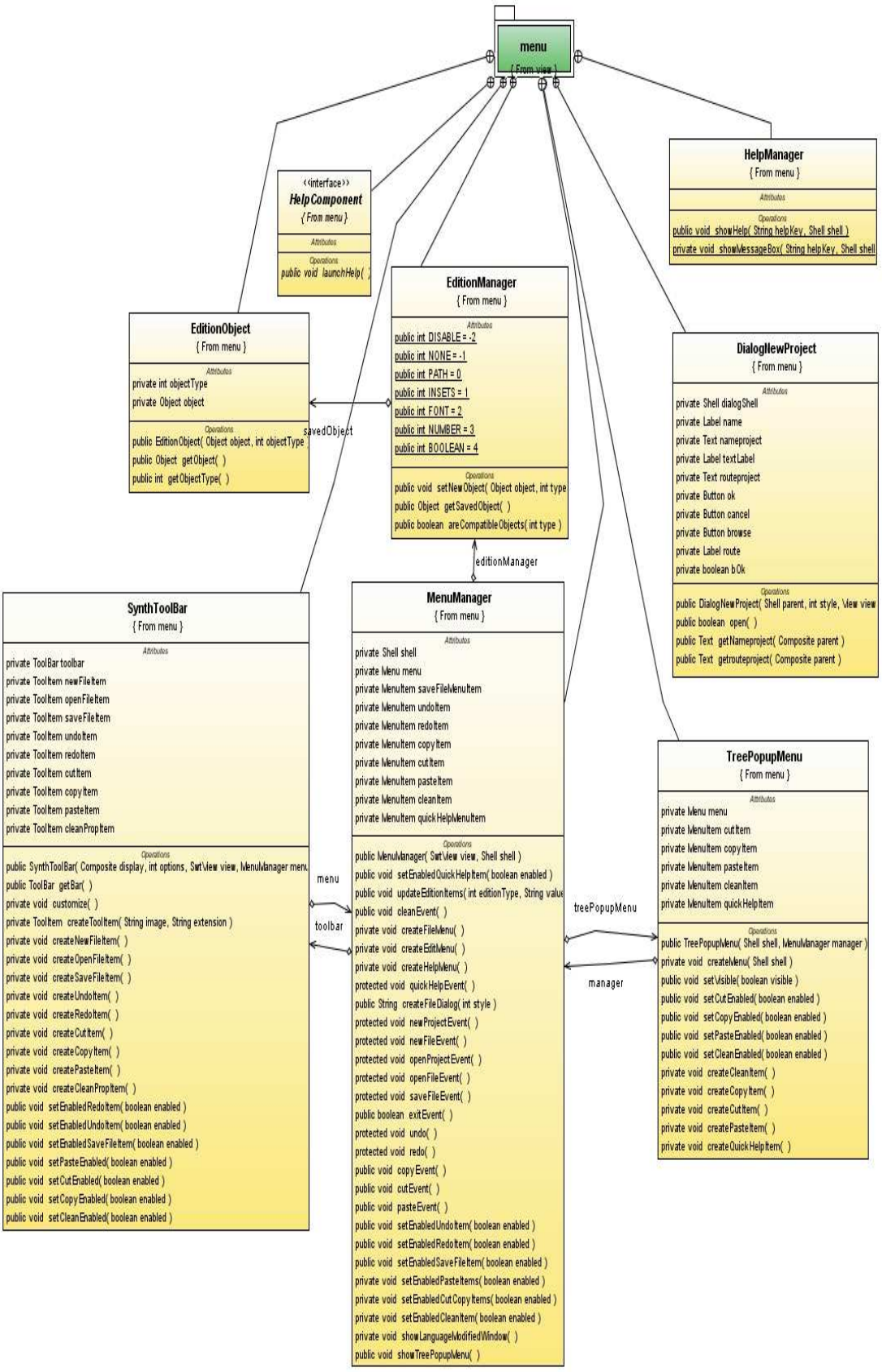
View-CellEdition-Insets: Funcionalidad para la ventana de edición de márgenes.



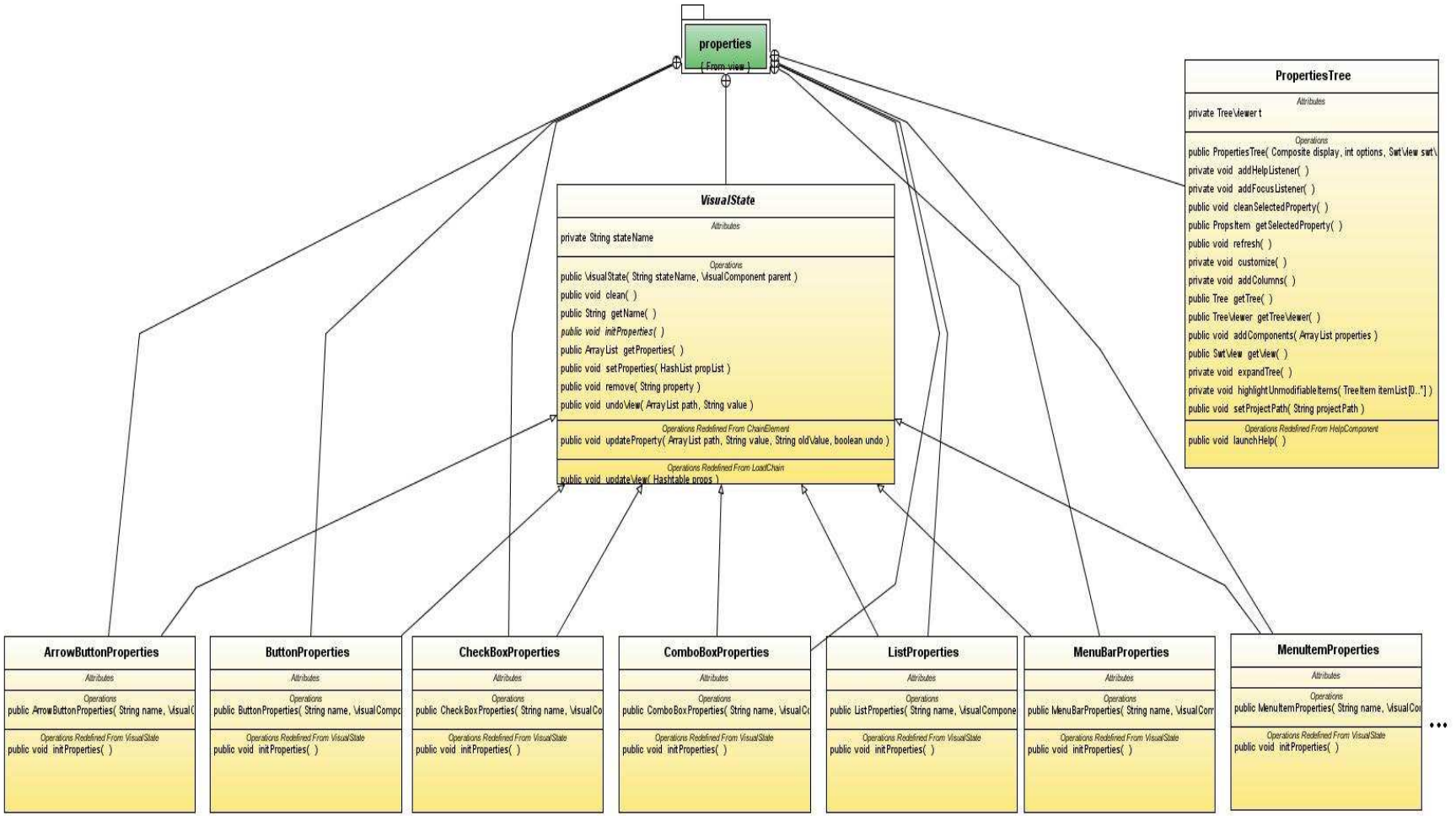
View-Demo: Ventana de demostración, en la que se muestran los cambios de apariencia.



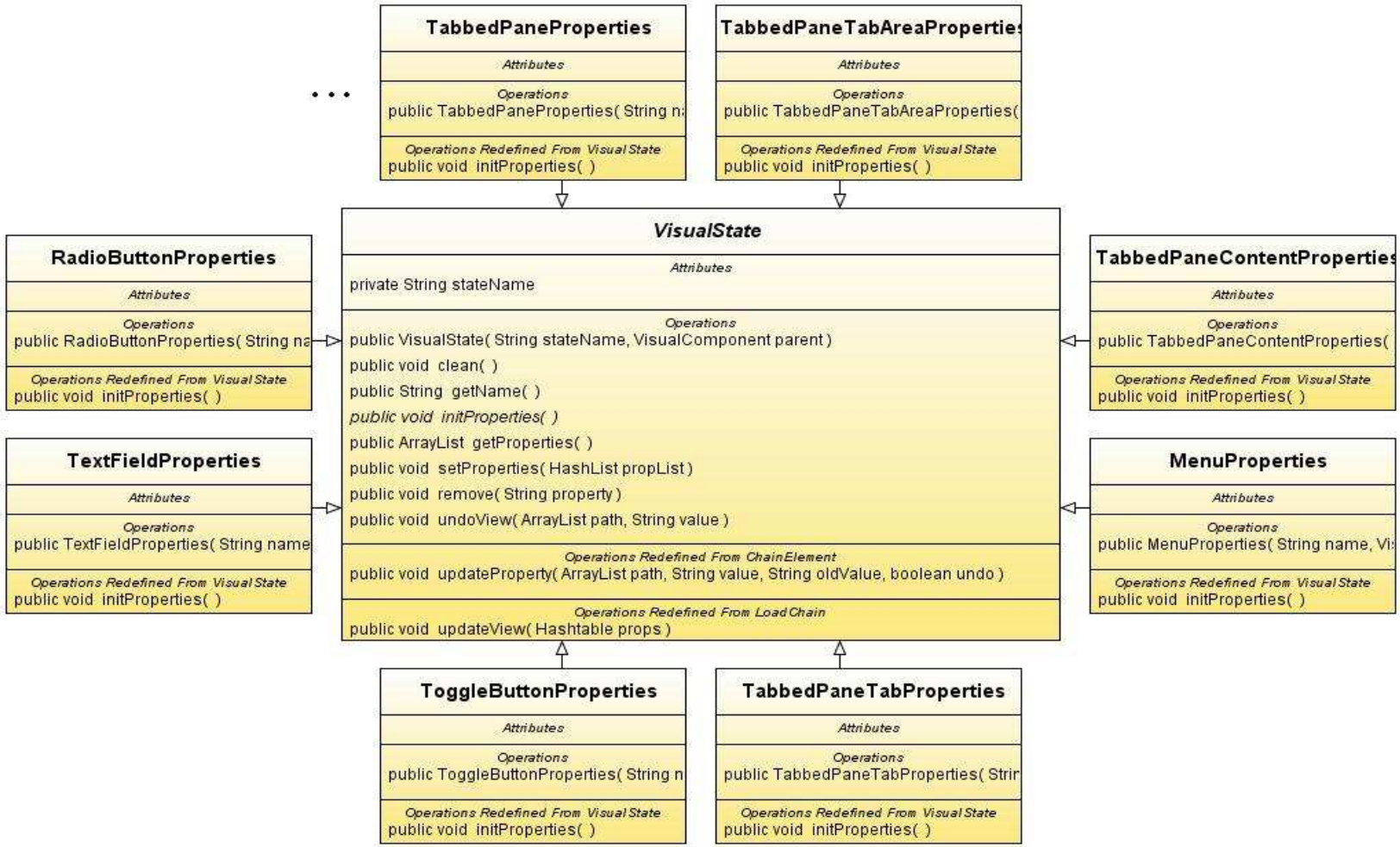
View-Menu: Clases referentes a los menús y la barra de herramientas.



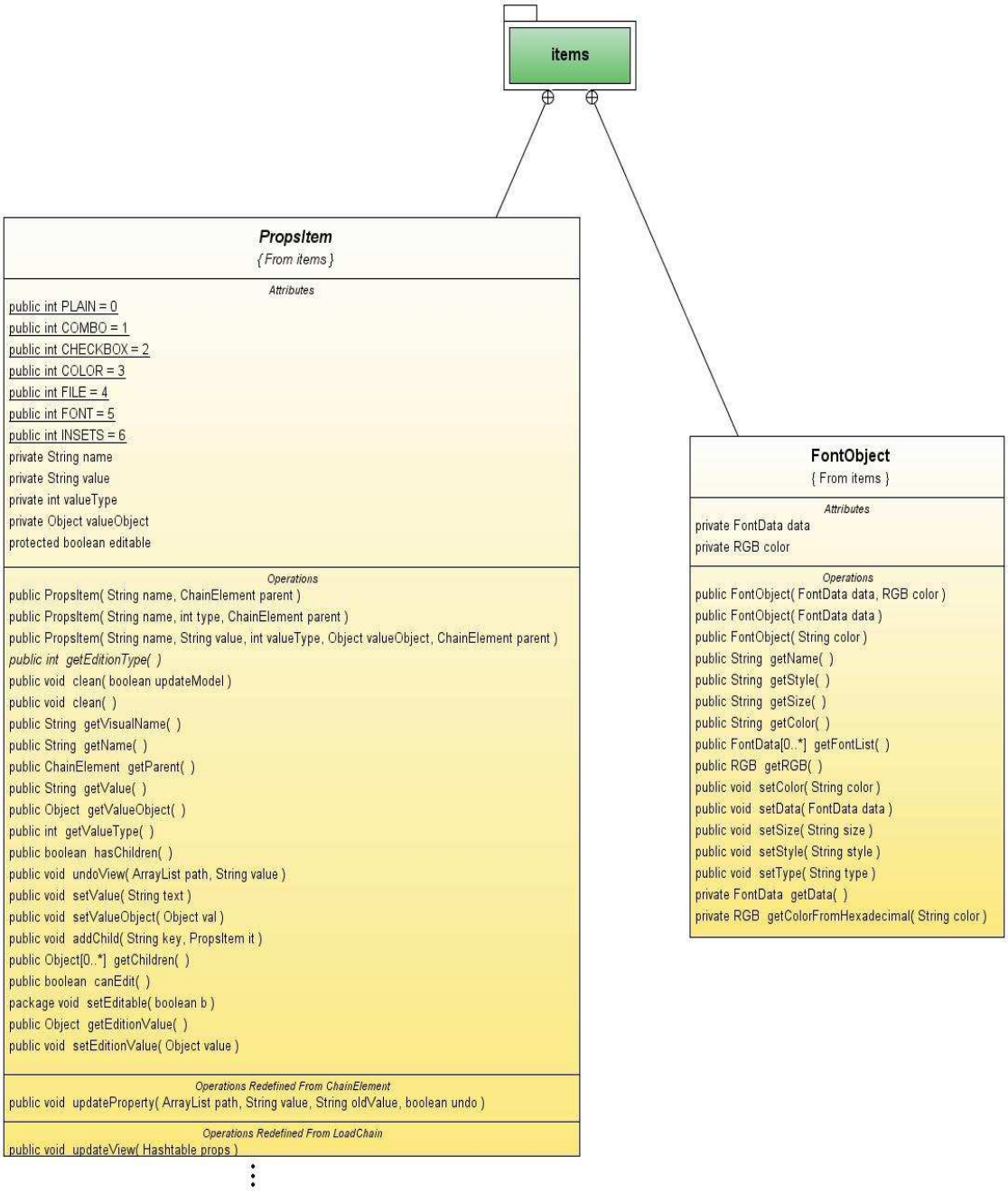
View-Properties (Parte 1) : Clases que almacenan las propiedades dependiendo del tipo de componente que se trate.



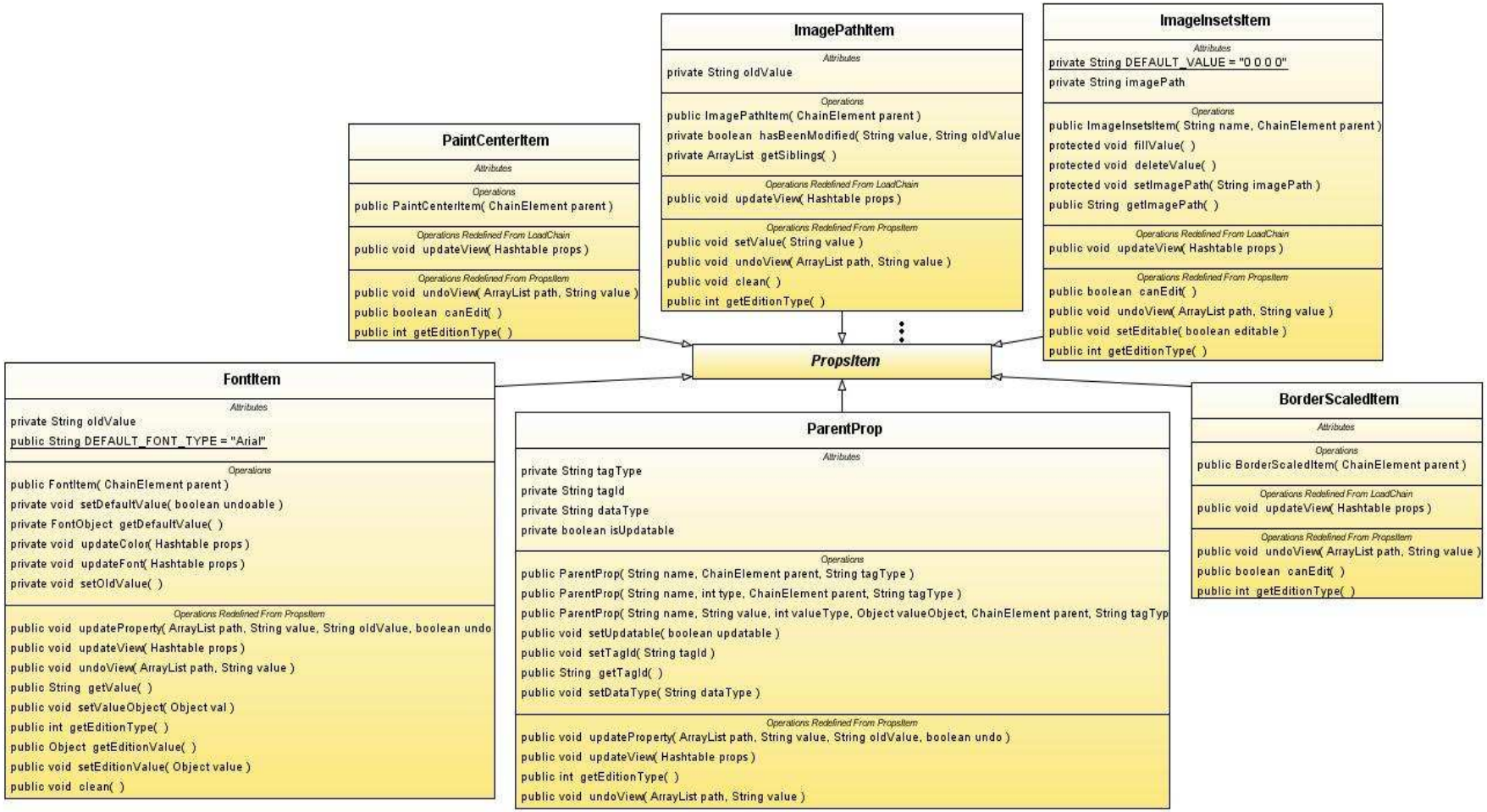
View-Properties (Parte 2): resto de propiedades para los respectivos componentes.



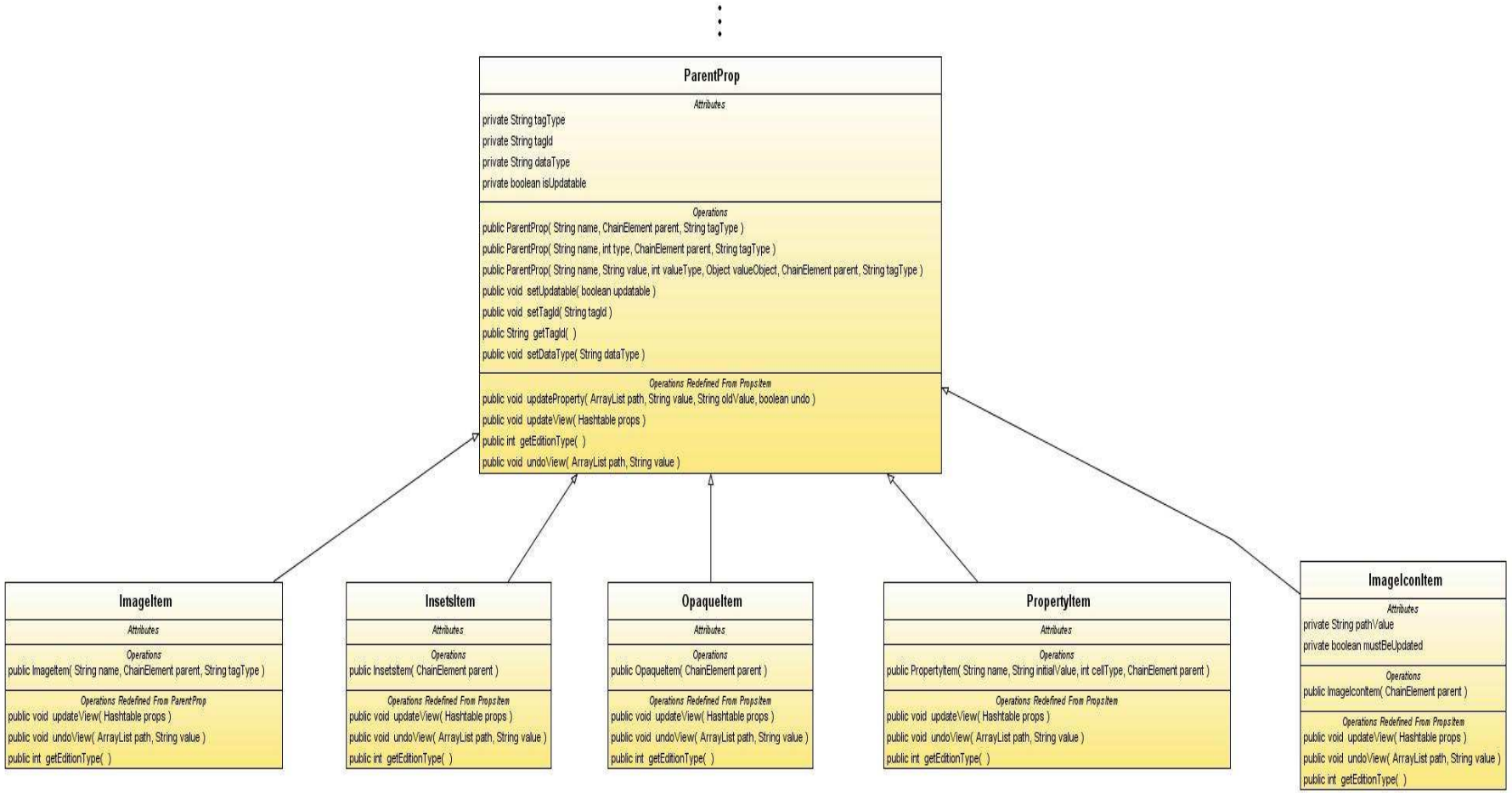
View-Properties-Items (Parte 1): Clase de la que heredan todos los tipos posibles de propiedades de un componente (PropsItem) y la estructura auxiliar para almacenar información sobre fuentes.



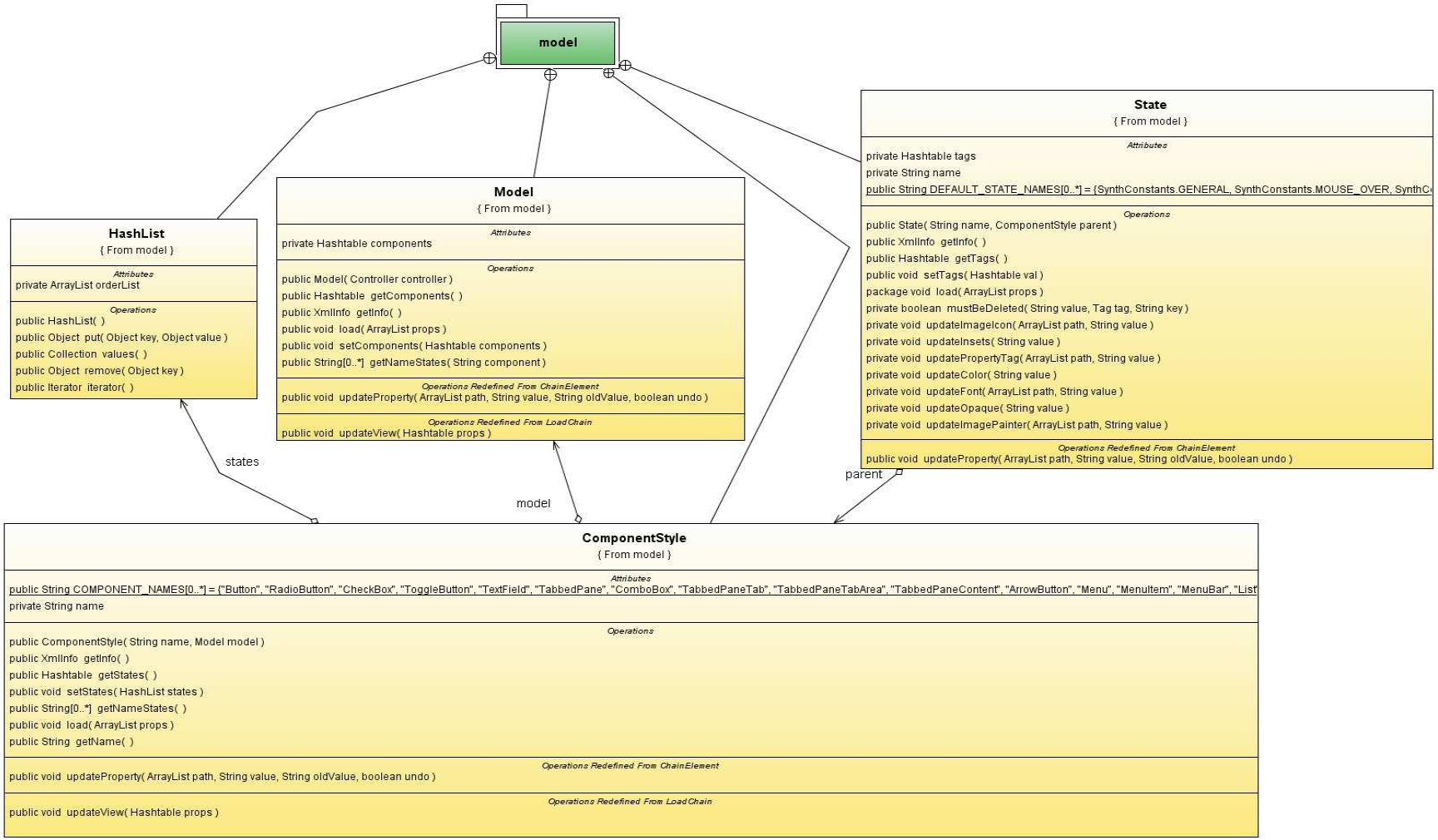
View-Properties-Items (Parte 2): Se muestran los distintos tipos de propiedades, que heredan todos de PropsItem.



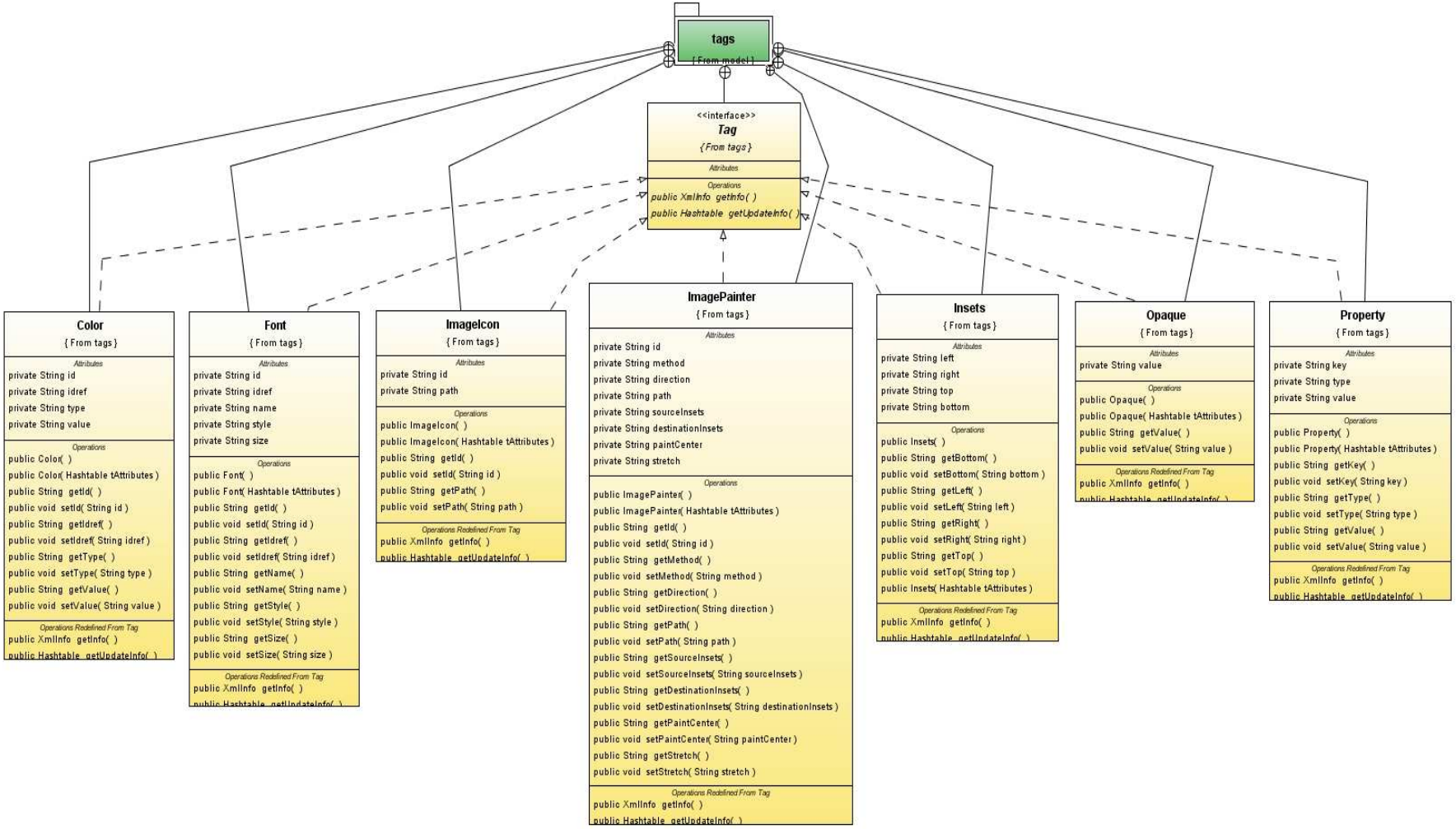
View-Properties-Items (Parte 2): El resto de tipos de propiedades, que también heredan de PropsItem



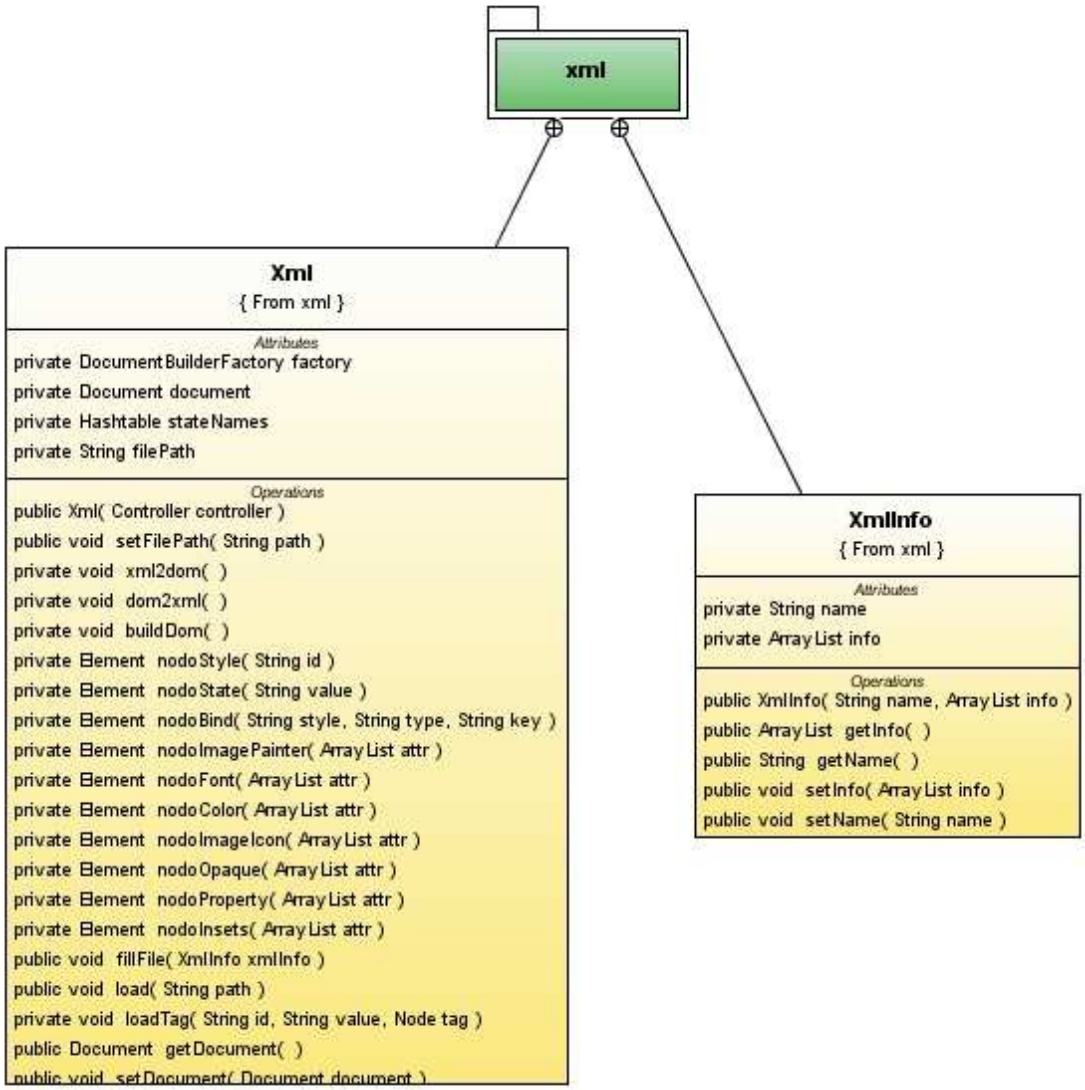
Model: Se corresponde con el modelo. Contiene clases para los estados, estilos y una estructura de datos consistente en una tabla hash ordenada.



Model-Tags: Clases que se corresponden con las etiquetas que pueden aparecer en el XML de Synth.



XML: Paquete que se encarga de la creación, guardado y carga en el modelo de los XML de Synth.



5. Implementación

5.1. Gestión de configuración

El primer paso para conseguir una buena gestión de configuración es establecer el cometido de cada miembro, asignando metas y prioridades de trabajo. Con este objetivo se establecieron revisiones semanales, a modo de reunión de los miembros del grupo con los directores de proyecto.

En estas reuniones se exponían los avances realizados durante la semana, y se fijaban objetivos para la semana siguiente, resultando de gran ayuda en las primeras fases, que requerían fijar ideas y solucionar de forma conjunta dudas las dudas surgidas, tanto de análisis como de diseño o implementación.

Al final de cada reunión, se elaboraba un acta fijando por escrito los contenidos surgidos, estas actas se incluyen para su consulta en el apéndice B. En las fases finales del proyecto, convenimos en cambiar las reuniones semanales por reuniones quincenales, debido al decremento de problemas y dudas, y a que las responsabilidades de trabajo de cada miembro estaban claras.

Esta pequeña explicación del método de trabajo que se llevó a cabo, pone en evidencia la necesidad de hacer uso de herramientas de gestión de configuración, pues el trabajo en equipo precisa llevar un control y registro de los cambios, con el fin de reducir errores, aumentar la calidad y la productividad, y evitar los problemas que puede acarrear una incorrecta sincronización de dichos cambios, al afectar a otros elementos del sistema o a las tareas realizadas por otros miembros del proyecto.

Con el objetivo de poder trabajar simultáneamente y en cualquier lugar sin tener que preguntarnos quién tiene la última versión del proyecto en cada momento, decidimos usar herramientas online de compartición de código fuente y documentación. En concreto elegimos un servidor CVS (*Concurrent Versions System*) para el código, y un *Grupo Google* para la documentación.

Como soporte a esas dos herramientas, y para comunicar avisos urgentes o cambios importantes en la versión principal, también hemos hecho uso del correo electrónico, apoyándonos en un etiquetado concreto de los mensajes para permitir la organización y asignar prioridad.

Control de versiones para el código fuente:

El repositorio de código pertenece a SourceForge[7], que presta servicios gratuitos a aquellos proyectos de código abierto que considera interesantes. Los datos de nuestro proyecto son:

Nombre: Synth Editor

Nombre UNIX: syntheditor

Web en Sourceforge: http://sourceforge.net/projects/syntheditor

Para trabajar con el código fuente y CVS hemos utilizado las opciones integradas en el entorno de desarrollo de NetBeans.

El uso del CVS resultó de gran utilidad, permitiendo trabajar de forma autónoma sobre la última versión del proyecto y sin suponer más que algún pequeño problema puntual al subir archivos, debido a problemas por la conexión a internet, o con los servidores de SourceForge.

El proyecto consta de distintas ramas que corresponden a distintas fases del proyecto, cuando hemos realizado algún cambio importante en el diseño hemos optado por crear una nueva para poder regresar a la versión anterior de forma segura si surgía algún problema. Durante la planificación inicial nos planteamos la posibilidad de crear una rama para cada miembro del grupo y otra rama principal en la que siempre tendríamos una versión estable del mismo, pero al final decidimos trabajar todos sobre la versión estable, debido a que en experiencias anteriores habíamos conseguido trabajar bien, evitando así conflictos en las fases de unión de los trabajos individuales.

[7] www.sourceforge.net

Control de versiones para la documentación:

El Grupo Google utilizado puede ser consultado en la siguiente dirección:

http://groups.google.com/group/proyectosynth

La herramienta ha resultado de gran ayuda por permitir la compartición inmediata de cada documento generado. En cuanto al control de versiones, se realizó mediante un protocolo para el nombre de los archivos, añadiendo al final de los mismos el número de versión. Este protocolo destacó en utilidad durante la documentación de las baterías de pruebas, que requerían revisión constante, indicando las pruebas fallidas en cada componente concreto, junto con la descripción del problema, solución y fecha en la que se solucionó el fallo.

5.2. Pruebas

Tan importante como el diseño y la implementación es saber que lo que estamos haciendo está bien y funciona correctamente, para ello hemos realizado pruebas a diferentes niveles:

- Pruebas de integración: se trata de probar la compatibilidad y funcionalidad de los interfaces entre los distintos módulos que componen nuestra aplicación. Al añadir nuevos componentes a la aplicación se realizan este tipo de pruebas que consisten en comprobar que para el nuevo componente, todos los posibles estados y atributos son correctos y que además el resto de la aplicación sigue funcionando con normalidad. Para ello, nos fijamos en que el XML generado es correcto, es decir, está bien formado y los atributos que hemos marcado se corresponden a los que nos muestra. También comprobamos en la ventana de demo que el resultado es el que esperábamos. Para hacer más sistemáticas estas pruebas, contamos con unas plantillas creadas previamente para cada componente donde anotamos si pasa la prueba o no y en caso negativo cual es el problema, su solución y el día que se solucionó. El resultado de estas pruebas lo podemos ver en apéndices.

- Pruebas de sistema: son de distintos tipos
  - Robustez: capacidad del programa para soportar entradas incorrectas. Después de las pruebas de integración, probamos a poner valores incorrectos en.
  - Usabilidad: calidad de la experiencia de un usuario en la forma en la que este interactúa con el sistema, facilidad de uso y grado de satisfacción. Estas pruebas se han ido realizando a medida que se introducían los distintos elementos de usabilidad, al introducir el deshacer/rehacer probamos que este funcionaba correctamente y según introducíamos una nueva funcionalidad lo volvíamos a poner a prueba.
  - Aceptación: Las que hará el cliente, si el sistema cumple con lo deseado. Este tipo se ponía a prueba en las distintas reuniones que hemos ido teniendo a lo largo del proyecto, nuestros directores de proyecto probaban las nuevas funcionalidades que iba teniendo nuestro proyecto.

5.3. Problemas surgidos

A continuación describimos los problemas más importantes con los que nos hemos encontrado a la hora de realizar nuestro proyecto y la solución que tomamos.

- En un principio pensamos en desarrollar una aplicación como plug-in para el entorno de desarrollo Eclipse pero tras realizar algunos plug-ins para manejarnos con las herramientas de creación de estos, nos dimos cuenta de que nuestra aplicación no tenía porque depender de este entorno de desarrollo ya que para modificar la apariencia no es necesario que el usuario se encuentre programando.

  La idea de crear un plug-in nació básicamente porque nos permitiría modificar programáticamente el código de la aplicación que se estuviese creando para que se mostrara directamente el Synth Look And Feel sin necesidad de que el programador se tuviera que preocupar de añadir líneas de código.

  El problema de esta idea es que la esencia en sí misma de un look and feel contradice esta aplicación. Un mismo look and feel normalmente será utilizado en infinidad de aplicaciones distintas, y una aplicación creada para un fin en particular nunca contendrá todos los componentes que pueden ser utilizados en una interfaz gráfica, por lo que cuando se quiera usar dicho look and feel para otra aplicación, habrá componentes que no estén definidos, y habrá que volver a retocarlo.

  Otra razón por la que el plug-in no encajaba con nuestra aplicación es que este tipo de tareas la suelen realizar diseñadores gráficos, y no programadores. El tiempo que le supone a un diseñador gráfico aprender a manejarse en Eclipse y posteriormente utilizar un plug-in puede ser infinitamente superior al que tardaría en aprender a utilizar una aplicación independiente, que puede ser diseñada además teniendo en cuenta que el usuario normalmente no será un programador.

  Solución: decidimos enfocar el proyecto a una aplicación independiente del entorno de programación. Esto nos aporta las ventajas anteriormente comentadas: el usuario de nuestra programación no necesita un tiempo de adaptación a Eclipse, no necesita ser programador para poder crear la interfaz gráfica personalizada y además permite crear interfaces gráficas independientes del software para el que estén siendo desarrollados, por lo que son completamente reutilizables.

  Además, el haber hecho el software independiente de Eclipse nos ha permitido una libertad mucho mayor a la hora de desarrollar la aplicación, lo que nos ha permitido realizar un producto mucho más manejable y fácil de usar.

- Al empezar a realizar la interfaz de ejemplo nos dimos cuenta del siguiente problema: como nuestra aplicación está dividida en dos ventanas, una en la que aparecen los distintos componentes, estados y aspectos que podemos modificar de cada uno de estos y otra en la que aparecen estos componentes por defecto y podemos previsualizar su aspecto al realizar algún cambio, teníamos el problema de que si queríamos cambiar el look and feel de la ventana de previsualización también se cambiaba en la ventana de

modificaciones con lo que se pueden dar situaciones un tanto extrañas para el usuario (como por ejemplo aplicar un look and feel vacío) .

El problema básicamente era que si desarrollábamos toda la aplicación en Swing, al modificar el Look and Feel, éste afecta a toda la aplicación, y no hay una manera fácil de indicar a cuál de las ventanas queremos que aplique la nueva apariencia. Si lo aplicábamos a las dos ventanas, en las primeras etapas de desarrollo de del Look and Feel, el usuario se iba a encontrar con una ventana sin ninguna apariencia, que hace muy difícil trabajar con ella.

Surgieron varias posibilidades:

- Una de ellas consistía en crear dos aplicaciones independientes que se comunicaran entre sí. Con esta idea conseguíamos que cada ventana tuviera su propia apariencia, pero la comunicación entre ambas ventanas se hacía mucho más difícil. Había que generar alguna especie de protocolo de comunicación entre ellas, y además el usuario percibiría que realmente había dos aplicaciones distintas ejecutándose, lo que le resultaría muy extraño e iría en contra de la usabilidad.
- La siguiente idea fue utilizar AWT para crear la ventana en la que el usuario especifica las propiedades de la interfaz y Swing en la ventana de demostración, ya que AWT toma su apariencia del sistema operativo sobre el que se ejecuta y por tanto no permite modificar su apariencia. La solución parecía perfecta, pero nos encontramos con que AWT no tiene algunos componentes indispensables para la creación de nuestra interfaz, como eran los árboles.
- Otra opción era adentrarnos en el API de Swing y modificar los métodos de repintado para que sólo se invocase bajo ciertas condiciones. La opción probablemente era la más limpia, pero era un desarrollo que requería un tiempo muy elevado de desarrollo, y si nos decantábamos por esta opción era muy probable que fuera lo único que consiguiéramos hacer durante todo el tiempo de proyecto, suponiendo que era una opción factible, algo de lo que no estábamos 100% seguros.

Solución: Finalmente decidimos utilizar SWT, la opción de desarrollo de interfaces gráficas implementada por IBM, principalmente para la creación de Eclipse. Puesto que ya teníamos conocimientos de su API al haber estado investigando para el desarrollo de plug-ins en Eclipse, no nos suponía ningún esfuerzo extra emplear esta tecnología. Además, SWT toma su apariencia del sistema operativo sobre el que se ejecuta, como AWT, con la diferencia de que SWT suministra todos los componentes de Swing (de hecho, algunos más), incluyendo los árboles, que era lo que no nos aportaba AWT.

La solución funcionó a la perfección, ya que aunque hay algún problema de incompatibilidad cuando hay ejecutándose una ventana de Swing y otra de SWT, el propio SWT proporciona métodos de sincronización entre ambas ventanas para que puedan ser utilizadas sin problemas.

- Existen problemas con las fuentes que son ajenos a nuestra implementación y por los que fallan determinadas propiedades como los insets, si no ponemos una fuente por defecto estos no funcionan. Hay campos que no deben estar vacios.

  Cuando estábamos realizando pruebas, nos dimos cuenta de que al modificar la propiedad de los márgenes sobre un documento Synth vacío se producía una excepción en el momento en que Swing intentaba repintar la ventana una vez se había modificado el Synth Look and Feel. Tras investigarlo, descubrimos que esa excepción desaparecía si personalizábamos la fuente del texto de dicho componente.

  La declaración de Synth no obliga a este hecho, por lo que parece un fallo de la implementación de Synth. Además, esto no aparece documentado en ninguna de las fuentes consultadas. Por lo que indica la excepción, parece ser que Synth no conoce el tamaño que tiene la fuente, por lo que no puede hacer bien los cálculos de tamaño y repintado de los componentes. En teoría debería tomar el tamaño de la fuente por defecto, pero no ocurre así, bien porque no hay ningún tamaño de fuente especificado en el Synth por defecto, bien porque lo está pero no se está tomando correctamente.

  Solución: Se declara una fuente por defecto para el estado GENERAL de todos los componentes. De este modo, Synth ya tiene un tamaño de fuente y puede repintar sin problemas todos los componentes de la interfaz. El problema desaparece por completo.

- Puesto que nuestra aplicación soporta el cambio de idioma (como ya se mencionó en el apartado de usabilidad), esto supone cambiar en tiempo de ejecución uno a uno todos los textos de los componentes de la interfaz, lo cual conlleva un coste y complejidad que consideramos innecesario para algo de lo que se va a hacer uso una vez en la aplicación.

Cambiar el idioma completo de una interfaz supone un elevado esfuerzo de programación. Hay que acceder a todos los componentes de una interfaz, y en nuestro caso, a todas las clases que permiten tener en cada momento las propiedades para componente y estado dado en el árbol de propiedades, e ir modificando dichas cadenas una a una, volviendo a tomar la información del fichero de idioma. Realmente en nuestra aplicación, esta segunda parte estaba solucionada, porque las cadenas se toman cuando se actualizan las propiedades del árbol, y las toma del fichero de idioma activo, que es el que el usuario haya decidido. La primera parte tiene una complejidad asumible, pero que no aporta mucho a la usabilidad. Un usuario normalmente cambiará el idioma de su aplicación una única vez, o incluso ninguna si el instalador permite seleccionar el idioma desde un principio, por lo que añadir complejidad al código que puede perjudicar los tiempos de ejecución de otras funcionalidades mucho más utilizadas no nos parecía una buena idea.

Solución: Se avisa al usuario de que los cambios de idioma no tendrán efecto hasta que no se reinicie la interfaz con una caja de texto. La aplicación entonces sigue funcionando normalmente en el idioma anterior, pero ya se ha almacenado en las propiedades de la aplicación el nuevo idioma, que será cargado la próxima vez que el usuario la ejecute.

Tomamos esta decisión ya que básicamente es la que toman prácticamente todas las aplicaciones comerciales que permiten el cambio de idioma. De hecho, muchas de ellas ni siquiera permiten cambios de idioma. Éste se decide durante el proceso de instalación y se mantiene así mientras no se reinstale el software.

- Al comenzar a implementar la aplicación utilizamos una ventana de previsualización predefinida de Swing, pero no nos era de mucha utilidad puesto que no mostraba todas las opciones y estados que podemos modificar.

  Esta ventana que utilizamos al principio es una demostración que se instala con el JDK de Java como ejemplo de aplicación Swing. Al principio creímos que nos sería de utilidad, porque incluye todos los componentes de Swing separados por pestañas, permitiendo ver cómo se mostraría esta ventana en el caso de que se cargara el Synth creado por nuestra aplicación.

  Esta ventana presentaba varios problemas. Uno de ellos era que estaba formado por varios paquetes con muchas clases difíciles de comprender y de personalizar en el caso de que fuera necesario. Además estaban muy mal comentadas y bastante desorganizadas, por lo que cuando algo no funcionaba era muy difícil encontrar el motivo.

  Como esta ventana incluía tanta funcionalidad, también tenía el problema de que consumía demasiada memoria y ralentizaba mucho tanto el uso de la aplicación como la carga del Synth Look and Feel para comprobar los cambios.

  Además, como era una interfaz que no estaba específicamente desarrollada para mostrar la apariencia de los componentes en todos sus posibles estados, había algunos de ellos (sobre todo estados desactivados) que no podían ser probados, y por tanto se desconocía

si su apariencia sería la correcta en caso de que se decidiese añadir un componente desactivado a la aplicación que quisiéramos crear empleando nuestro Look and Feel.

Solución: Implementamos nuestra propia ventana de previsualización con todas las opciones y estados que queremos mostrar al usuario. De esta forma podemos mostrar los estados desactivados de los componentes, podemos añadir todos los elementos nuevos que queramos sin problema alguno, y sobre todo nos deshacíamos de toda la funcionalidad que no venía al caso en la anterior ventana, liberando una importante carga de memoria que hace que nuestra aplicación consuma muchos menos recursos.

Además, al haber sido desarrollada por nosotros, está hecha de tal forma que sea mucho más simple de entender y con un código menos complejo y mucho mejor comentado para permitir que cualquier modificación sea mucho más rápida y que la resolución de errores sea menos complicada.

- En principio teníamos un Synth vacío, por lo que en un primer momento lo que el usuario ve es una ventana totalmente blanca con los textos en negro. En esta interfaz no se puede identificar nada, ya que todo tiene el mismo fondo blanco y no permite identificar dónde comienza y termina cada componente.

  Synth Look and Feel no suministra ninguna interfaz por defecto. Si un usuario quiere crear una apariencia nueva con la tecnología Synth, debe especificar la apariencia de todos y cada uno de los componentes en todos sus posibles estados. Una buena idea habría sido que Synth permitiera por ejemplo que para todo componente no personalizado, la apariencia sea la misma que la que tienen las ventanas en Java por defecto. Pero no es el caso, ya que cada componente no personalizado aparece con el fondo blanco y las letras en negro.

  Esto desmotiva mucho y hace de la creación de interfaces personalizadas una tarea dura y aburrida. Aunque con nuestra aplicación el proceso es mucho más sencillo e intuitivo, el comenzar con una ventana en blanco es complicado y difícil de entender.

  Solución: Decidimos tener un Synth por defecto con el aspecto de todos los componentes de los que disponemos. De esta forma, el usuario ya se hace una idea de cómo funciona la aplicación con sólo echar un vistazo y modificar algunas propiedades. Además, teniendo una interfaz como referencia, es mucho más fácil modificarla para que tome el aspecto deseado que crearla desde cero.

  Como añadido, si un usuario no quiere crear toda la interfaz completa, puede modificar los aspectos que crea convenientes y dejar que sea la apariencia por defecto la que se muestre en el resto de casos.

6. Conclusiones

Para concluir vamos a analizar si después de este tiempo se han cumplido los objetivos que nos hemos marcado, para ello pasamos a detallar objetivo por objetivo si se ha cumplido o no y por qué.

- *Creación de una aplicación que permita la personalización de todos los componentes de una interfaz gráfica de usuario realizada con Swing, mediante el empleo de la funcionalidad Synth Look and Feel que aporta el API de Java.*

  La idea inicial fue la creación de una aplicación que sentara las bases para una mayor implantación de esta tecnología de Java, que tras varios años de existencia aún no ha tomado una importancia muy relevante entre los desarrolladores.

  A nuestro juicio, la potencia de *Synth* está aún por ser descubierta, y gracias a aplicaciones como ésta, se da un paso adelante para que sea así.

  Este objetivo se ha cumplido en su mayoría ya que hemos implementado los componentes más utilizados en cualquier aplicación Swing. En particular, hemos desarrollado el código necesario para poder personalizar botones, botones de radio, cajas de selección, botones de toggle, pestañas, barras de menú, combo boxes y todos los subcomponentes necesarios.

  Es cierto que aún quedan varios componentes por incluir, pero la arquitectura de la aplicación permite añadir nuevas funcionalidades de manera muy sencilla debido a la jerarquía de clases y de herencia desarrollada.

  Sólo el tiempo ha sido el motivo que ha hecho que no podamos finalizarlo, puesto que las barreras tecnológicas ya fueron superadas desde el primer componente.

- *Creación y mantenimiento de una web para unificar la poca información que existe en la red sobre Synth y añadir nueva información a partir de nuestras investigaciones. Esta web se podrá consultar en dos idiomas: inglés y español.*

  Pensamos en un principio que la web era una de las maneras de llegar a la gente con más facilidad. Decidimos crear un sitio web que unificara la poca información que existe en la red sobre *Synth*, y añadir todo lo que fuéramos descubriendo durante nuestro desarrollo. De este modo, todos los que, como nosotros, sintieran curiosidad por esta tecnología, podrían encontrar toda la información requerida para seguir adelante. Además, emplearíamos esta web como medio de descarga de nuestra aplicación.

  Proporcionar la página en inglés era indispensable para poder llegar a un número considerable de personas, pero pensamos que también era necesario tener toda la información en español.

Se compró el dominio, se buscó alojamiento (servicio de hosting), se diseñó la web y se utilizó un gestor de contenidos CMS (Drupal) para su creación. Se preparó para el soporte de múltiples idiomas y se inició la subida de información.

Pero volvimos a encontrarnos con limitaciones de tiempo. Todos los documentos que creamos había que traducirlos y subirlos. Además, mucha de la documentación que era lógico incluir en la web se distanciaba mucho de lo que necesitábamos para crear nuestra aplicación, y lógicamente dimos prioridad a esta última.

Es por ello que de la web sólo se puede observar la estructura y algún tutorial, pero no ha avanzado todo lo que hubiésemos deseado.

- *La aplicación será un plugin de Eclipse, para poder unificar la creación de la aplicación y la apariencia de la interfaz gráfica personalizada.*

  Esto se descartó al principio de la implementación ya que no le vimos utilidad. Crear una apariencia personalizada para una interfaz es totalmente independiente de cualquier proceso en un proyecto, debido a que la apariencia debe poder ser utilizada en cualquier aplicación Java independientemente de la funcionalidad de la misma. Por eso, se ganaba muy poco creando un plugin (poder insertar automáticamente el código que se necesita para que se cargue el *look and feel* que hemos creado).

  Sin embargo, tenía muchos problemas. La faceta de creación de la apariencia normalmente sería llevada a cabo por un diseñador gráfico, que no tiene ninguna necesidad de aprender nada relacionado con la programación, ni mucho menos sobre Eclipse. También se limita con ello el uso de nuestro software a un solo entorno de desarrollo, con la pérdida de usuarios que eso supone. También tuvimos en cuenta la limitación que conlleva el crear un plugin, ya que hay que ceñirse a ciertas estructuras para poder integrarlo en el IDE.

  Por ello, al final omitimos este objetivo, y decidimos crear una aplicación de escritorio totalmente independiente. Esto nos ha supuesto muchas ventajas, sobre todo en cuanto a la mayor libertad para desarrollar la arquitectura como más nos ha interesado.

- *Se dotará de todas las funcionalidades de usabilidad que sea posible aplicar, como herramientas de edición, de internacionalización, etc.*

  Este objetivo se ha cumplido y es uno de los más importantes. A la hora de diseñar tuvimos en cuenta todos estos factores para que la aplicación sea más atractiva para el usuario.

  La utilidad de herramientas de edición es indispensable en cualquier aplicación de hoy en día. El miedo que genera a un usuario el no poder volver atrás en una decisión tomada, hace que no experimente y no consiga los objetivos marcados. Además, herramientas como copiar y pegar facilitan mucho la labor de tareas repetitivas.

  La internacionalización de las aplicaciones también es una utilidad muy típica en los proyectos de software libre, y que hace que las personas se impliquen más en su uso. La

posibilidad de poder cambiar el idioma de forma fácil y de poder añadir nuevos idiomas fue un objetivo marcado desde el principio y en el que pusimos un gran esfuerzo. El resultado fue muy bueno, y se consiguió sin problemas cambiar del inglés al español y viceversa, traduciendo absolutamente todas las cadenas que aparecen en la interfaz. Además, se ha dejado todo perfectamente preparado para que se puedan incluir más idiomas sin necesidad de tocar el código. Tan solo es necesario añadir un archivo a la carpeta de idiomas con las cadenas traducidas.

Todas estas herramientas de usabilidad han contribuido a crear una herramienta con un aspecto mucho más profesional, que confiere a los usuarios mayor seguridad y seriedad.

- *La aplicación será provista de una ayuda inteligente, que permita en cualquier momento conocer información muy concreta de la tarea que estemos realizando, permitiendo así al usuario avanzar a mayor rapidez.*

  La idea era que el usuario tuviera en todo momento información concreta sobre cualquier aspecto de la interfaz del que tuviera alguna duda, sin necesidad de tener que navegar por ayudas enormes y con mucha información irrelevante en ese momento.

  Por ello se tenían en mente dos ayudas principales: la primera se denomina ayuda rápida, y consiste en que pulsando F1 o con el botón derecho del ratón se pudiera acceder a ayuda sobre el elemento de la interfaz seleccionado en ese momento. De este modo, se le evita al usuario el tener que buscar en la ayuda principal, ahorrándole mucho tiempo y esfuerzo.

  Por otro lado, se pretendía dar cobertura a una ayuda estándar, explicando todos los aspectos relacionados con la interfaz y el funcionamiento de la misma, así como información sobre *Synth* que pudiera aclarar al usuario conceptos necesarios para poder realizar correctamente su tarea.
  Este objetivo se ha cumplido aunque no en su totalidad. La funcionalidad de ayuda rápida está desarrollada, pero la información de ayuda no ha sido introducida para todos los componentes y todas las propiedades de los mismos. La ayuda general no fue descrita.

- *Dejar un código abierto y preparado para que sea fácil su modificación y añadir nuevas funcionalidades. Para ello, se debe realizar un código limpio, bien estructurado, organizado y comentado.*

  Este objetivo, uno de los más importantes, se ha cumplido en su totalidad ya que además de tener un código estructurado y totalmente comentado en ingles, añadir nuevos componentes es bastante sistemático gracias al gran esfuerzo invertido en la fase de diseño.

  Todo el desarrollo fue organizado de tal manera que obtuviéramos como resultado una arquitectura fácil de entender por personas ajenas al proyecto, y sobre todo fácil de modificar, atributo indispensable en cualquier proyecto de software libre. Aunque algunas funcionalidades no hayan sido finalizadas, el código ha quedado en perfecto

estado para que cualquier persona interesada en el mismo pueda seguir la labor de creación.

Creemos que hemos conseguido un gran resultado, y que en un futuro será posible que otras personas finalicen nuestra labor, para que por fin exista una aplicación que permita explotar todas las capacidades que otorga la tecnología de creación de apariencias personalizadas creada por Sun.

Podemos concluir diciendo que hemos cumplido la mayoría de los objetivos que nos fijamos. Nos han sido de gran ayuda los conocimientos adquiridos en Ingeniería del Software que hemos puesto en práctica en varios momentos del desarrollo de esta aplicación, a la hora de refactorizar, utilizar patrones de diseño que facilitaran la inclusión de herramientas como el deshacer, rehacer, etc…

Además el desarrollo de esta aplicación nos ha servido para ampliar nuestros conocimientos tanto en el desarrollo de interfaces gráficas como en aspectos de la tecnología Java que desconocíamos hasta ahora como el tratamiento de XML o el uso de Synth.

Ha sido una labor dura de investigación, ya que la falta de información en internet ha hecho que muchos de nuestros hallazgos hayan tenido que ser realizados mediante métodos de ensayo y error, lo que ha conllevado una enorme cantidad de tiempo.

Pero gracias a esto, nuestros conocimientos sobre Synth probablemente sean mayores que los de la gran mayoría de usuarios de Java y Swing, ya que es una tecnología muy desconocida y dominada por muy pocas personas en el mundo.

7. Trabajo Futuro

Aunque hemos realizado un avance muy importante y la aplicación está prácticamente finalizada, aún hay ciertos objetivos que no han sido cumplidos en su totalidad. Es ahí donde se debe poner un mayor énfasis a la hora de avanzar en el proyecto.

Las tareas a realizar en un futuro serían:

- Incorporar los componentes que faltan para permitir la personalización de todos los componentes de Swing. En particular son los siguientes:
    - Desktop Icon
    - Desktop Pane
    - Editor Pane
    - File Chooser
    - Formatted Text Field
    - Internal Frame
    - Internal Frame Title Pane
    - Label
    - List
    - Option Pane
    - Panel
    - Password Field
    - Popup Menu
    - Popup Menu Separator
    - Progress Bar
    - Root Pane
    - Scroll Bar
    - Scroll Bar Thumb
    - Scroll Bar Track
    - Scroll Pane

  - Slider
  - Slider Thumb
  - Slider Track
  - Spinner
  - Split Pane Divider
  - Table
  - Table Header
  - Text Area
  - Text Pane
  - Tool Bar
  - Tool Bar Content
  - Tool Bar Drag Window
  - Tool Bar Separator
  - Tool Tip
  - Tree
  - Tree Cell
  - Viewport

- Finalizar la ayuda. Aunque están creadas las estructuras necesarias para albergar una ayuda (sobre todo para la ayuda rápida), aún se debe redactar toda la ayuda. Es un paso importante para que los usuarios se animen a usar esta herramienta

- Mejorar la página web. Faltan por añadir algunos tutoriales, y sobre todo mucha información sobre *Synth* y sobre cómo trabajar con él, así como enlaces a otros sitios de interés. La página web es un recurso indispensable para dar a conocer nuestra aplicación, y por tanto sería necesario dedicarle mucho esfuerzo en el futuro.

8. Bibliografía

Sitios web:

- Información general y ejemplos sobre *Synth:*
  - http://www.javadesktop.org/articles/synth/index.html
  - http://www.ibm.com/developerworks/java/library/j-tiger10194/
  - http://www.ibm.com/developerworks/java/library/j-synth/
- Documentación sobre *Synth:*
  - Formato: http://java.sun.com/j2se/1.5.0/docs/api/javax/swing/plaf/synth/doc-files/synthFileFormat.html
  - Regiones modificables: http://java.sun.com/j2se/1.5.0/docs/api/javax/swing/plaf/synth/Region.html
  - Propiedades específicas de cada región: http://java.sun.com/j2se/1.5.0/docs/api/javax/swing/plaf/synth/doc-files/componentProperties.html
- Internacionalización en Java y Netbeans
  - http://java.sun.com/docs/books/tutorial/i18n/intro/quick.html
- Página oficial de SWT:
  - http://www.eclipse.org/swt/
- Tutoriales sobre SWT:
  - http://www.oehive.org/node/150

Libros:

- Robert Eckstein (Author), Marc Loy (Author), Dave Wood (Author). ***Java Swing***. O'Reilly.1998.
- Kathy Walrath, Mary Campione, Alison Huml y Sharon Zakhour. ***The JFC Swing Tutorial Second Edition***. Addison-Wesley. 2004.
- David M. Geary . ***Graphic Java 2: Mastering the JFC, Volume II: Swing***. Sun Microsystems Press Java Series. 2001.
- Robert Harris and Rob Warner**. The *Definitive Guide to SWT and JFACE***. Apress. 2004.
- Jackwind Li Guojie . ***Professional Java Native Interfaces with SWT/JFace (Programmer to Programmer)***. Wrox. 2005
- Eric Clayberg and Dan Rubel. ***Eclipse: Building Commercial-Quality Plug-ins (2nd Edition) (The Eclipse Series)***. Addison-Wesley Professional. 2006.

APÉNDICE A: Documentación de las pruebas

Tal y como se explica en el capítulo 5.2, durante las diferentes etapas del desarrollo de la aplicación se realizaron baterías de pruebas.

Dada la imposibilidad de la automatización de éstas, se creó la siguiente plantilla para la exposición estandarizada del resultado de las mismas:

Componente: **Versión:** **Fecha:** de de 2008

1. **Pruebas sobre propiedades individuales (probadas en el estado general, o si no procede, en uno de los estados en los que se encuentre la propiedad):**

Propiedad	Prueba Satisfactoria		Observaciones	Solucionado
	Sí	No		

2. **Pruebas sobre estados individuales (rellenar sólo si todas las pruebas de la primera parte son satisfactorias):**

Estado	Prueba Satisfactoria		Observaciones	Solucionado
	Sí	No		

3. **Prueba general (rellenar sólo si todas las pruebas de la segunda parte son satisfactorias):**

Prueba Satisfactoria		Observaciones:	Solucionado
Sí	No		

Tomando como base la anterior plantilla, se realizaron las pruebas de integración correspondientes a cada componente de forma manual, modificando una a una todas las propiedades de cada posible estado, comprobando si se obtenía el resultado deseado, y anotándolo en la plantilla.

A continuación se expone la documentación de los resultados de las pruebas de integración realizadas a la aplicación durante su desarrollo:

Componente: Botón **Versión:** 1 **Fecha:** 2 de abril de 2008

4. **Pruebas sobre propiedades individuales (probadas en el estado general, o si no procede, en uno de los estados en los que se encuentre la propiedad):**

Propiedad	Prueba Satisfactoria		Observaciones	Solucionado
	Sí	No		
Márgenes	x			
Imagen de Fondo		x	Al abrir un proyecto, no se está actualizando la ruta, porque el insets coge "null" como ruta del proyecto y da error al abrir el insets chooser.	5-04-2008
Fuente	x			
Opaco	¿?	¿?	No sé probarlo	5-04-2008 -> el xml se genera bien
Rellenar contenido	¿?	¿?	No sé probarlo	5-04-2008 -> el xml se genera bien.
Píxeles entre imagen y texto		x	No se deja espacio entre la imagen del botón y del texto, da igual el valor que se le ponga	5-04-2008
Tecla Intro pulsa el foco		x	En nuestra aplicación no hay manera de probarlo	
Imagen del borde	-	-	Al abrir un proyecto, no se está actualizando la ruta, porque el insets coge "null" como ruta del proyecto y da error al abrir el insets chooser.	5-04-2008
Píxeles efecto 3D	x			

5. **Pruebas sobre estados individuales (rellenar sólo si todas las pruebas de la primera parte son satisfactorias):**

Estado	Prueba Satisfactoria		Observaciones	Solucionado
	Sí	No		

6. **Prueba general (rellenar sólo si todas las pruebas de la segunda parte son satisfactorias):**

Prueba Satisfactoria		Observaciones:	Solucionado
Sí	No		

Componente: Button **Versión:** 2 **Fecha:** 2 de abril de 2008

1. **Pruebas sobre propiedades individuales (probadas en el estado general, o si no procede, en uno de los estados en los que se encuentre la propiedad):**

Propiedad	Prueba Satisfactoria		Observaciones	Solucionado
	Sí	No		
Márgenes	X			
Imagen de Fondo	X			
Fuente	X			
Opaco	¿?	¿?	No se puede probar en nuestra aplicación	
Rellenar contenido	¿?	¿?	No se puede probar en nuestra aplicación	
Píxeles entre imagen y texto	X		Solucionado sobre la marcha. Había un fallo al generar el XML que no lo incluía.	
Tecla Intro pulsa el foco	¿?	¿?	No se puede probar, ¿pero realmente sirve de mucho?	
Imagen del borde	X			
Píxeles efecto 3D	X		Solucionado sobre la marcha. Tenía puesto 1 como valor por defecto, y Synth tiene un 0 por defecto	

2. **Pruebas sobre estados individuales (rellenar sólo si todas las pruebas de la primera parte son satisfactorias):**

Estado	Prueba Satisfactoria		Observaciones	Solucionado
	Sí	No		
DEFAULT	X			
MOUSE OVER	X		El color de fuente no funciona, como de costumbre.	
MOUSE OVER AND FOCUSED	X			
FOCUSED	X			
FOCUSED AND PRESSED	X			
DISABLED	X			

3. **Prueba general (rellenar sólo si todas las pruebas de la segunda parte son satisfactorias):**

Prueba Satisfactoria		Observaciones:	Solucionado
Sí	No		
X			

Componente: TextField **Versión:** 1 **Fecha:** 10 de abril de 2008

1. **Pruebas sobre propiedades individuales (probadas en el estado general, o si no procede, en uno de los estados en los que se encuentre la propiedad):**

Propiedad	Prueba Satisfactoria		Observaciones	Solucionado
	Sí	No		
Márgenes	X			
Imagen de Fondo	X			
Fuente	X			
Opaco	¿?			
Tasa de parpadeo del cursor	X			
Imagen del borde	X			

2. **Pruebas sobre estados individuales (rellenar sólo si todas las pruebas de la primera parte son satisfactorias):**

Estado	Prueba Satisfactoria		Observaciones	Solucionado
	Sí	No		
GENERAL	X			
DISABLED	X			

3. **Prueba general (rellenar sólo si todas las pruebas de la segunda parte son satisfactorias):**

Prueba Satisfactoria		**Observaciones:**	**Solucionado**
Sí	**No**		
X			

Componente: ToggleButton **Versión:** 1 **Fecha:** 10 de abril de 2008

1. **Pruebas sobre propiedades individuales (probadas en el estado general, o si no procede, en uno de los estados en los que se encuentre la propiedad):**

Propiedad	Prueba Satisfactoria		Observaciones	Solucionado
	Sí	No		
Márgenes	X			
Icono	X			
Imagen de Fondo	X			
Fuente	X			
Opaco	¿?			
Rellenar contenido	¿?			
Píxeles entre imagen y texto	X			
Tecla Intro pulsa el foco	¿?			
Imagen del borde	X			
Píxeles efecto 3D	X			

2. **Pruebas sobre estados individuales (rellenar sólo si todas las pruebas de la primera parte son satisfactorias):**

Estado	Prueba Satisfactoria		Observaciones	Solucionado
	Sí	No		
GENERAL	X			
MOUSE OVER	X			
MOUSE OVER AND FOCUSED	X			
FOCUSED	X			
DISABLED	X			
SELECTED	X			
SELECTED AND DISABLED	X			
SELECTED AND MOUSE OVER	X			

3. **Prueba general (rellenar sólo si todas las pruebas de la segunda parte son satisfactorias):**

Prueba Satisfactoria		Observaciones:	Solucionado
Sí	No		
X			

Componente: CheckBox **Versión:** 1 **Fecha:** 16 de abril de 2008

1. **Pruebas sobre propiedades individuales (probadas en el estado general, o si no procede, en uno de los estados en los que se encuentre la propiedad):**

Propiedad	Prueba Satisfactoria		Observaciones	Solucionado
	Sí	No		
Márgenes	X			
Icono	X			
Imagen de fondo	X			
Píxeles efecto 3D	X			
Fuente		X	Falla el color de la fuente	
Opaco	¿?			
Rellenar contenido		X	Error en el programa: no se genera bien el Xml	16-04-08
Píxeles entre imagen y texto	X			
Tecla Intro pulsa el foco	¿?			
Imagen del borde	¿?			

2. **Pruebas sobre estados individuales (rellenar sólo si todas las pruebas de la primera parte son satisfactorias):**

Estado	Prueba Satisfactoria		Observaciones	Solucionado
	Sí	No		
GENERAL				
MOUSE OVER				
MOUSE OVER AND FOCUSED				
FOCUSED				
DISABLED				
SELECTED				
SELECTED AND DISABLED				
SELECTED AND MOUSE OVER				

3. **Prueba general (rellenar sólo si todas las pruebas de la segunda parte son satisfactorias):**

Prueba Satisfactoria		Observaciones:	Solucionado
Sí	No		

Componente: CheckBox **Versión:** 2 **Fecha:** 16 de abril de 2008

1. **Pruebas sobre propiedades individuales (probadas en el estado general, o si no procede, en uno de los estados en los que se encuentre la propiedad):**

Propiedad	Prueba Satisfactoria		Observaciones	Solucionado
	Sí	No		
Márgenes	X			
Icono	X			
Imagen de fondo	X			
Píxeles efecto 3D	X			
Fuente	X	X	Funciona a veces sí y a veces no	
Opaco	¿?			
Rellenar contenido	X			
Píxeles entre imagen y texto	X			
Tecla Intro pulsa el foco	¿?			
Imagen del borde	¿?			

2. **Pruebas sobre estados individuales (rellenar sólo si todas las pruebas de la primera parte son satisfactorias):**

Estado	Prueba Satisfactoria		Observaciones	Solucionado
	Sí	No		
GENERAL	X			
MOUSE OVER	X			
MOUSE OVER AND FOCUSED	X			
FOCUSED	X			
DISABLED	X			
SELECTED	X			
SELECTED AND DISABLED	X			
SELECTED AND MOUSE OVER	X			
SELECTED AND MOUSE OVER AND FOCUSED	X			

3. **Prueba general (rellenar sólo si todas las pruebas de la segunda parte son satisfactorias):**

Prueba Satisfactoria		Observaciones:	Solucionado
Sí	No		
X		Salvo por el color de la fuente	

Componente: RadioButton **Versión:** 1 **Fecha:** 16 de abril de 2008

1. **Pruebas sobre propiedades individuales (probadas en el estado general, o si no procede, en uno de los estados en los que se encuentre la propiedad):**

Propiedad	Prueba Satisfactoria		Observaciones	Solucionado
	Sí	No		
Márgenes	X			
Icono	X			
Imagen de fondo	X			
Píxeles efecto 3D	X			
Fuente	X	X	El color de la fuente no funciona siempre	
Opaco	¿?			
Rellenar contenido	¿?			
Píxeles entre imagen y texto	X			
Tecla Intro pulsa el foco	¿?			
Imagen del borde	¿?			

2. **Pruebas sobre estados individuales (rellenar sólo si todas las pruebas de la primera parte son satisfactorias):**

Estado	Prueba Satisfactoria		Observaciones	Solucionado
	Sí	No		
GENERAL	X			
MOUSE OVER	X			
MOUSE OVER AND FOCUSED	X			
FOCUSED	X			
DISABLED	X			
SELECTED	X			
SELECTED AND DISABLED	X			
SELECTED AND MOUSE OVER	X			
SELECTED AND MOUSE OVER AND FOCUSED	X			

3. **Prueba general (rellenar sólo si todas las pruebas de la segunda parte son satisfactorias):**

Prueba Satisfactoria		Observaciones:	Solucionado
Sí	No		
X		Salvo por el color de la fuente	

Componente: TabbedPaneTab **Versión:** 1 **Fecha:** 17 de mayo de 2008

1. **Pruebas sobre propiedades individuales (probadas en el estado general, o si no procede, en uno de los estados en los que se encuentre la propiedad):**

Propiedad	Prueba Satisfactoria		Observaciones	Solucionado
	Sí	No		
Márgenes	X			
Imagen de fondo	X			
Fuente	X	X	El color de la fuente no funciona siempre	
Opaco	¿?			
Imagen del borde	X			

2. **Pruebas sobre estados individuales (rellenar sólo si todas las pruebas de la primera parte son satisfactorias):**

Estado	Prueba Satisfactoria		Observaciones	Solucionado
	Sí	No		
GENERAL	X			
MOUSE OVER	X			
MOUSE OVER AND FOCUSED	X			
FOCUSED	X			
DISABLED	X			
SELECTED	X			
SELECTED AND DISABLED	X			
SELECTED AND MOUSE OVER	X			
DISABLED	X			

3. **Prueba general (rellenar sólo si todas las pruebas de la segunda parte son satisfactorias):**

Prueba Satisfactoria		Observaciones:	Solucionado
Sí	No		
X		Salvo por el color de la fuente	

Componente: TabbedPaneTabArea **Versión:** 1 **Fecha:** 17 de mayo de 2008

1. **Pruebas sobre propiedades individuales (probadas en el estado general, o si no procede, en uno de los estados en los que se encuentre la propiedad):**

Propiedad	Prueba Satisfactoria		Observaciones	Solucionado
	Sí	No		
Márgenes	X			
Imagen de fondo	X			
Fuente	¿?		No podemos poner texto	
Opaco	¿?			
Imagen del borde	X			

2. **Pruebas sobre estados individuales (rellenar sólo si todas las pruebas de la primera parte son satisfactorias):**

Estado	Prueba Satisfactoria		Observaciones	Solucionado
	Sí	No		
GENERAL	X			

3. **Prueba general (rellenar sólo si todas las pruebas de la segunda parte son satisfactorias):**

Prueba Satisfactoria		Observaciones:	Solucionado
Sí	No		
X			

Componente: TabbedPane **Versión:** 1 **Fecha:** 17 de mayo de 2008

1. **Pruebas sobre propiedades individuales (probadas en el estado general, o si no procede, en uno de los estados en los que se encuentre la propiedad):**

Propiedad	Prueba Satisfactoria		Observaciones	Solucionado
	Sí	No		
Márgenes	X			
Imagen de fondo	X			
Opaco	¿?			
Márgenes pestaña selección	X			
Píxeles de superposición	X			
Selección sigue el foco	¿?			
Imagen del borde	X			

2. Pruebas sobre estados individuales (rellenar sólo si todas las pruebas de la primera parte son satisfactorias):

Estado	Prueba Satisfactoria		Observaciones	Solucionado
	Sí	No		
GENERAL	X			

3. Prueba general (rellenar sólo si todas las pruebas de la segunda parte son satisfactorias):

Prueba Satisfactoria		Observaciones:	
Sí	No		
X			

Componente: TabbedPaneContent **Versión:** 1 **Fecha:** 17 de mayo de 2008

1. **Pruebas sobre propiedades individuales (probadas en el estado general, o si no procede, en uno de los estados en los que se encuentre la propiedad):**

Propiedad	Prueba Satisfactoria		Observaciones	Solucionado
	Sí	No		
Márgenes	X			
Imagen de fondo	X			
Opaco	¿?			
Imagen de borde	X			

2. **Pruebas sobre estados individuales (rellenar sólo si todas las pruebas de la primera parte son satisfactorias):**

Estado	Prueba Satisfactoria		Observaciones	Solucionado
	Sí	No		
GENERAL	X			

3. **Prueba general (rellenar sólo si todas las pruebas de la segunda parte son satisfactorias):**

Prueba Satisfactoria		Observaciones:	Solucionado
Sí	No		
X			

APÉNDICE B: Actas

ACTA DE LA REUNIÓN DE SISTEMAS INFORMÁTICOS

PROYECTO SYNTH EDITOR

Acta 1

Fecha: 18 Octubre 2007

TEMAS TRATADOS:

Se ha analizado la herramienta que tenemos y hemos decidido descartarla ya que tiene muchos fallos, funciona mal y el código está sin comentar.

PLAN DE TRABAJO:

- Fecha entrega: 25 Octubre
  Investigación sobre cómo hacer un plug-in para eclipse.

- Fecha entrega: 8 Noviembre
  - Implementación de un pequeño plug-in.
  - Escritura de un pequeño tutorial sobre cómo hemos realizado el plug-in.
- Crear un grupo en Sourceforge.
- Grupo para documentación.

ACTA DE LA REUNIÓN DE SISTEMAS INFORMÁTICOS

PROYECTO SYNTH EDITOR

Acta 2

Fecha: 8 Noviembre 2007

TEMAS TRATADOS:

- Desarrollo de un plug-in para Eclipse
  Hemos descartado desarrollar un plug-in para Eclipse ya que nos hemos dado cuenta de que no es útil para el usuario que va a manejar nuestra aplicación, para poder modificar la apariencia no tiene porqué ser necesario que el usuario esté programando en el entorno de desarrollo de Eclipse.

- Desarrollo de una aplicación separada
  Nos decantamos por los motivos anteriormente citados a desarrollar una aplicación independiente del entorno de programación.

- Descartamos Eclipse en favor de NetBeans o SunOne Studio.

- Método de trabajo:
  Utilización del CVS:
  Rama principal (trunk): En esta rama dispondremos de la última versión estable del proyecto (Biblioteca de trabajo).
  Ramas secundarias: Utilizaremos una rama secundaria para cada uno (bibliotecas de trabajo) para asi poder ir desarrollando en paralelo sin estropear la versión estable. A la hora de integrar alguna de estas con otra rama secundaria podremos crear una rama secundaria auxiliar como biblioteca de integración.

- Documentación que debe tener el proyecto:

  - Diagramas UML

  - Documento donde vayamos identificando las dificultades o posibles dificultades que pueden ir surgiendo, aquí se expondrá el problema, las posibles soluciones y la solución adoptada.

  - Otros posibles documentos en los que apliquemos los conocimientos adoptados en la asignatura de Ingeniería del Software.

PLAN DE TRABAJO:

- Fecha entrega: 15 Noviembre
    - Diseño de una pequeña interfaz.
    - Investigar sobre los distintos métodos de parseo xml y decantarnos por uno, explicar por qué lo rechazamos frente a otros.
    - Crear las actas de reunión sobre las distintas reuniones que hemos tenido hasta ahora.
- Fecha entrega: 22 Noviembre
    - Definición inicial del formato del XML.
    - Incluir los componentes: Botones, ChekBox y RadioButton.
- Fecha entrega: 29 Noviembre
    - Deshacer.
    - Cambio de idioma.
- Fecha entrega: 6 Diciembre
    - Cargar y Guardar simples.
- Fecha entrega: 13 Diciembre
    - Prototipo con todo lo citado anteriormente.

TEMAS A TRATAR POSTERIORMENTE:

- En cuanto a la usabilidad, debemos pensar si queremos ofrecer la posibilidad de mostrar u ocultar componentes.
- Organización de la interfaz: pestañas,…
- Posibilidad de definir un aspecto distinto para componentes que tienen un mismo nombre predeterminado.
- Decidir qué patrones de usabilidad vamos a utilizar.
- Diseño de la interfaz.

ACTA DE LA REUNIÓN DE SISTEMAS INFORMÁTICOS

PROYECTO SYNTH EDITOR

Acta 3

Fecha: 15 Noviembre 2007

TEMAS TRATADOS:

- Posibilidad de cambiar en la interfaz de ejemplo el árbol por una tabla árbol.
- Al realizar la interfaz de ejemplo detectamos el siguiente problema: si cambiamos el look and feel de la ventana de previsualización, también cambia el look and feel de la ventana donde introducimos los cambios. Se plantean las siguientes posibles soluciones:
    - Lanzar cada ventana en una aplicación por separado.
    - Desarrollar la ventana de cambios con AWT y la de previsualización con Swing.
    - Modificar el código de la clase UIManager para que se adapte a nuestras necesidades.
    - Modificar el método repintar para que vaya cambiando el look and feel dependiendo de la ventana.
- Al rediseñar la interfaz debemos tener en cuenta:
    - Ofrecer al usuario la posibilidad de elegir en las opciones de los componentes sólo aquellas que sean posibles (en la interfaz de ejemplo se da la posibilidad de elegir combinaciones que no se pueden dar).
    - Mostrar de un color las opciones que han sido modificadas y las otras de otro color.
    - Posibilidad de volver a valores por defecto.
    - Utilizar layouts como AbsoluteLayout o GridBagLayout.
    - Mostrar en la interfaz de forma más visible únicamente las opciones más habituales y ocultas en los menús las opciones avanzadas.
    - Opción de poder eliminar.
- Retrasamos el plan de trabajo de la reunión anterior una semana debido a las dificultades surgidas.

PLAN DE TRABAJO:

- Fecha entrega: 22 Noviembre
    - Investigar sobre cómo vamos a hacer las dos interfaces teniendo en cuenta el problema que ha surgido con el look and feel y las posibles soluciones tratadas en la reunión.
    - Averiguar todas las propiedades o estados de los tres componentes que vamos a añadir a nuestra interfaz de prototipo (button, chekBox y radioButton).
    - Parser.
- Fecha entrega: 29 Noviembre
    - Definición inicial del formato del XML.
    - Incluir los componentes: Botones, CheckBox y RadioButton.
- Fecha entrega: 6 Diciembre
    - Deshacer.
    - Cambio de idioma.
- Fecha entrega: 13 Diciembre
    - Cargar y Guardar simples.
- Fecha entrega: 20 Diciembre
    - Prototipo con todo lo citado anteriormente.

ACTA DE LA REUNIÓN DE SISTEMAS INFORMÁTICOS

PROYECTO SYNTH EDITOR

Acta 4

Fecha: 22 Noviembre 2007

TEMAS TRATADOS:

- Debemos tener cuidado con las licencias de los componentes que reutilicemos.
- Mostrar en la interfaz SWT, de forma más llamativa, que componente y a qué estado le estamos cambiando las propiedades.
- Probar el método invalidate/validate para cambiar el LookAndFeel en la interfaz Swing.
- Comenzar creando el formato XML para el Button.
- Cambiamos la planificación inicial del 13 de diciembre al día 6 de diciembre.

PLAN DE TRABAJO:

- Fecha entrega: 29 Noviembre
  - Definición inicial del formato del XML.
  - Incluir los componentes: Botones, CheckBox y RadioButton.
- Fecha entrega: 6 Diciembre
  - Cargar y Guardar simples.
- Fecha entrega: 13 Diciembre
  - Deshacer.
  - Cambio de idioma.
- Fecha entrega: 20 Diciembre
  - Prototipo con todo lo citado anteriormente.

ACTA DE LA REUNIÓN DE SISTEMAS INFORMÁTICOS

PROYECTO SYNTH EDITOR

Acta 5

Fecha: 29 Noviembre 2007

TEMAS TRATADOS:

- Posibilidad de utilizar el patrón cadena de responsabilidades para resolver los problemas que surgen en casos como el Scroll ya que queremos ocultar al usuario el hecho de que éste está compuesto por tres elementos distintos, con lo que cada componente decidiría que parte de lo que se cambia es suya.

PLAN DE TRABAJO:

- Fecha entrega: 6 Diciembre
    - Conectar la interfaz gráfica con el modelo y el xml.
- Fecha entrega: 13 Diciembre
    - Deshacer.
    - Cambio de idioma.
- Fecha entrega: 20 Diciembre
    - Prototipo con todo lo citado anteriormente.

ACTA DE LA REUNIÓN DE SISTEMAS INFORMÁTICOS

PROYECTO SYNTH EDITOR

Acta 6

Fecha: 13 Diciembre 2007

TEMAS TRATADOS:

- Árbol de propiedades de la vista
  Se ha decidido que cada propiedad sea un objeto diferenciado que herede de una clase padre para permitir mayor grado de personalización, en particular para que:
    - Al modificar una propiedad, se modifiquen las que son dependientes de ella, poniendo un valor por defecto u obligando al usuario a insertarlas.
    - Al modificar una propiedad en un estado, se modifique en varios para dar un aspecto lógico en caso de que el usuario decida no personalizar algunos estados.

  Quitar de los estados aquellas propiedades que sólo tengan sentido para un estado (por ejemplo, el efecto 3D al pulsar el botón).
- Centrarnos en hacer más robusto lo que ya está implementado, en lugar de seguir avanzando en la implementación.
    - Solucionar el comportamiento del estado “General”.
    - Añadir el estado “Default”.
    - Mejorar la opción de “Guardar”, para que guarde por defecto en la carpeta de trabajo.
- Terminar la opción de “Abrir”.
- En caso de que haya tiempo, ir pensando en el “Deshacer”.

PLAN DE TRABAJO:

- Fecha entrega: 20 Diciembre
    - Prototipo con las funciones de Abrir, Guardar y manejo completo para el componente Botón.

TEMAS A TRATAR POSTERIORMENTE:

- Se desplaza la funcionalidad de “Deshacer”
- En cuanto a la usabilidad, debemos pensar si queremos ofrecer la posibilidad de mostrar u ocultar componentes.
- Organización de la interfaz: pestañas,…
- Posibilidad de definir un aspecto distinto para componentes que tienen un mismo nombre predeterminado.
- Decidir qué patrones de usabilidad vamos a utilizar.
- Diseño de la interfaz.

ACTA DE LA REUNIÓN DE SISTEMAS INFORMÁTICOS

PROYECTO SYNTH EDITOR

Acta 7

Fecha: 10 Enero 2007

TEMAS TRATADOS:

- Con el fin de mejorar la usabilidad, se plantea la posibilidad de coordinar la ventana de cambios con la ventana de visualización, de forma que si queremos realizar un cambio en un determinado atributo de un componente automáticamente se muestre la pestaña de dicho componente en la ventana de visualización y viceversa.

- La posibilidad de cambiar el idioma de la aplicación conllevará obligatoriamente a reiniciar la aplicación.

- Synth por defecto
  Podemos intentar emular el Synth de Swing ya que el Synth vacío puede resultar algo extraño al usuario. Dando la posibilidad a un usuario avanzado de empezar con el vacío. Para ver la carga de trabajo que nos puede acarrear, decidimos intentar emular el Synth de Swing para los componentes que ya tenemos.

PLAN DE TRABAJO

- Arreglar lo siguiente:
  - Guardar correctamente el color de la fuente.
  - Deshacer, no funciona correctamente y además hacer que funcione con el color de la fuente.
  - Abrir, no funciona correctamente.
  - Hacer que al seleccionar algo en el combo de la ventana de cambios, se cambie a la pestaña correspondiente en la ventana de visualización SwingDemo.
- Probar la aplicación en busca de nuevos fallos.

ACTA DE LA REUNIÓN DE SISTEMAS INFORMÁTICOS

PROYECTO SYNTH EDITOR

Acta 9

Fecha: 31 Enero 2007

TEMAS TRATADOS:

- Los problemas al cambiar de pestaña en el SwingDemo parecen ser producidos porque estamos tratando el evento *mouseClicked* en lugar de *mousePressed*.

- Utilizar valores por defecto para cada componente por los problemas que existen con las fuentes y que son ajenos a nuestra implementación.
  Como por ejemplo con los insets, si no ponemos una fuente por defecto, no funcionan.

- A raíz del punto anterior, debemos tener en cuenta que hay campos que no se tienen que dejar vacíos, no debemos permitir borrarlos como es el caso de la fuente, podemos hacer que al borrarlos se establezca la fuente por defecto.

- Fallo en el deshacer que parece ser producido porque se llega al estado inicial, debemos investigar por qué se produce.

- Fallo en la clase SwingDemoNuevo al hacer *loadSynthLF* nos da el error:
  "Parsing error: Synth XML malformed"
  también debemos investigar por qué.

- Planteamos una duda sobre si las imágenes que elija el usuario y que no se encuentren en la carpeta del proyecto, deberían copiarse a este por parte de la aplicación o debería ser el usuario el que las copiase.
  Concluimos en que el usuario copiará las imágenes en una carpeta específica del proyecto.

- Debemos probar a mantener la versión estable del proyecto en la rama principal.

ACTA DE LA REUNIÓN DE SISTEMAS INFORMÁTICOS

PROYECTO SYNTH EDITOR

Acta 9

Fecha: 31 Enero 2007

TEMAS TRATADOS:

- Los problemas al cambiar de pestaña en el SwingDemo parecen ser producidos porque estamos tratando el evento *mouseClicked* en lugar de *mousePressed*.

- Utilizar valores por defecto para cada componente por los problemas que existen con las fuentes y que son ajenos a nuestra implementación.
  Como por ejemplo con los insets, si no ponemos una fuente por defecto, no funcionan.

- A raíz del punto anterior, debemos tener en cuenta que hay campos que no se tienen que dejar vacíos, no debemos permitir borrarlos como es el caso de la fuente, podemos hacer que al borrarlos se establezca la fuente por defecto.

- Fallo en el deshacer que parece ser producido porque se llega al estado inicial, debemos investigar por qué se produce.

- Fallo en la clase SwingDemoNuevo al hacer *loadSynthLF* nos da el error:
  "Parsing error: Synth XML malformed"
  también debemos investigar por qué.

- Planteamos una duda sobre si las imágenes que elija el usuario y que no se encuentren en la carpeta del proyecto, deberían copiarse a este por parte de la aplicación o debería ser el usuario el que las copiase.
  Concluimos en que el usuario copiará las imágenes en una carpeta específica del proyecto.

- Debemos probar a mantener la versión estable del proyecto en la rama principal.

ACTA DE LA REUNIÓN DE SISTEMAS INFORMÁTICOS

PROYECTO SYNTH EDITOR

Acta 10

Fecha: 28 Febrero 2008

TEMAS TRATADOS:

- Actualmente tenemos una lista de estados común para todos los posibles elementos, debemos poner sólo los estados posibles para cada elemento.
- Fallo al cambiar las pestañas del SwingDemo, al cargar el LookAndFeel se cambian los componentes de la ventana principal con una pestaña de retraso.
- Creemos que no se ha subido al CVS la solución al fallo que daba al elegir un color y guardarlo, los valores R y G están intercambiados.
- Problema al cambiar el color en un estado que no es el general, la fuente no se muestra bien.
  Posibles soluciones:
  - Establecer una fuente por defecto.
  - Hacer dos cambios de LookAndFeel, uno de ellos que no sea Synth.
- En las pruebas ha dado un fallo el deshacer al cambiar el fondo del CheckBox.
- Debemos probar a mantener la versión estable del proyecto en la rama principal.
- Cambiar el abrir y guardar para guardar un proyecto en lugar de un xml.
- Documentación:
  - Estructura del proyecto.
  - Actas.
  - Problemas que hayan surgido y como los hemos solucionado.

ACTA DE LA REUNIÓN DE SISTEMAS INFORMÁTICOS

PROYECTO SYNTH EDITOR

Acta 11

Fecha: 26 Marzo 2008

TEMAS TRATADOS:

- Para que sea más cómodo para el usuario cambiar los insets de una imagen podemos poner flechas para aumentar y disminuir y que cambien los números en las cajas de texto.
- Al pulsar el botón para elegir una imagen, actualmente se abre la carpeta del proyecto, sería más cómodo abrir directamente la carpeta de imágenes.
- Eliminar la opción de cambiar el color del caret.
- Sería interesante tener un Synth por defecto .
- Debemos ir pensando en las ayudas que se van a dar al usuario, podemos ir implementando las que sabemos que no van a cambiar a lo largo del desarrollo como por ejemplo un “¿qué es esto?” para cada uno de los estados.
- Web:
    - Intentar que no desaparezcan los menús al entrar en el submenú de screenshoots.
- Documentación:
    - Problemas o dificultades que hayan surgido y cómo los hemos solucionado.

ACTA DE LA REUNIÓN DE SISTEMAS INFORMÁTICOS

PROYECTO SYNTH EDITOR

Acta 12

Fecha: 2 Abril 2008

TEMAS TRATADOS:

- Quitar el * al abrir un proyecto.
- Añadir el estado MouseOver and Focus en el look and feel por defecto.
- MouseOver and Focus parece que va mal en el componente CheckBox.
- Realizar una batería de pruebas para todos los componentes y estados que tenemos ya que nos puede resultar útil a la hora de realizar cambios.
- Pensar para la próxima reunión una lista de cosas que hemos hecho para ir realizando la estructura de lamemoria.
- En las siguientes semanas hasta la entrega del proyecto nos queda: introducir nuevos componentes (en principio los menús y las pestañas), establecer el look and feel por defecto para todos los componentes, la ayuda, documentación y pruebas realizadas.

ACTA DE LA REUNIÓN DE SISTEMAS INFORMÁTICOS

PROYECTO SYNTH EDITOR

Acta 13

Fecha: 9 Abril 2008

TEMAS TRATADOS:

- Posibles formas de hacer la ayuda para que resulte útil al usuario:
    - Como un html normalito con una página con índices al contenido.
    - Utilizar las herramientas de Google Desktop
    - Dirigir a nuestra página web, donde podemos tener una caja de búsqueda.
- Describir la estructura que va tomando el XML.
- Probar que todo funciona para el CheckBox.
- Próxima reunión: 23 Abril

ACTA DE LA REUNIÓN DE SISTEMAS INFORMÁTICOS

PROYECTO SYNTH EDITOR

Acta 14

Fecha: 23 Abril 2008

TEMAS TRATADOS:

- Hacer que al pinchar un menú en la ventana de SwingDemo, se vean en la otra ventana los estados y propiedades de estos.
- Arreglar fallo en Campo de texto, al poner centro a false.
- En la lista de cosas que tenemos para la memoria falta añadir el estudio de la aplicación que teníamos en Internet.
- Próximo componente a desarrollar: Pestañas.
  Para visualizar los cambios en la ventana de SwingDemo podemos colocar una pestaña con varias pestañas dentro.
- Próxima reunión: 7 Mayo

ACTA DE LA REUNIÓN DE SISTEMAS INFORMÁTICOS

PROYECTO SYNTH EDITOR

Acta 15

Fecha: 7 Mayo 2008

TEMAS TRATADOS:

- Sería interesante tener una ventana de prueba con todos los elementos y distintos Synth de prueba.
- Próximo componente a implementar: ComboBox o ScrollBar
- Próxima reunión: 21 Mayo

ACTA DE LA REUNIÓN DE SISTEMAS INFORMÁTICOS

PROYECTO SYNTH EDITOR

Acta 16

Fecha: 21 Mayo 2008

TEMAS TRATADOS:

- Al probar la aplicación ha dado un error en SWTView->ChangeCombo(), null pointer.
- Próximos componentes a implementar: Lista y PopUp Menu.
- Ver si podemos aprovechar los componentes que ya tenemos para hacer otros fácilmente.
- Nuevos apartados en la memoria:
    - Objetivos Concretos: lo que pretendemos hacer.
    - Conclusiones: Objetivo por objetivo ir diciendo si se ha cumplido o no y por qué.
    - Sección Bibliografía (entre conclusiones y apéndices).
- Próxima reunión: 4 Junio.

ACTA DE LA REUNIÓN DE SISTEMAS INFORMÁTICOS

PROYECTO SYNTH EDITOR

Acta 17

Fecha: 4 Junio 2008

TEMAS TRATADOS:

- En la pasada reunión convenimos en implementar el componente lista y popUp menú ya que el combo box estaba compuesto entre otros por estos dos, una vez implementados nos dimos cuenta de que la lista no funciona como esperábamos ya que si queremos cambiar por ejemplo el estado MOUSE_OVER, se cambia de toda la lista y no de uno de los componentes de la lista con lo que no le encontramos mucha utilidad, como conclusión es posible que descartemos este componente.

- Tenemos la documentación en diferentes documentos, debemos juntar todo en uno solo para ir dando formato a la memoria.

- Aumentar el documento de Gestión de Configuración, explicando cómo nos hemos organizado, cómo hemos trabajado con el CVS etc.

- Próxima reunión: Como ya no tenemos clase, cuando vayamos teniendo la memoria mandamos un e-mail y nos reunimos. Preferiblemente lunes y miércoles por la mañana ya que es cuando Gonzalo tiene tutorías.

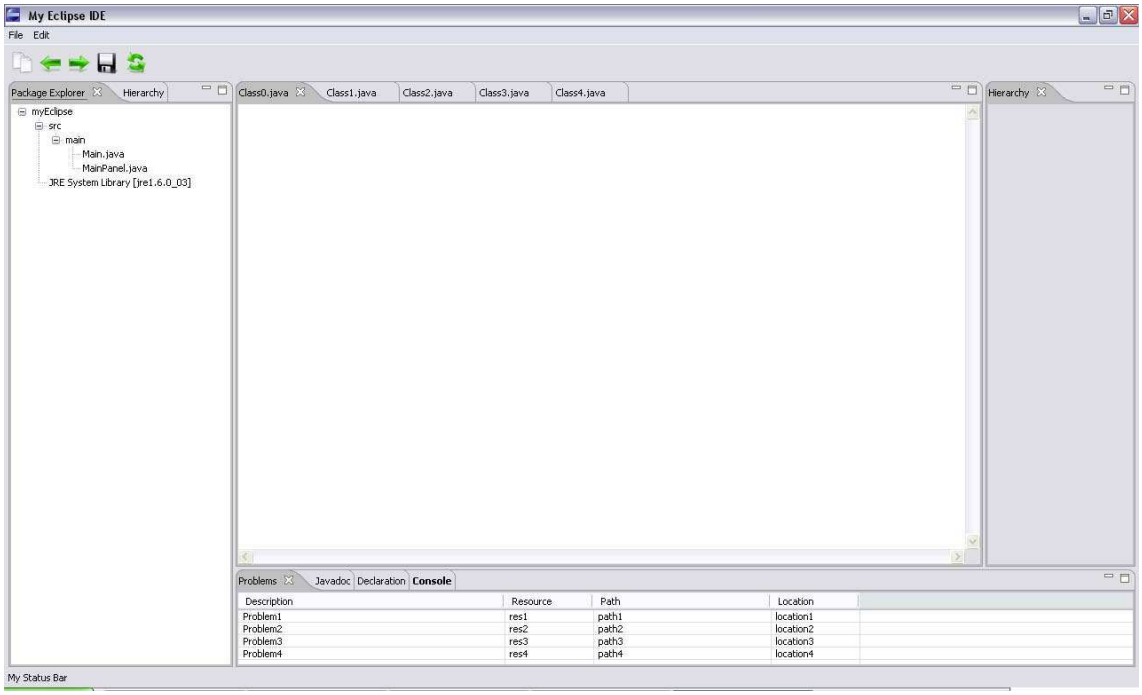
APÉNDICE C: Tutoriales

C.1. Tutorial de SWT: Creando un IDE similar a Eclipse

En este tutorial se va a crear una aplicación empleando la librería SWT. Esta librería permite la creación de interfaces de usuario (como Swing o AWT). Fue creada por IBM para la construcción de Eclipse, pero es suficientemente general como para poder emplearla para muchos tipos de proyectos.

Se advierte que la intención de este tutorial es la de familiarizarse con el API de la librería SWT. En ningún caso se pretende dar funcionalidad a la interfaz, ni hacer una interfaz igual que la de Eclipse.

El resultado final de la aplicación será el siguiente:



Este tutorial se va a realizar bajo eclipse. También necesitamos descargarnos la librería SWT de su página oficial:

http://www.eclipse.org/swt/

Una vez en ella, elige tu sistema operativo y descarga el archivo. En mi caso he descargado la versión 3.3 para Windows.

1. Creando un proyecto en Eclipse

Selecciona File -> new -> Project… y elige Java Project. Pulsa Next. Aparecerá el siguiente diálogo:

Llamaré al proyecto myEclipse. También elige la opción “Create separate source and output folders”. El resto de valores son por defecto.

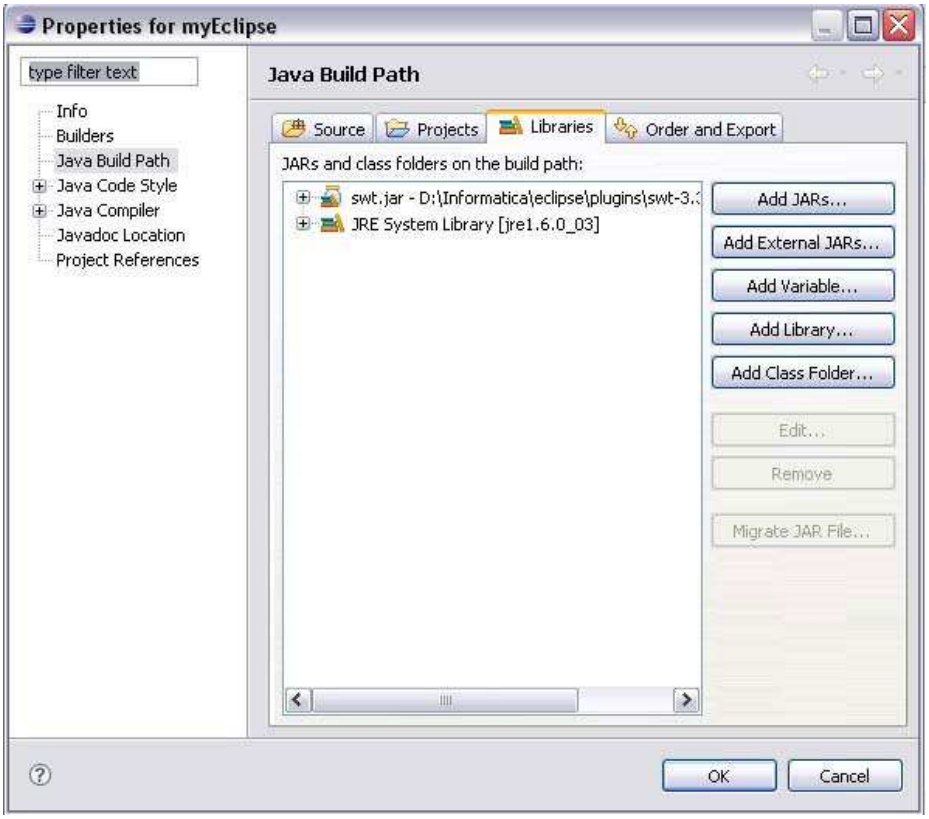
Pulsa Finish.

2. Configurando Eclipse para trabajar con SWT

Lo primero que hay que hacer es descomprimir el archivo descargado. Yo lo he descomprimido en la carpeta plugins de Eclipse.

Una vez descomprimido hacemos click con el botón derecho sobre nuestro proyecto y elegimos Build Path -> Configure Build Path…

Ahora añadimos la librería con la opción “add external jars…”. Seleccionamos swt.jar en la ruta en la que lo hayamos descomprimido. El resultado será el siguiente:



3. Creando nuestra primera aplicación en SWT

Creamos un nuevo paquete llamado “main” y la clase “Main”.

Para poder usar las clases SWT, importaremos los siguientes paquetes:

```
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
```

Ahora ya podemos comenzar a escribir el código de nuestra aplicación.

Puesto que SWT emplea las interfaces nativas de las sistemas operativos sobre los que corre la máquina virtual, se necesita una clase que realice la conexión. Esta clase es “Display”. Crearemos una instacia de dicha clase:

```
Display display = new Display();
```

Para crear una nueva ventana lo único que tenemos que hacer es usar la clase Shell (el equivalente a JFrame en Swing). Le pasaremos como argumento el display que utilizará nuestra aplicación.

```
Shell shell = new Shell(display);
```

Ahora le daremos un tamaño a nuestra aplicación mediante el método **setSize()** y lanzaremos la aplicación con **open()**.

Si ejecutas la aplicación, verás que aparece una ventana durante muy poco tiempo y desaparece de nuevo. Esto ocurre porque SWT necesita un bucle constantemente ejecutándose para funcionar. El bucle es el siguiente:

```
while (!shell.isDisposed()){
	if (!display.readAndDispatch()){
		display.sleep();
	}
}
display.dispose();
```

El bucle continúa iterando mientras que la ventana de nuestra aplicación no sea destruya (por ejemplo, cuando el usuario cierre la ventana). En este bucle lo que se hace es leer los eventos lanzados por el sistema operativo. Si no hay ninguno en la cola de eventos, el hilo que lanza nuestra aplicación "se duerme" para no consumir ciclos de CPU.

Cuando salimos del bucle, debemos destruir el display antes de terminar la ejecución de la aplicación.

Ahora ya podemos ejecutar nuestra aplicación. Le añadiremos un título a nuestra ventana con el método **setText()**.

```
shell.setText("My Eclipse IDE");
```

Nota: Todas las inicializaciones sobre el shell deben ir antes del bucle de ejecución, puesto que no se saldrá de él mientras se mantenga la ejecución, o bien en hilos que se ejecuten en paralelo.

También podemos añadirle una imagen a la ventana, mediante **setImage()**. Busca un icono que te guste y guárdalo en la carpeta del proyecto. Yo me he creado una carpeta images para guardar las imágenes:

```
shell.setImage(new Image(display,"images/icon.png"));
```

Para ello necesitamos incluir la librería: `import org.eclipse.swt.graphics.*;`

También vamos a maximizar por defecto la ventana, que es la forma más común de usar nuestro Eclipse. No sería por tanto necesaria el comando setSize(). Sólo es necesario el siguiente código:

```
shell.setMaximized(true);
```

El código final es el siguiente:

```
package main;

import org.eclipse.swt.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.widgets.*;

public class Main {

	static public void main(String[] args){

		//Manages the connection between the OS and our application
		Display display = new Display();

		//Main window of our application
		Shell shell = new Shell(display);

		//Initializing and launching the application
		shell.setMaximized(true);
		shell.open();
		shell.setText("My Eclipse IDE");
		shell.setImage(new Image(display,"images/icon.png"));

		/*
		 * The loop is executed while the main window is not
		 * disposed. The loop reads and dispatches events
		 * from the OS queue. When this queue is empty, the
		 * thread that runs the aplication sleeps.
		 */
		while (!shell.isDisposed()){
			if (!display.readAndDispatch()){
				display.sleep();
			}
		}
		display.dispose();
	}
}
```

Y aquí podemos ver el resultado de nuestra ventana inicial:

4. Los menús en SWT

Para nuestra aplicación, vamos a necesitar un menú. No reproduciremos todas las opciones de cada menú.

La idea de los menús es un poco diferente de la de Swing. En SWT, existen dos clases básicas para generar nuestros menús: **Menu** y **MenuItem**.

Los **MenuItem** son los elementos de nuestro menú, mientras que **Menu** es un contenedor de **MenuItems**.

Viéndolo de manera abstracta, nuestra barra de menús será de tipo **Menu**. Esta barra tendrá varios **MenuItems** como son “File”, “Edit”, “Source”, “Refactor”.

Centrándonos en “File”, necesitaremos añadir un **Menu** a dicho **MenuItem** para contener sus nuevos elementos del menú. “New” será un **MenuItem** que añadiremos al **Menu** de File. Este new a su vez contendrá un **Menu** que incluirá sus **MenuItem** correspondiente.

Para distinguir entre las distintas funcionalidades que pueden llevar a cabo los objetos de estas clases, SWT aporta unas constantes que deberán ser pasadas por parámetro.

Para los menús:

- SWT.BAR: indica que el menú es en particular la barra de menús.
- SWT.DROP_DOWN: un menú clásico dentro de la barra de menús.

Para los elementos de menú:

- SWT.PUSH: es un elemento seleccionable, sobre el que se puede hacer click para lanzar una acción.
- SWT.CASCADE: este elemento a su vez incluirá un submenú.
- SWT.CHECK: el elemento es un checkbox, que se puede marcar y desmarcar.
- SWT.RADIO: un radio button.
- SWT.SEPARATOR: permite añadir separadores (líneas horizontales) a nuestros menús.

Con estas ideas, ya podemos crear nuestra barra de menús. La crearemos en una nueva clase llamada MyMenu. El único atributo de la clase será el siguiente:

```
private Menu menuBar;
```

Este atributo es el que realmente contendrá el menú. En la constructora escribiremos la siguiente instrucción.

```
menuBar = new Menu(shell,SWT.BAR);
```

El menú necesita saber el shell en el que va a posicionarse. Se lo pasaremos como atributo a la constructora de clase. También le indicaremos que va a ser la barra de menús mediante el segundo parámetro.

Ahora le añadiremos el elemento “File”:

```
MenuItem file = new MenuItem(menuBar,SWT.CASCADE);
file.setText("File");
```

Con esto creamos un nuevo elemento que se incluirá en la barra de menú. Con el segundo parámetro le indicamos que este elemento contendrá un menú. Este menú lo tenemos que crear y añadírselo con el método **setMenu()**:

```
Menu fileMenu = new Menu (menuBar.getShell(),SWT.DROP_DOWN);
file.setMenu(fileMenu);
```

El primer parámetro vuelve a ser el Shell de la aplicación, que podemos obtenerlo de la barra de menús, y le indicamos que indicamos que es del tipo clásico, que se despliega hacia abajo.

Al menú “File” le añadimos un par de elementos:

```
MenuItem newItem = new MenuItem(fileMenu,SWT.CASCADE);
newItem.setText("New");

MenuItem openFile = new MenuItem(fileMenu,SWT.PUSH);
openFile.setText("Open file...");
```

El primero es el submenú New, que requiere el parámetro SWT.CASCADE. El segundo es el que nos permite abrir un archivo, que es un elemento final, sin submenús, lo que se indica con el parámetro SWT.PUSH.

Añadiremos también un separador con la instrucción:

```
new MenuItem(fileMenu, SWT.SEPARATOR);
```

Para crear el submenú New hay que seguir los mismos paso: crear un nuevo menú que se añadirá al elemento, y a este añadirle los nuevos elementos.

```
Menu newMenu = new Menu (menuBar.getShell(),SWT.DROP_DOWN);
newItem.setMenu(newMenu);

MenuItem project = new MenuItem(newMenu,SWT.PUSH);
project.setText("Project...");

MenuItem newClass = new MenuItem(newMenu,SWT.PUSH);
newClass.setText("Class");
```

Mediante este método puedes continuar creando toda la barra de menús. Para añadir el menú al Shell, en el método main añadiremos la siguiente instrucción:

```
shell.setMenuBar(menu.getMenuBar());
```

Al final conseguirás un resultado similar al de la imagen:

Si queremos darle funcionalidad a los menús, sólo tenemos que controlar un evento. Haremos la explicación sobre el elemento New -> Project…

```
project.addListener(SWT.Selection, new Listener(){

	public void handleEvent(Event e) {
		System.out.println("Creating a new project...");

	}
}
);
```

Primero se le indica con una constante con qué acción sobre el elemento se lanzará el evento. En este caso, cuando se seleccione. Además se crea un nuevo oyente en el que hay que sobrescribir el método **handleEvent().** En este caso sólo mostraremos un mensaje por consola.

5. La barra de herramientas

Al igual que en los menús, la barra de herramientas se crea a partir de dos clases: **ToolBar** y **ToolItem**. Nos creamos una clase nueva llamada MyToolbar con el siguiente atributo:

```
private ToolBar toolbar;
```

Y añadimos en el constructor el siguiente código:

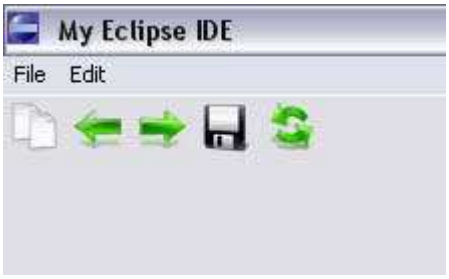
```
toolbar = new ToolBar(shell,SWT.HORIZONTAL);
toolbar.setSize(800,35);
```

Con ello añadimos la barra de herramientas (de posición horizontal) y le damos un tamaño inicial. También se añade ya al Shell principal. Para añadir un elemento hacemos lo siguiente:

```
Image img = new Image(toolbar.getDisplay(),path);
ToolItem item = new ToolItem(toolbar,SWT.PUSH);
item.setImage(img);
```

Cargamos una imagen (path es la ruta de la imagen, un String). Posteriormente creamos un ítem que añadimos a la barra de herramientas (PUSH porque no va a tener ningún menú desplegable) y le ponemos la imagen al ítem. En vez de imágenes, se puede añadir un texto (o ambos). Puedes añadir tantos botones como quieras con el método anterior.

La apariencia que nos queda es la siguiente:



Nota: Los iconos usados están tomados de http://www.vistaico.com.

Esta barra de herramientas es muy simple. La de Eclipse emplea otra clase llamada **CoolBar**, que permite integrar varias barras de herramientas que pueden moverse de posición y reordenarse del modo que el usuario quiera. Se deja a la voluntad del lector la investigación al respecto.

```
import org.eclipse.swt.graphics.Image;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.*;

/**
 * Generates and adds the toolbar to the shell
 */
public class MyToolbar {

	/**
	 * The toolbar
	 */
	private ToolBar toolbar;

	/**
	 * Class constructor
	 * @param shell The main shell
	 */
	public MyToolbar(Shell shell){

		//Creates and resizes the toolbar
		toolbar = new ToolBar(shell,SWT.HORIZONTAL);
		toolbar.setSize(800,35);

		//Adds the items to the toolbar
		addNewItem("images/PNG/Copy.png");
		addNewItem("images/PNG/Arrow-Right.png");
		addNewItem("images/PNG/Arrow-Left.png");
		addNewItem("images/PNG/Save.png");
		addNewItem("images/PNG/Refresh.png");
	}

	/**
	 * Add a new item to the toolbar
	 * @param path Image path to use in the item
	 */
	private void addNewItem(String path) {

		Image img = new Image(toolbar.getDisplay(),path);
		ToolItem item = new ToolItem(toolbar,SWT.PUSH);
		item.setImage(img);
```

```
        }
}
```

6. El layout principal

SWT dispone de cuatro layouts básicos diferentes:

- FillLayout: posiciona los elementos uno detrás de otro en posición horizontal o vertical, expandiéndolos de manera que ocupen todo el espacio de la ventana.
- RowLayout: permite la recolocación dinámica de los objetos en la ventana en forma de columnas. Esto es, si al principio todos los componentes caben en una fila y la ventana se redimensiona, los componentes se recolocan creándose nuevas filas si son necesarias.
- GridLayout: Los componentes se posicionan en una cuadrícula
- FormLayout: se emplea para crear formularios.

Si estuviéramos trabajando en Swing, nos sería muy útil el BorderLayout para colocar la barra de herramientas en el norte, la barra de estado en el sur y el resto en el centro. En SWT no existe ningún layout parecido, pero podremos simularlo gracias al GridLayout.

Nuestro grid estará formado por una sola columna. Sólo será necesario el siguiente código:

```
GridLayout layout = new GridLayout();
layout.numColumns = 1;
shell.setLayout(layout);
```

Ahora añadiremos a nuestro Shell dos etiquetas con borde que representarán nuestra zona central y nuestra barra de estado:

```
Label label1 = new Label(shell,SWT.BORDER);
label1.setText("Middle area");

Label label2 = new Label(shell,SWT.BORDER);
label2.setText("Status bar");
```

Ahora ejecuta tu aplicación. Obtendrás un resultado similar al siguiente:

No se parece mucho al resultado que andamos buscando. El problema es que el GridLayout mantiene el tamaño original de los objetos. Lo que en realidad querríamos es que la zona central ocupara todo el espacio vacío tanto vertical como horizontalmente, y también que la barra de estado ocupe todo el ancho de la ventana.

Para esto, SWT pone a nuestra disposición los **Layout Data**, que son clases que nos permiten personalizar estos parámetros. En particular, vamos a usar la clase **GridData**. Escribiremos el siguiente código:

```
GridData label1Data = new GridData(
      GridData.HORIZONTAL_ALIGN_FILL |
      GridData.VERTICAL_ALIGN_FILL  |
      GridData.GRAB_HORIZONTAL |
      GridData.GRAB_VERTICAL);
```

Con HORIZONTAL_ALIGN_FILL estamos indicando que la celda que ocupa la etiqueta debe llenar todo el espacio horizontal que quede libre. La constante GRAB_HORIZONTAL le indica a la propia etiqueta que ocupe todo el espacio horizontal que queda libre en su celda. La dirección vertical es análoga.

Ahora incluimos el gridData a la etiqueta:

```
label1.setLayoutData(label1Data);
```

Lo mismo haremos con label2, pero sólo en la dirección horizontal:

```
GridData label2Data = new GridData(
      GridData.HORIZONTAL_ALIGN_FILL |
      GridData.GRAB_HORIZONTAL);
label2.setLayoutData(label2Data);
```

Por fin hemos conseguido situar las etiquetas como queríamos. El resultado es el siguiente:

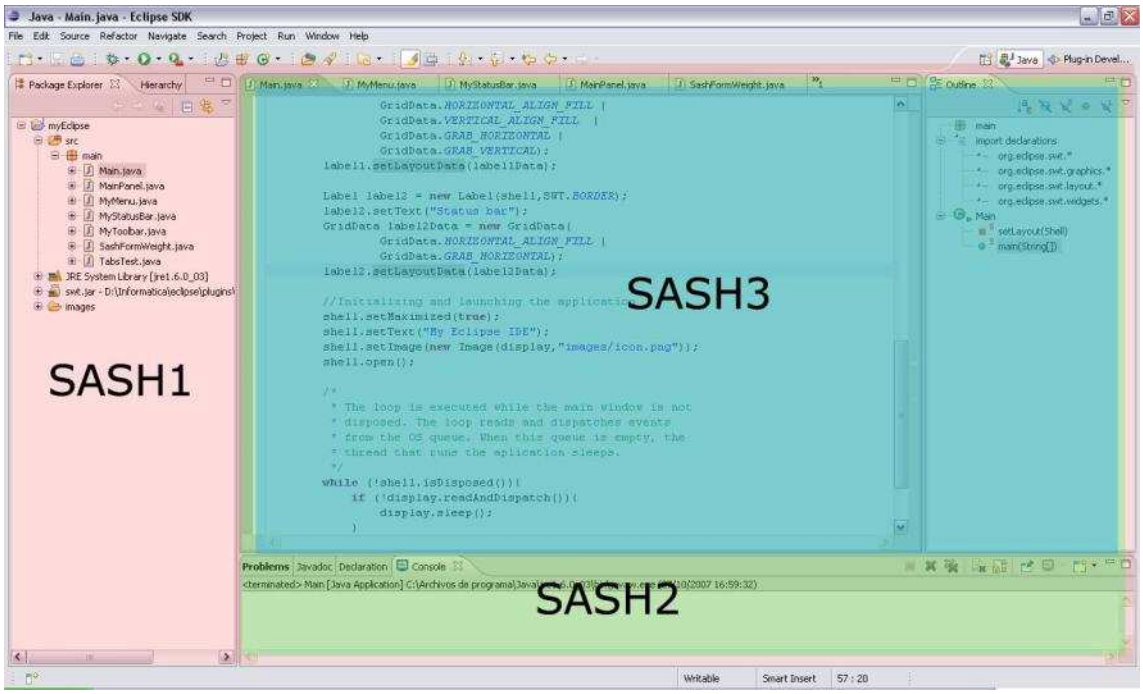
7. Área central: los Sash Forms

No entraremos en más detalles con respecto a la barra de estado, pero lo que sí es más importante es saber cómo podemos crear nuestro área central.

Como podemos obsevar, en eclipse aparecen una serie de paneles partidos, que pueden aumentar y disminuir de tamaño a gusto del usuario. Esto es lo que en AWT y Swing se conoce como SplitPane, que en SWT corresponde a **SashForm**.

Cada sash form está formado por dos paneles separados por una barra desplazable. Cada panel de estos puede a su vez contener otro sash form de manera recursiva. Si nos fijamos, la vista principal de Eclipse contiene cuatro paneles. De la manera en la que están colocados, no podemos conseguir los cuatro paneles con sólo dos sash forms.

Podemos realizar la siguiente descomposición:



Usaremos 3 sash forms. El sash1 será de tipo horizontal y contendrá en su parte izquierda el panel con la pestaña de projects y en la derecha el sash2. El sash2 será vertical y tendrá en la parte inferior el área de los problemas, la consola, etc y en la parte superior el sash3. Y éste último sash contendrá en su parte izquierda las áreas de texto y en la derecha el árbol de outline.

Creamos entonces el primer sash, que incluiremos en el Shell en el lugar donde antes teníamos la label1:

```
SashForm sash1 = new SashForm(shell,SWT.HORIZONTAL);
```

Ahora le añadimos una etiqueta que corresponderá a la zona de Projects:

```
Label projects = new Label(sash1,SWT.BORDER);
projects.setText("Projects");
```

Esta etiqueta se añade a la izquierda del sash1. En la derecha añadiremos un nuevo sash:

```
SashForm sash2 = new SashForm(sash1,SWT.VERTICAL);
```

En la parte superior del sash2 debemos añadir el sash3:

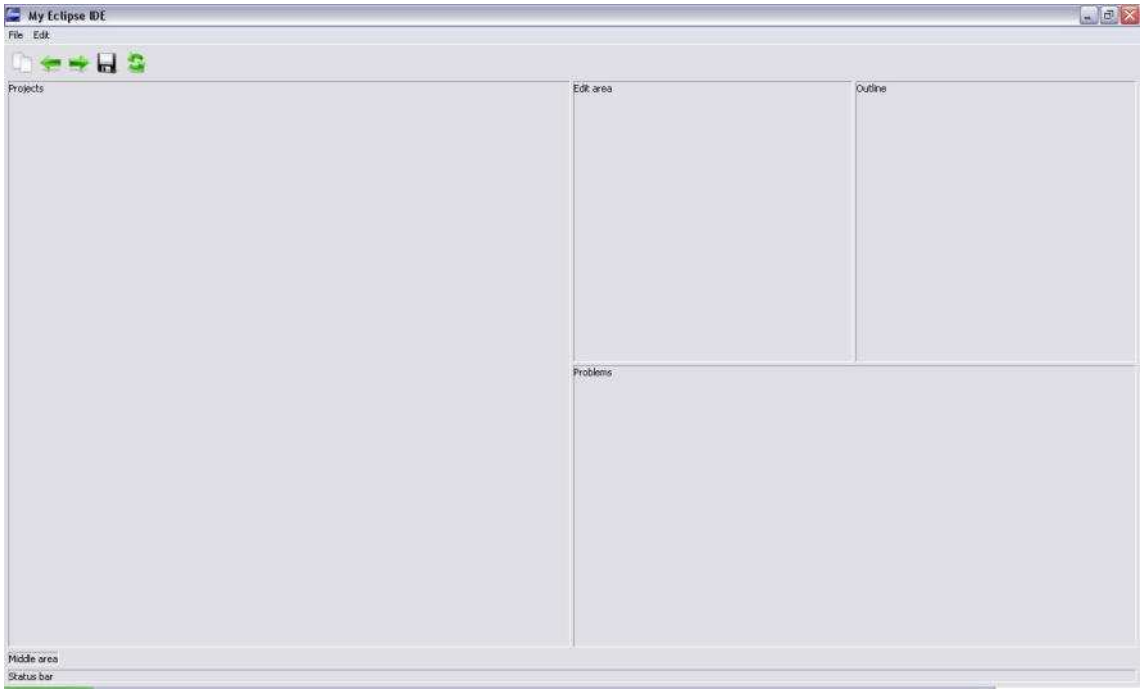
```
SashForm sash3 = new SashForm (sash2,SWT.HORIZONTAL);
```

Y ahora añadimos etiquetas en el resto de áreas. Recuerdo que estas etiquetas no son necesarias, sólo son útiles para hacernos una idea de cómo quedará la aplicación resultante:

```
Label problems = new Label(sash2,SWT.BORDER);
projects.setText("Problems");
Label editArea = new Label(sash3,SWT.BORDER);
editArea.setText("Edit area");
```

```
Label outline = new Label(sash3,SWT.BORDER);
outline.setText("Outline");
```

Acuérdate de usar el GridData del antiguo label1 para el sash1. El resultado será el siguiente:



Como ves, la barra divisoria aparece centrada en cada uno de los sash. Pero nosotros lo que queremos es dejar la mayor parte del espacio para la zona de edición. Para esto, existe un método de **SashForm** llamado **setWeights()** que recibe como parámetro de entrada un array de enteros. El número de enteros del array tiene que ser igual al número de elementos que hayamos insertado en el sash, en nuestro caso 2 en cada uno.

Estos enteros representan tamaños relativos. Si le pasamos el array {2,1}, obtendremos como resultado que el primer panel del sash será 2 veces más grande que el segundo. En nuestro caso, lo podemos configurar aproximadamente de la siguiente manera:

```
sash1.setWeights(new int[] {1,4});
sash2.setWeights(new int[] {5,1});
sash3.setWeights(new int[] {5,1});
```

Ahora ejecuta tu aplicación:

Mucho mejor, ¿verdad?

8. Pestañas

Ahora vamos a crear unas pestañas parecidas a las de Eclipse. Para ello necesitamos la clase **TabFolder**, que se corresponde con JTabbedPane en Swing. En realidad, la clase que usaremos se llama **CTabFolder**, para lo que necesitaremos importar la siguiente librería:

```
import org.eclipse.swt.custom.*;
```

Se trata de un conjunto de clases personalizadas que contienen características diferentes a las representaciones nativas del sistema operativo.

A diferencia de Swing, en SWT sí que existe una clase que representa una pestaña, el **TaBItem**. En nuestro caso usaremos **CTabItem**.

Sustituiremos la etiqueta Projects por dos pestañas: Package Explorer y Hierarchy.

Creamos nuestro TabFolder :

```
CTabFolder tabFolder = new CTabFolder(sash1,SWT.NONE);
```

Y añadimos las dos pestañas:

```
CTabItem pExplorer = new CTabItem(tabFolder, SWT.NONE);
pExplorer.setText("Package Explorer");

CTabItem hierarchy = new CTabItem(tabFolder, SWT.NONE);
hierarchy.setText("Hierarchy");
```

Comprueba el resultado:

Nuestras dos pestañas están creadas, pero aún son muy diferentes de las de eclipse. Por ejemplo, no tienen la cruz para poder cerrarlas, ni tienen forma redondeada, ni aparece un borde alrededor, ni los botones de minimizar y maximizar del tab folder. Todas estas opciones son muy fáciles de especificar.

Al crear el tab folder, en lugar de utilizar la etiqueta SWT.NONE, escribiremos SWT.BORDER, así conseguiremos el borde de alrededor.

```
CTabFolder tabFolder = new CTabFolder(sash1,SWT.BORDER);
```

Permitir la opción de cerrar también es muy fácil. Al crear las pestañas, añadiremos la opción SWT.CLOSE:

```
CTabItem pExplorer = new CTabItem(tabFolder, SWT.CLOSE);
```

Si te fijas, este tab folder en eclipse sólo te permite cerrar una pestaña si está seleccionada. Para conseguirlo sólo necesitamos llamar al siguiente método:

```
tabFolder.setUnselectedCloseVisible(false);
```

Para ver las pestañas redondeadas, escribiremos lo siguiente:

```
tabFolder.setSimple(false);
```

Y para que nos aparezcan los botones de maximizar y minimizar:

```
tabFolder.setMinimizeVisible(true);
tabFolder.setMaximizeVisible(true);
```

Ya solo nos queda hacer las pestañas un poco más altas,

```
tabFolder.setTabHeight(22);
```

Y cambiar el color de la pestaña seleccionada a gris:

```
tabFolder.setSelectionBackground(tabFolder.getDisplay().getSystemColor
(SWT.COLOR_GRAY));
```

Si quieres añadir una imagen a la etiqueta, sólo tienes que llamar al método **setImage()** de **TabItem**. Comprueba el resultado:

Con el mismo método puedes construir el resto de paneles.

Es recomendable que al añadir varios tab folders en un proyecto, añadas la siguiente instrucción para cada uno:

```
tabFolder.setSelection(0);
```

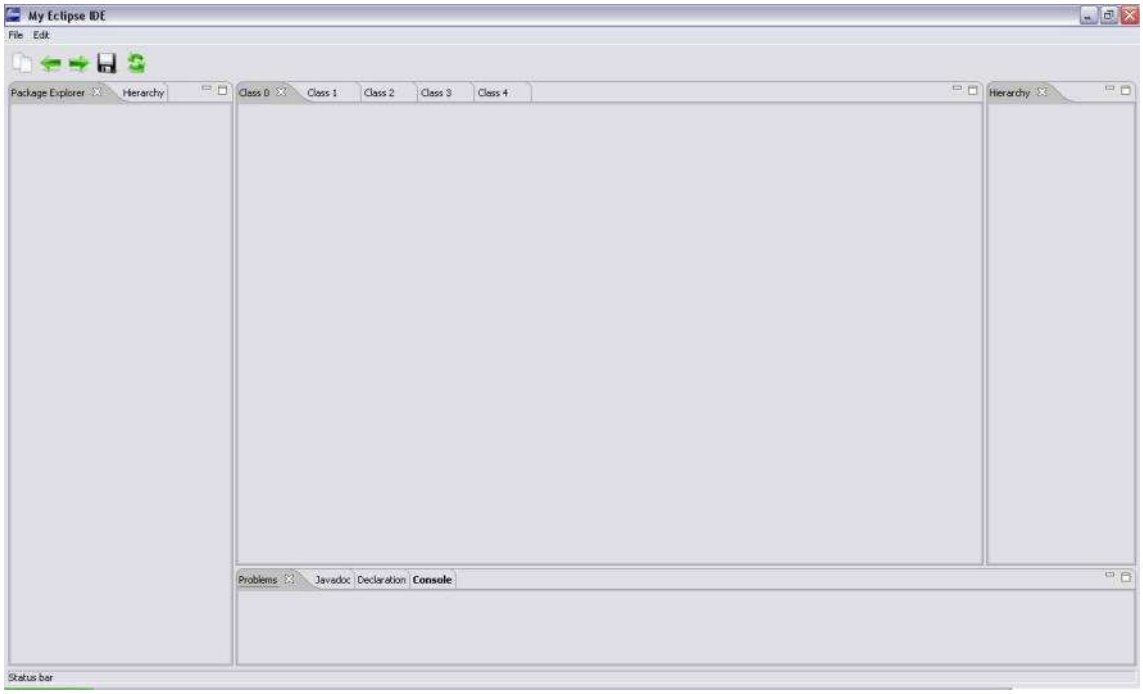
De esta manera conseguirás que aparezca seleccionada desde el principio la pestaña primera seleccionada. De lo contrario aparecerán todas deseleccionadas y el efecto es menos vistoso.

Si quieres cambiar la fuente a negrita en una pestaña, como por ejemplo en Console, necesitas acceder a la fuente. La clase **Font** contiene la fuente y una lista de **FontData**, que es información sobre la fuente, entre otras cosas el estilo de la misma. En Windows esta lista sólo devuelve un elemento.

Una manera de modificar la fuente consiste en obtener el FontData de la fuente de la pestaña, modificarlo y crear una nueva fuente con él, que es la que se asignará a la pestaña:

```
FontData fontData = console.getFont().getFontData()[0];
fontData.setStyle(SWT.BOLD);

console.setFont(new Font(console.getDisplay(), fontData));
```

El resultado hasta el momento es el siguiente:



9. Áreas de texto

Para los documentos que se abren en la zona central, crearemos un área de texto multilínea y que tenga barras de desplazamiento tanto vertical como horizontal. La instanciación de la pestaña será de la siguiente forma:

```
CTabItem item = new CTabItem(tabFolder, SWT.CLOSE);
item.setText("Class.java");
```

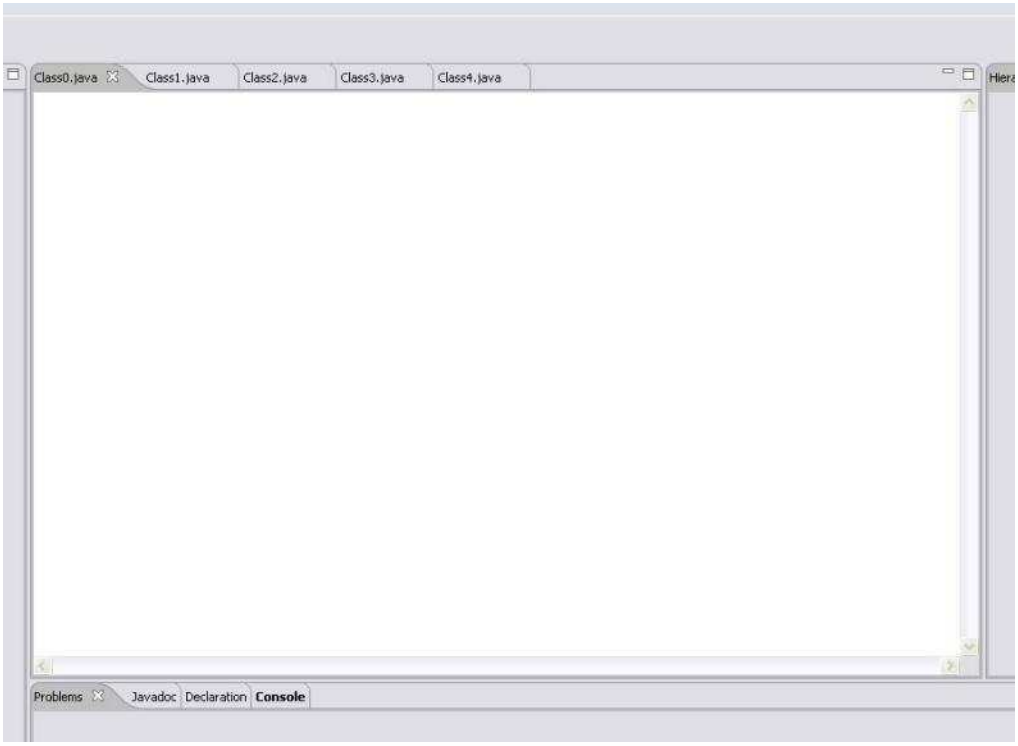
Creamos una nueva area de texto:

```
Text text = new Text(tabFolder, SWT.MULTI | SWT.V_SCROLL |
SWT.H_SCROLL);
```

SWT.MULTI indica que el texto será de más de una línea, y V_SCROLL y H_SCROLL activan las barras de desplazamiento vertical y horizontal respectivamente.

Para añadirlo a la pestaña sólo tenemos que usar el método **setControl()**:

```
item.setControl(text);
```



10. Tablas

Vamos a crear una tabla para la pestaña de Problems. Las tablas se crean a partir de tres clases: **Table**, **TableColumn** y **TableItem**.

La **Table** es el contenedor principal de la estructura. Su instanciación es de la siguiente manera:

```
Table table = new Table (tabFolder, SWT.FULL_SELECTION);
```

Así creamos una tabla en la que al seleccionar un elemento, aparece seleccionada la fila entera.

Ahora le añadiremos las columnas, creando objetos de la clase **TableColumn**:

```
TableColumn description = new TableColumn (table, SWT.LEFT);
description.setText("Description");
description.setWidth(300);

TableColumn resource = new TableColumn (table, SWT.LEFT);
resource.setText("Resource");
resource.setWidth(100);

TableColumn path = new TableColumn (table, SWT.LEFT);
path.setText("Path");
path.setWidth(200);
```

```
TableColumn location = new TableColumn (table, SWT.LEFT);
location.setText("Location");
location.setWidth(100);
```

En la declaración le indicamos que queremos alineación izquierda, el texto de la cabecera y un ancho para la columna. Si queremos que la columna sea tan ancha como la palabra de la cabecera sustituiremos **setWidth()** por **pack()**.

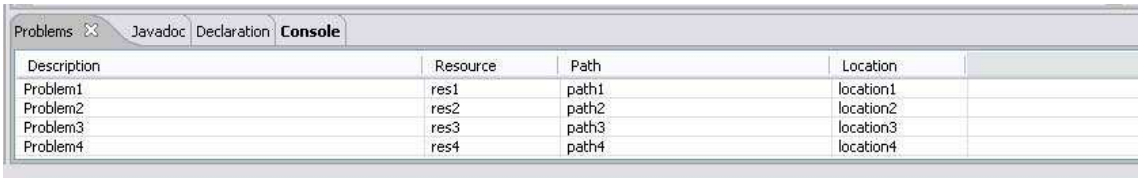
Ahora ya sólo nos queda añadir elementos a la tabla. Esto se hace de una manera muy fácil con la clase **TableItem**:

```
TableItem item1 = new TableItem(table, SWT.NONE);
item1.setText(new     String[]     {"Problem1",     "res1",     "path1",
"location1"});
```

Crea varios items para ver mejor la estructura.

Nuestra tabla ya está casi lista, pero aún no aparecen ni las cabeceras ni las líneas propias de una tabla. Es sólo cuestión de escribir el siguiente código:

```
table.setLinesVisible(true);
table.setHeaderVisible(true);
```



Como siempre, puedes añadir imágenes a cualquier celda de la tabla con el método **setImage()** de la clase **TableItem**, y a las cabeceras con el correspondiente de **TableColumn**.

11. Árboles

Por último, crearemos un árbol para la pestaña Package Explorer.

Para los árboles, necesitamos las clases **Tree** y **TreeItem**.

Crearemos primero el árbol de selección múltiple con la siguiente instrucción:

```
Tree tree = new Tree (tabFolder, SWT.MULTI);
```

Ahora le añadimos un hijo:

```
TreeItem myEclipse = new TreeItem (tree, SWT.NONE);
myEclipse.setText("myEclipse");
```

Para añadir hijos a myEclipse, sólo tenemos que pasarlo como parámetro en la constructora de **TreeItem**:

```
TreeItem src = new TreeItem (myEclipse, SWT.NONE);
src.setText("src");
```

Así podemos ir añadiendo ramas e hijos al árbol de manera muy sencilla. Crearemos los siguientes hijos para tener un árbol un poco mayor:

```
TreeItem main = new TreeItem (src, SWT.NONE);
main.setText("main");

TreeItem mainClass = new TreeItem (main, SWT.NONE);
mainClass.setText("Main.java");

TreeItem mainPanel = new TreeItem (main, SWT.NONE);
mainPanel.setText("MainPanel.java");

TreeItem jre = new TreeItem (myEclipse, SWT.NONE);
jre.setText("JRE System Library [jre1.6.0_03]");
```

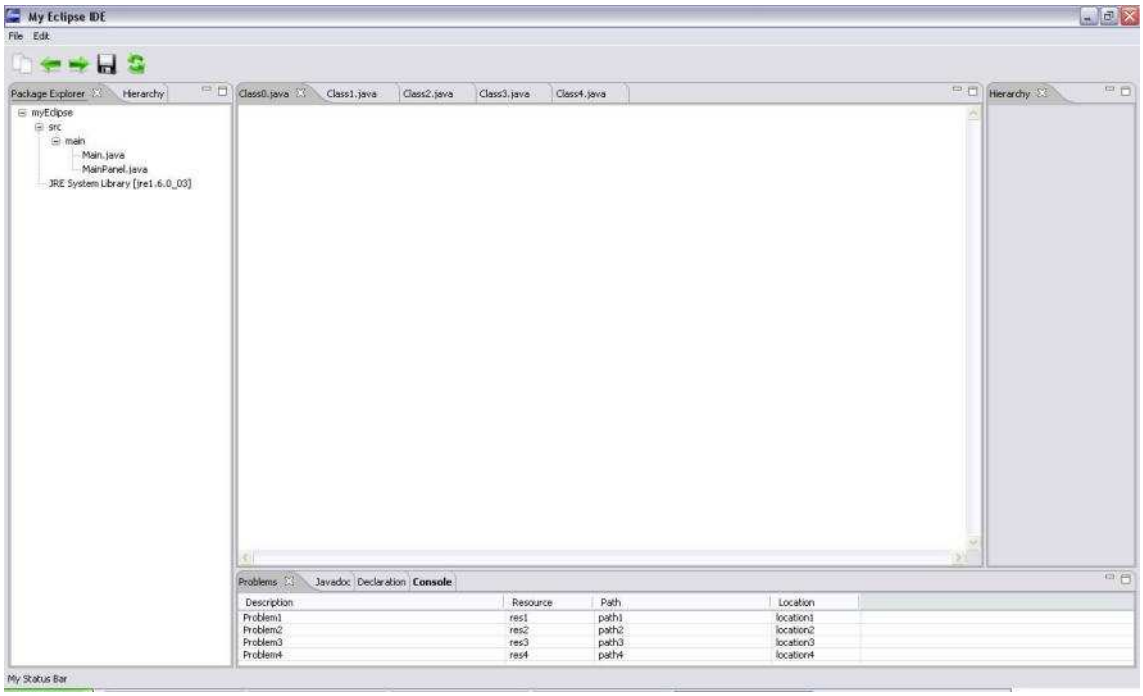
Si ejecutas ahora, verás que el árbol está completamente contraído. Es más útil en Eclipse tener el árbol expandido desde un principio. Para ello puedes emplear el método **setExpanded()**. Úsalo al menos en cada nodo que no sea hoja para expandir el árbol completo.

```
myEclipse.setExpanded(true);
src.setExpanded(true);
main.setExpanded(true);
```

Debes tener en cuenta que el método setExpanded() debe ser usado tras haber incluido todos los hijos en la rama. Si no, esta sentencia no surtirá ningún efecto. Para no tener problemas, expande el árbol cuando termines de generarlo.

No olvides tampoco que puedes insertar imágenes en cualquier **TreeItem** con el método **setImage()**.

Este es el resultado final:



12. Conclusión

Aquí termina nuestro tutorial sobre SWT. Aún quedan muchos detalles para hacer una interfaz similar a Eclipse, pero gracias a este tutorial ya conoces la mayoría de las herramientas básicas para crear tu propia aplicación en SWT.

Como hemos podido comprobar, aunque SWT es un poco menos flexible que Swing, permite generar elementos con mucha mayor facilidad, sobre todo los tab folders, árboles y tablas. El código es mucho más limpio y más fácil de entender, y sobre todo no es necesario el uso del cell renderers para personalizar árboles, tablas, etc. Añadir imágenes a los componentes se vuelve mucho más cómodo.

Esperamos que este tutorial te haya ayudado a comprender el funcionamiento de SWT de una manera fácil y comprensible.

Podrás encontrar el código completo de este ejemplo en el área de descargas.

C.2. ¿Cómo integrar Swing en una aplicación SWT?

En nuestro intento por crear un plugin para Eclipse, nos ha surgido un pequeño problema. Eclipse está creado mediante la librería SWT, mientras que nuestro plugin pretende modificar el look and feel de una aplicación Swing.

Puesto que los plugins han de funcionar en Eclipse, necesitamos encontrar una manera de poder añadir a un componente SWT otro componente Swing.

Para ello, las últimas versiones de SWT incluyen una clase llamada **SWT_AWT**. Como podéis imaginar por el nombre, no vamos a poder mostrar ningún elemento Swing sin tener algún componente AWT por medio.

¿Cuál es la idea? Básicamente crear un panel AWT en el que incluiremos todos los elementos Swing que queramos. Este panel será añadido a un componente SWT (ahora veremos cuál), que mediante el método **new_Frame** de la clase **SWT_AWT** se integrará sin mayores problemas en nuestra aplicación.

Vamos a crear una pequeña aplicación que nos permitirá mostrar un mensaje por consola a partir de un botón y un campo de texto en Swing.

Primero vamos a crear la ventana en SWT. Si tienes alguna duda de cómo configurar tu Eclipse para trabajar en SWT, visita nuestro “Tutorial SWT” en la sección de tutoriales:

```
final Display display = new Display();
final Shell shell = new Shell(display);
shell.setSize(500,400);
shell.setText("Integrating Swing and SWT");
shell.setLayout(new FillLayout());
```

De esta forma hemos creado la ventana con un título y un tamaño determinados, y le hemos puesto un Fill Layout. Ahora creamos un elemento de la clase **Composite** de SWT, que será quien contenga nuestros componentes Swing.

```
Composite item = new Composite(shell, SWT.EMBEDDED);
```

Utilizaremos la etiqueta SWT.EMBEDDED para indicar que en este componente vamos a incrustar elementos AWT.

Con nuestro componente especial, ahora tenemos que llamar al siguiente método:

```
Frame frame = SWT_AWT.new_Frame(item);
```

El método **new_Frame()** del que hablábamos antes necesita como parámetro de entrada el componente especial que creamos. Este método estático nos devuelve un **Frame** AWT donde insertaremos nuestros elementos Swing.

Creamos ahora nuestros componentes Swing y los añadimos al frame:

```
Panel panel = new Panel();
frame.add(panel);

final JTextField text = new JTextField(20);
final JButton button = new JButton("Change title");

panel.add(text);
panel.add(button);
```

El panel debe ser de la librería AWT porque si no algunos componentes no funcionan correctamente si los incluimos en un JPanel. Parece ser algún problema con la librería SWT.

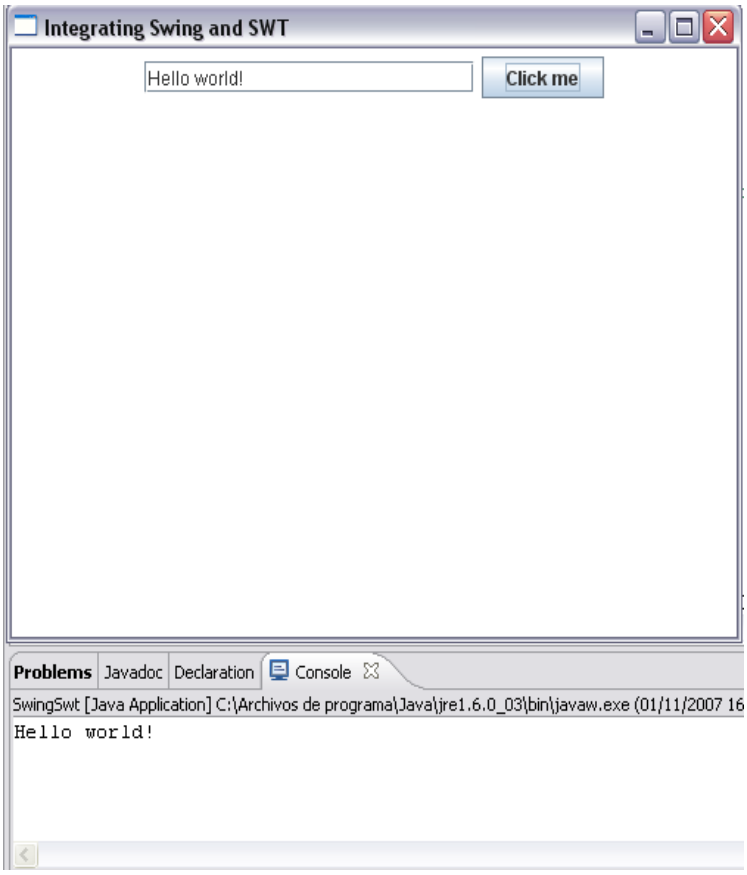
Añade un oyente al botón que al pulsarlo muestre por consola el mensaje del campo de texto:

```
button.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent arg0) {
            System.out.println(text.getText());
            }
      }
);
```

Lo único que queda por hacer es escribir el bucle de ejecución de los programas SWT:

```
shell.open();
while (!shell.isDisposed()){
      if (!display.readAndDispatch()){
            display.sleep();
      }
}
display.dispose();
```

Ya tendrás tu aplicación perfectamente funcionando. Ya has visto que necesitamos al menos un frame y un panel de AWT para insertar los elementos Swing.



C.3. The Synth Series vol. 1 – Introducción al Synth Look & Feel

Puesto que nuestra aplicación se basa en esta clase para generar el look and feel personalizado, vamos a crear una serie de tutoriales en torno a ella.

De esta forma podrás adquirir una base al respecto, y decidir llegado el momento si prefieres usar nuestra aplicación o realizar el trabajo a mano. Además, una herramienta siempre es mucho más fácil de utilizar si entiendes cómo funciona por debajo.

SynthLookAndFeel apareció por primera vez en la versión 5.0 de java con la idea de solucionar el problema que supone el enorme trabajo para realizar una interfaz personalizada en java.

SynthLookAndFeel es inicialmente una apariencia en blanco. Si intentas ejecutar una aplicación con este look and feel sin más, no verás nada. Esto es debido a que no contiene parámetros por defecto, lo que quiere decir que tendrás que dar apariencia a todos los componentes Swing uno a uno (o al menos a todos los que vaya a usar tu aplicación).

La ventaja de crear tus interfaces personalizadas con Synth look and feel en lugar de sobrescribir todos los métodos y clases asociadas de **BasicLookAndFeel** es que toda la apariencia queda definida en un archivo XML, independiente del código de tu aplicación. Esto añade muchas más ventajas a la opción anterior:

- El diseño de la apariencia puede ser delegado a diseñadores, que no tienen por qué saber programar (y en el caso de utilizar nuestra aplicación, no necesitan ni tan siquiera conocer el formato de XML necesario).
- Sólo programadores muy expertos son capaces de sobrescribir todos los métodos requeridos para hacer una interfaz personalizada, y además requiere mucho tiempo.
- Se puede cambiar de apariencia dinámicamente, sin necesidad de tocar código, lo que permite el uso de skins personalizadas.

¿Cómo establecer SynthLookAndFeel como el look and feel de tu aplicación?

Hay varias formas de establecer el SynthLookAndFeel. Para mí, la más fácil de entender es esta:

```
try {
            SynthLookAndFeel synth = new SynthLookAndFeel();
            File xml = new File("synth.xml");
            synth.load(new URL("file:///" + xml.getAbsolutePath()));
            UIManager.setLookAndFeel(synth);

      } catch (ParseException e) {
            System.out.println("Parsing error: Synth XML malformed");

      } catch (UnsupportedLookAndFeelException e) {
            System.out.println("This look and feel is not supported in
            your OS");

      } catch (MalformedURLException e) {
            System.out.println("The URL you gave for the xml is not
            correct");
```

```
} catch (IOException e) {
      e.printStackTrace();

}
```

Se crea una instancia SynthLookAndFeel. Después, se crea un instancia de la clase File para el synth.xml, con el objetivo de conseguir su ruta absoluta para pasarla como parámetro al cargar el synth. Como se requiere una URL en el método load, ésta se crea. Hay que recordar que una URL es por defecto una dirección web, y por tanto para que reconozca que es un archivo del propio equipo hay que indicárselo con `file:///`. Finalmente, sólo hay que establecer el look and feel en la aplicación. Hay que capturar cuatro excepciones:

- *ParseException:* es lanzada en caso de que el xml no esté bien formado. El método **load()** es el que obliga a usarla.
- *UnsupportedLookAndFeelException:* Es lanzada por el método **setLookAndFeel()** de UIManager. En el caso de Synth no tiene mucho sentido porque es soportado en todos los sistemas operativos.
- *MalformedURLException:* por si la URL que se le pasa al constructor no está bien formada.
- *IOException:* Al crear la instancia de File.

Aunque todos los enlaces de interés pueden ser encontrados en nuestra sección “enlaces”, puedes acceder al API de **SynthLookAndFeel** desde aquí:

http://java.sun.com/javase/6/docs/api/javax/swing/plaf/synth/SynthLookAndFeel.html

¿Cuál es el formato del archivo XML?

Podrás encontrar una definición más formal (la DTD del XML) en este enlace:

http://java.sun.com/j2se/1.5.0/docs/api/javax/swing/plaf/synth/doc-files/synthFileFormat.html

El formato básico consiste en un conjunto de etiquetas <style>,de la siguiente forma:

```
<style id= “identificador”>
      //Descripción de colores, fuentes, estados, etc
</style>
<bind style=”identificador” type=”región” key= “nombre_region”/>
```

Se crea un estilo que contiene cada uno de los valores de los elementos a personalizar. Después se aplica dicho estilo a una región, que puede ser un botón, un campo de texto, etc.

Todos los nombres de región están determinados por las constantes de la clase **Region**. Puedes consultar el API aquí:

http://today.java.net/download/jdk6/doc/api/javax/swing/plaf/synth/Region.html

Debes utilizar el nombre que viene en la descripción, no el de la constante. Por ejemplo, si quieres asignar una apariencia a al separador de un menú emergente, debes usar como key “PopupMenuSeparator” y no “POPUP_MENU_SEPARATOR”.

Si lo que quieres es dar un estilo igual para todas las regiones sólo tienes que poner key=".*".

Otra etiqueta importante es <state>, que permite definir distintas apariencias dependiendo del estado del componente: activado (ENABLED), con el ratón sobre él (MOUSE_OVER), pulsado (PRESSED), desactivado(DISABLED), con el foco (FOCUSED), seleccionado (SELECTED) y por defecto (DEFAULT).

```
<state value = "SELECTED AND PRESSED">
      //Personalización cuando el componente está seleccionado o
pulsado
</state>
```

Luego existen etiquetas para el color (la etiqueta <color>), para modificar la fuente (<font>), cada una con sus atributos personalizables.

Todas estas etiquetas las iremos viendo sobre la marcha en próximos tutoriales, ya que es más fácil comprenderlas sobre ejemplos reales.

Por último, presentaremos la etiqueta <property>. Esta etiqueta sirve para personalizar elementos propios de cada una de las posibles regiones, pudiendo así por ejemplo insertar una imagen para los checkboxes cuando están seleccionados o deseleccionados.

Existe una lista de unos 100 elementos, que puedes consultar aquí:

http://today.java.net/download/jdk6/doc/api/javax/swing/plaf/synth/doc-files/componentProperties.html

Como puedes ver, SynthLookAndFeel es una herramienta muy potente, pero a la vez un poco tediosa de utilizar, ya que el XML que hay que generar es bastante complejo y grande. En próximos tutoriales te enseñaremos cómo personalizar distintos componentes, de manera que puedas ir comprendiendo mejor la estructura y el funcionamiento del XML.

No te olvides de visitar la sección de enlaces para descubrir otras páginas en las que se habla de este tema y que te pueden ser de mucha utilidad.

C.4. The Synth Series vol. 2 – Personalizar un botón

En este tutorial vamos a personalizar uno de los componentes más simples de Swing, un **JButton**, para profundizar en el estudio de Synth.

Lo primero es crear nuestra interfaz en Swing, que constará únicamente de un **JFrame** y un **JButton**, de la siguiente manera:

```
JFrame frame = new JFrame("My Synth Test");
frame.setSize(200,100);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setLayout(new FlowLayout());

frame.add(new JButton("Test button"));
frame.setVisible(true);
```

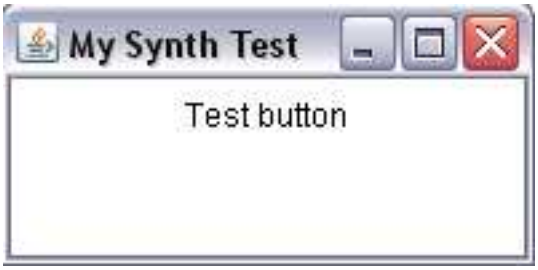
Creamos con ello una ventana con su título, su tamaño, su operación al cerrar la ventana, un layout, y le añadimos un botón. Por último lo mostramos. Si ejecutas verás lo siguiente:

Ahora vamos a cambiarle el look and feel a nuestra aplicación. Necesitamos asignar un SynthLookAndFeel. Inserta mejor este código antes de crear el frame:

```
try {
      SynthLookAndFeel synth = new SynthLookAndFeel();
      File xml = new File("synth.xml");
      synth.load(new URL("file:///" + xml.getAbsolutePath()));
      UIManager.setLookAndFeel(synth);

} catch (ParseException e) {
      System.out.println("Parsing error: Synth XML malformed");

} catch (UnsupportedLookAndFeelException e) {
      System.out.println("This look and feel is not supported in
      your OS");

} catch (MalformedURLException e) {
      System.out.println("The URL you gave for the xml is not
      correct");

} catch (IOException e) {
      e.printStackTrace();

}
```

Parece complicado, pero casi todo es captura de excepciones. Si no recuerdas algo, échale un vistazo al tutorial anterior de Synth. Crea un archivo synth.xml (con el bloc de notas, por ejemplo) en la carpeta principal del proyecto.

Ejecuta tu aplicación de nuevo:



Como puedes ver, el botón no tiene ninguna apariencia. Sólo somos capaces de ver las letras del mismo.

Ahora empezaremos a escribir nuestro xml. Puedes crearlo en el bloc de notas o usar cualquier herramienta de edición de xml que prefieras.

Recuerda añadir a tu xml las etiquetas <synth> y </synth> al principio y al final respectivamente.

Escribiremos un nuevo estilo para nuestro botón. Para ellos necesitamos añadir las etiquetas <style id=”button> y </style>. Dentro aplicaremos nuestra personalización al botón.

Primero añadiremos una propiedad muy útil que hará que el texto de nuestro botón baje un pixel cuando lo pulsemos, dando efecto tridimensional:

```
<property key="Button.textShiftOffset" type="integer" value="1"/>
```

Con la siguiente etiqueta,indicamos cuántos pixels debe dejar entre la letra y el borde de nuestro botón por cada uno de los cuatro lados:

```
<insets top="8" left="12" bottom="8" right="12"/>
```

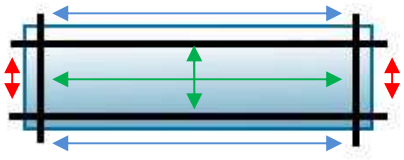
Después con <state> crearemos una apariencia general para nuestro botón, la que tendrá el botón por defecto. Voy a usar la siguiente imagen como fondo de botón:

Para ello, dentro de <state> escribe lo siguiente:

```
<imagePainter method="buttonBackground" path="images/button.jpg"
sourceInsets="10 10 10 10" />
```

Le estamos indicando que utilice esta imagen como fondo de nuestro botón (el path es la ruta relativa en la que está guardada la imagen).

El sourceInsets es un poco más difícil de explicar. Con ello le estamos indicando que coloque una especie de divisiones en la imagen de la siguiente forma:



Estas divisiones están determinadas por el número de píxeles indicados en el sourceInsets. Cuando un botón tenga un tamaño distinto al de nuestra imagen (que será casi siempre), la imagen se encogerá o estirará de la siguiente manera: las esquinas no cambian de tamaño, la parte superior e inferior sólo se estira a lo ancho, la izquierda y derecha a lo alto, y el centro en todas direcciones.

En nuestro caso, al ser las esquinas cuadradas, nos bastan con no estirar los bordes. 10 píxeles serán más que suficiente.

Ahora cambiamos la fuente:

```
<font name="Dialog" size="12" style="BOLD"/>
<color type="TEXT_FOREGROUND" value="#000000"/>
```

Con ello, le añadimos una fuente tipo Dialog, de tamaño 12 y negrita, y además al color del texto le asignamos el negro. En value puedes usar como nombre las constantes de Swing para los colores en lugar de su valor hexadecimal.

No te olvides de asignar el estilo a la región con la etiqueta <bind>:

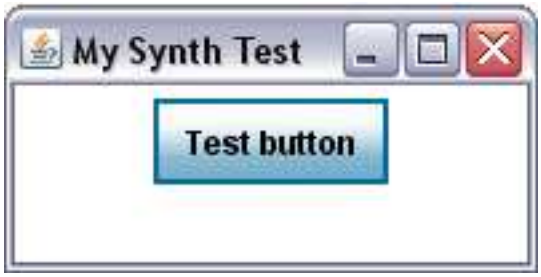
```
<bind style="button" type="region" key="Button"/>
```

Aquí tienes el código completo de lo hecho hasta ahora:

```
<synth>

	<style id="button">
		<!-- Shift the text one pixel when pressed -->
		<property key="Button.textShiftOffset" type="integer"
		value="1"/>

		<!-- set size of buttons -->
		<insets top="8" left="12" bottom="8" right="12"/>

		<state>
			<imagePainter method="buttonBackground"
			path="images/button.jpg"
			 sourceInsets="10 10 10 10" />

			<font name="Dialog" size="12" style="BOLD"/>
			<color type="TEXT_FOREGROUND" value="#000000"/>
		</state>

	</style>
	<bind style="button" type="region" key="Button"/>
```

```
</synth>
```

Ejecuta tu aplicación. Verás lo siguiente:

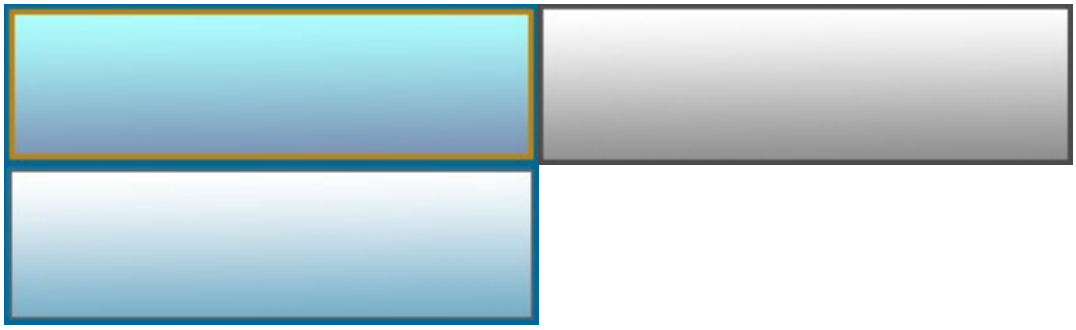


Ahora le vamos a añadir distintas imágenes en función del estado del botón. Para cuando el ratón está sobre el botón, añadimos una imagen nueva:

```
<state value = "MOUSE_OVER">
      <imagePainter method="buttonBackground"
      path="images/button_over.jpg" sourceInsets="10 10 10 10" />
</state>
```

La imagen es:

Y estas imágenes para cuando está pulsado, desactivado y con el foco:



El texto que he generado es el siguiente:

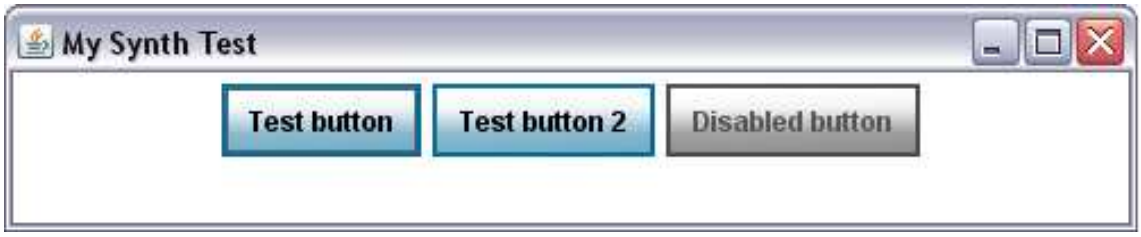
```
<state value = "MOUSE_OVER AND FOCUSED">
      <imagePainter method="buttonBackground"
      path="images/button_over.jpg" sourceInsets="10 10 10 10" />
</state>

<state value = "FOCUSED">
      <imagePainter method="buttonBackground"
      path="images/button_focused.jpg" sourceInsets="10 10 10 10" />
</state>

<state value = "FOCUSED AND PRESSED">
```

```
      <imagePainter method="buttonBackground"
      path="images/button_pressed.jpg" sourceInsets="10 10 10 10" />
      <color type="TEXT_FOREGROUND" value="#555555"/>
</state>

<state value = "DISABLED">
      <imagePainter method="buttonBackground"
      path="images/button_disabled.jpg"
      sourceInsets="10 10 10 10" />
      <color type="TEXT_FOREGROUND" value="#555555"/>
</state>
```

A algunos estados les he cambiado el color de la fuente. También ha sido necesario un estado especial, MOUSE_OVER AND FOCUSED, porque si no al estar el foco en un botón, si pones el ratón sobre él, no aparece la imagen de MOUSE_OVER. También hay que añadir FOCUSED para cuando se pulsa el botón, obteniendo FOCUSED AND PRESSED para que aparezca el fondo correcto.

Para probar el efecto del foco, inserta dos botones, e inserta uno más desactivado. Pulsa el tabulador para cambiar el foco de botón.

Este es el resultado final:



Como siempre, puedes encontrar el código y las imágenes en la sección de descargas.

APÉNDICE D: Plugins

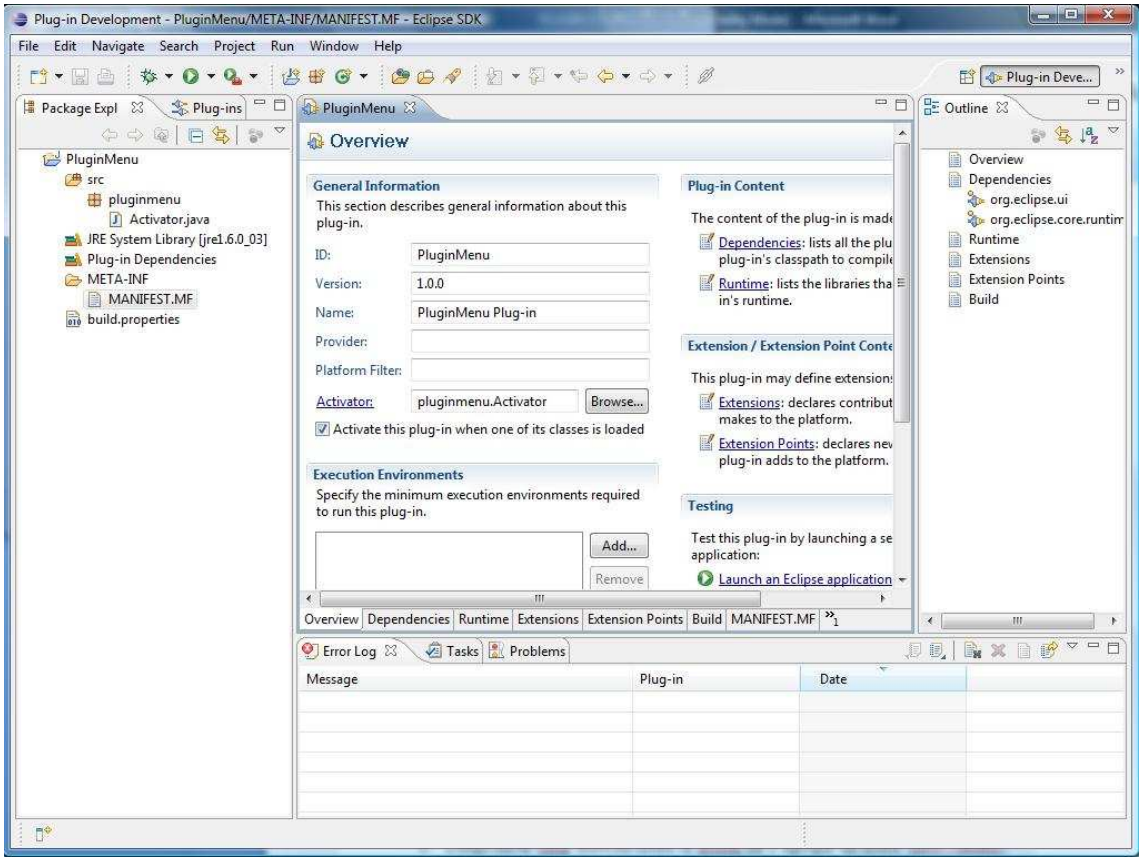
D.1. ¿Cómo crear un plugin menú?

Una vez inicializado Eclipse, creamos un nuevo proyecto **File->New Proyect->Plug-in Proyect** y le asignamos el nombre que queramos, para este ejemplo: PluginMenu.

Si no tenemos la perspectiva de plugin abierta nos preguntará si queremos abrirla, aceptamos.

Seguimos los siguientes pasos:

1. Con el "Plug-in Manifest Editor" abrimos el archivo plugin.xml
2. Seleccionamos la pestaña **Extensions** del editor.



3. Agregamos la extensión **org.eclipse.ui.actionSets**
4. Vamos a crear un nuevo **actionSet**, si no ha aparecido por defecto, hacemos click con el botón derecho sobre la extensión y seleccionamos actionSet.
5. Rellenamos los campos obligatorios de identificador único y etiqueta, por ejemplo:
   id: PluginMenu.ejemplo
   label: EtiquetaMenu

6. Ahora vamos a agregar un menú, para ello hacemos click con el botón derecho sobre el **actionSet** y hacemos new->menú.

7. Escribimos un id para el menú y un nombre.
   id: menuPrueba
   label: NuevoMenu

8. Al nuevo menú que hemos creado le podemos añadir nuevos grupos de acciones mediante new->groupMaker y separadores para estos grupos con new->Separator.

9. Creamos una nueva acción haciendo click con el botón derecho sobre el ActionSet: new->Action.

10. Rellenamos los atributos de la acción:

    a. Id: Identificador único de la acción
       id: PluginMenu.accion

    b. Label: Es la etiqueta que aparecerá en el menú
       label: NombreEtiqueta

    c. menubarPath: Indica a qué menú y a qué grupo se debe agregar la acción. El menú puede ser uno de los ya existentes en Eclipse o uno creado para el plugin. La estructura de menubarPath es: idMenu/nombreGrupo. Este campo puede ser vacío.

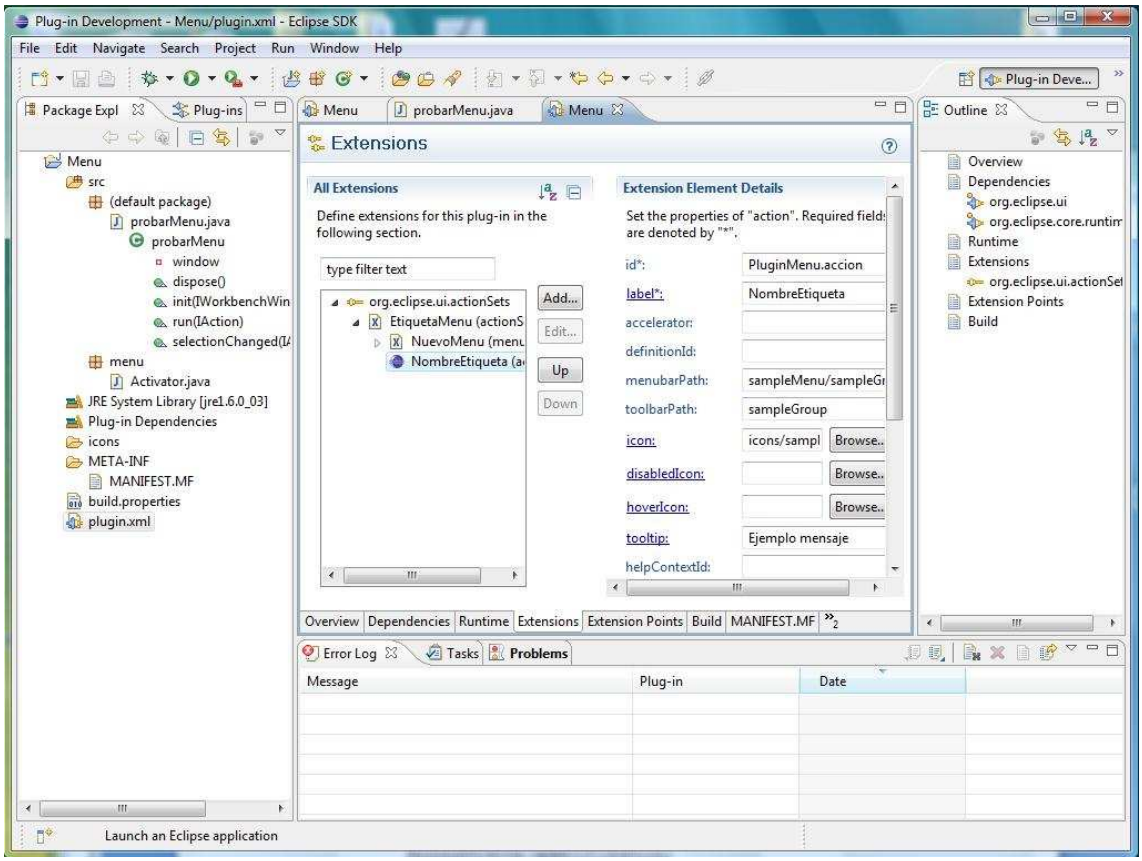
    d. toolbarPath: Se coloca el grupo al cual pertenece el botón en la barra de herramientas. Puede ser vacío

    e. icon: Icono asociado a la acción.
       icon: icons/sample.gif

    f. tooltip: mensaje que será desplegado cuando el ratón se encuentre sobre la acción.
       tooltip: Ejemplo mensaje

    g. class: clase asociada a la acción. Se creará una instancia de esta clase cuando se active la acción.
       class: probarMenu

El método más importante de una clase asociada a una acción es run, es aquí donde debemos colocar el código que será ejecutado cuando se active la acción.



Para nuestro ejemplo, escribimos en run:

```
MessageDialog.openInformation(
      window.getShell(),
            "Menu Plug-in",
            "Hola!, mensaje enviado desde el menú");
```

Necesitaremos como atributo de la clase:

private IWorkbenchWindow window;

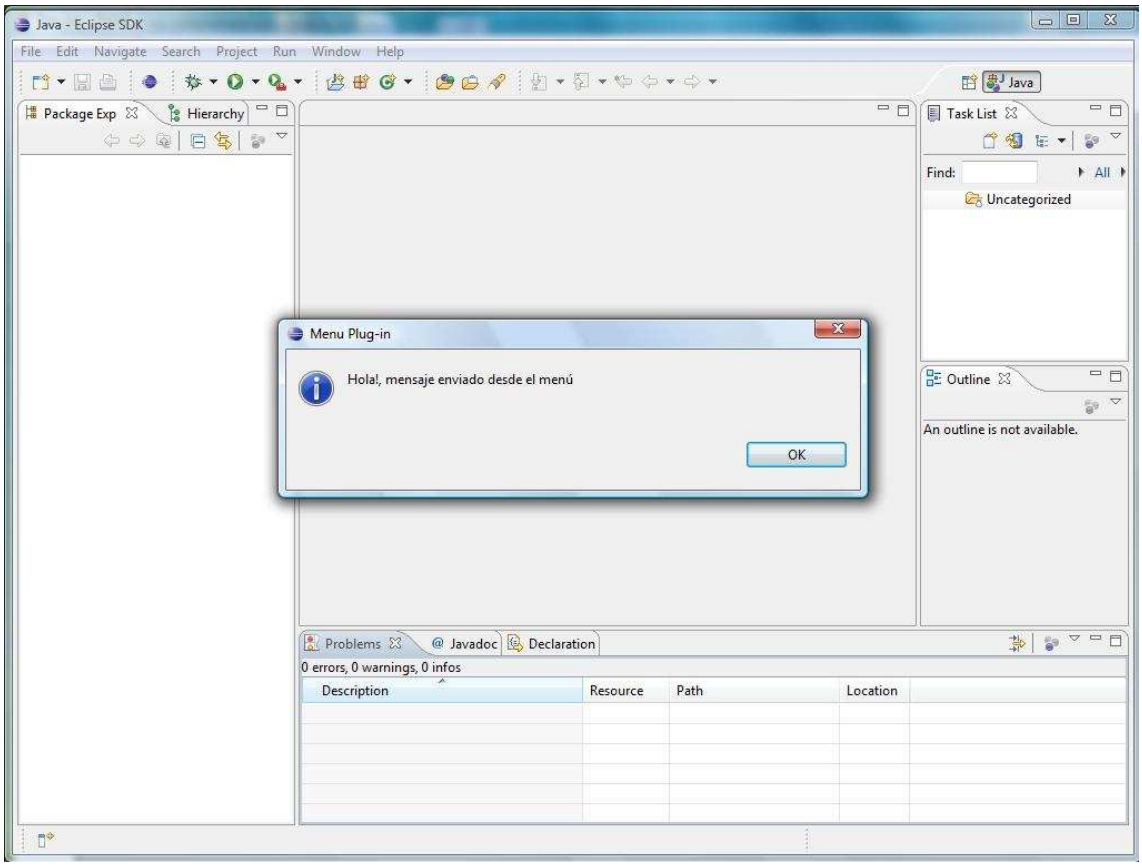
Además escribimos en el método init:

```
public void init(IWorkbenchWindow window) {
      this.window = window;

}
```

11. Para probar nuestro plugin nos situamos en la parte Testing de la pestaña **Overview** en plugin.xml y pulsamos sobre **Launch an Eclipse application**, se lanzará una nueva ventana de Eclipse donde podremos visualizar el nuevo plugin.

12. En esta nueva ventana podemos ver el nuevo menú que hemos creado con la imagen que elegimos sobre la barra de herramientas:

13. Si pulsamos sobre el botón podremos ver como se lanza una nueva ventana con el texto que introducimos anteriormente.



D.2. ¿Cómo crear un plugin vista?

Una vez inicializado Eclipse, creamos un nuevo proyecto **File->New Proyect->Plug-in Proyect** y le asignamos el nombre que queramos, para este ejemplo: PluginVista.

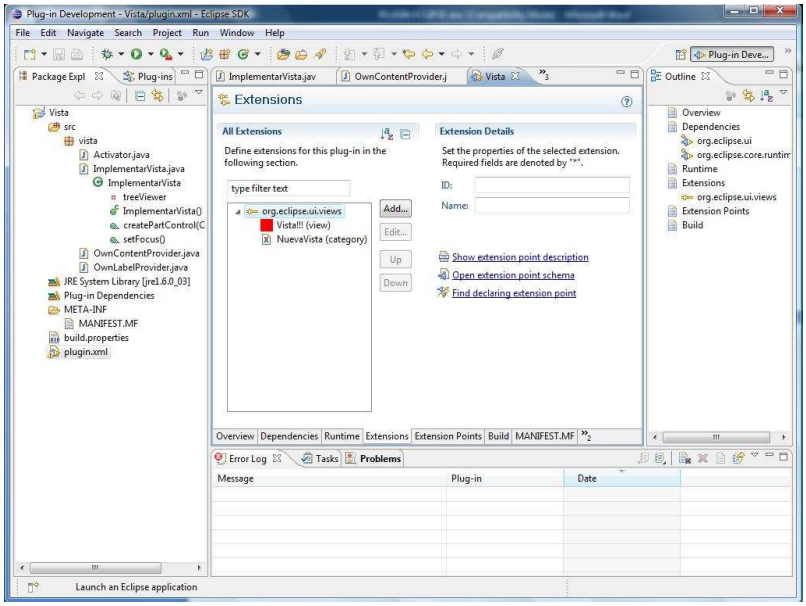
1. En la perspectiva Plug-in development realizamos los siguientes cambios:

2. En la ficha Extensions, pulsamos sobre el botón Add para añadir un nuevo punto de extensión y seleccionamos org.eclipse.ui.views.

3. Vamos a añadir una nueva categoría, que será desplegada en el momento de acceder al menú de Eclipse "Window.>Show View->Other", debemos ponerle un nombre significativo.
   name: NuevaVista

4. Añadimos una nueva vista seleccionando la extensión y pulsando con el botón derecho new->view.

5. Rellenamos los campos de la extensión:
   a. id: cada vista necesita un identificador único
      id: org. views.Vista
   b. Name: nombre con el que aparecerá la vista
      name: Vista!!!
   c. Class: clase que implementa la vista
      class: si pulsamos en class directamente nos permite crear la clase en este ejemplo la llamaremos ImplementarVista
   d. Category: para organizar las vistas
   e. Icon: icono de la vista
      icon: icono.png



6. Ya tenemos la nueva clase Vista creada, es necesario que extienda de ViewPart, ahora tenemos que especificar que es lo que queremos incluir en ella, para este ejemplo:

   En “CreatePartControl” añadimos:

```
public void createPartControl(Composite parent) {
      treeViewer = new TreeViewer(parent, SWT.BORDER | SWT.MULTI
      |SWT.V_SCROLL);
      //Establecemos el “LabelProvider”
      treeViewer.setLabelProvider(new OwnLabelProvider());
      //Establecemos el “ContentProvider”
      treeViewer.setContentProvider(new OwnContentProvider());
      //Establecemos el contenido raíz
      File f = new File("c:\\");
      treeViewer.setInput(f);
}
```

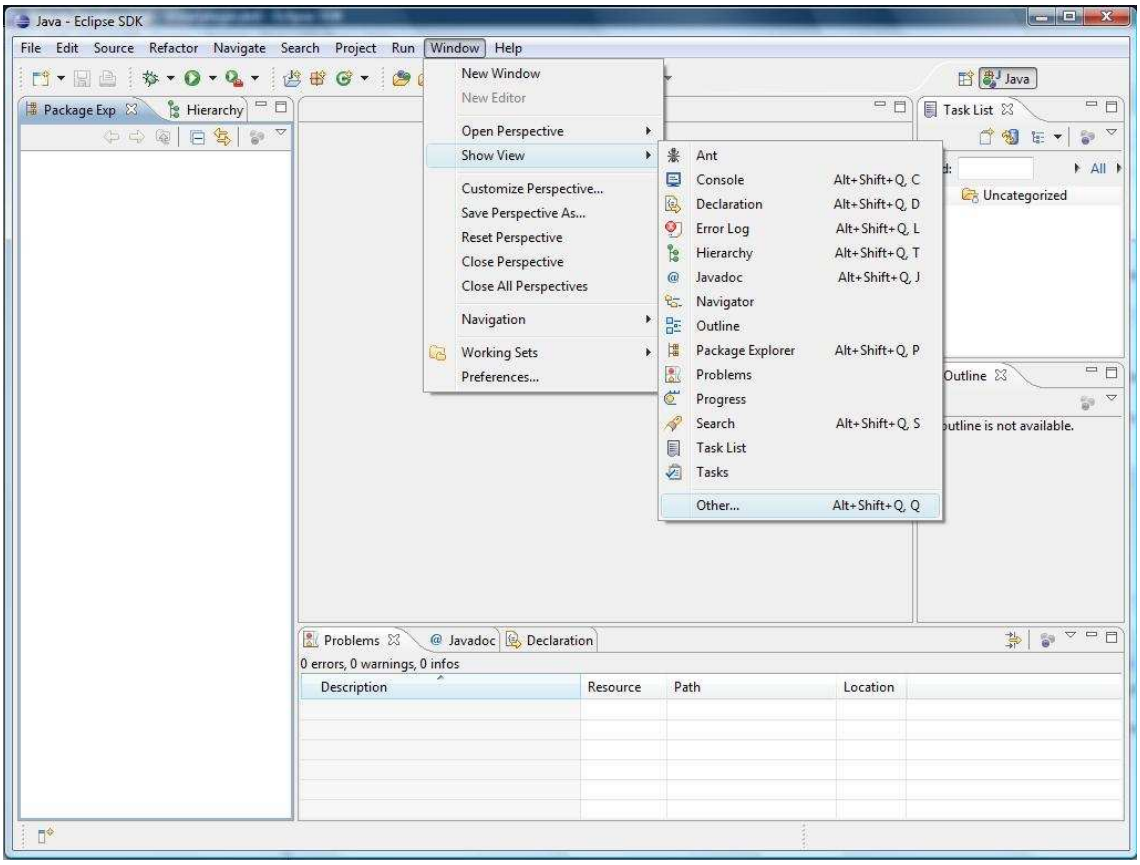
   Tendremos que importar algunos elementos de swt y de jface además de crear dos nuevas clases que definimos a continuación:

```
class OwnLabelProvider extends LabelProvider{

      public String getText(Object obj) {
            if (obj instanceof File)
                  return ((File)obj).getName();
            return obj.toString();
      }

      public org.eclipse.swt.graphics.Image getImage(Object obj) {
            if (obj instanceof File){
                  if (((File)obj).isFile())
                        return

      AbstractUIPlugin.imageDescriptorFromPlugin("HelloWorldView",
                        "text-x-generic.png").createImage();
                  if (((File)obj).isDirectory())
                        return

      AbstractUIPlugin.imageDescriptorFromPlugin("HelloWorldView",
                        "folder.png").createImage();
            }
            return null;
      }
}



class OwnContentProvider implements ITreeContentProvider{

      public Object[] getChildren(Object parentElement) {
            if (parentElement instanceof File)
                  return ((File)parentElement).listFiles();
            return null;
      }

      public Object getParent(Object element) {
            if (element instanceof File)
                  return ((File)element).getParentFile();
```

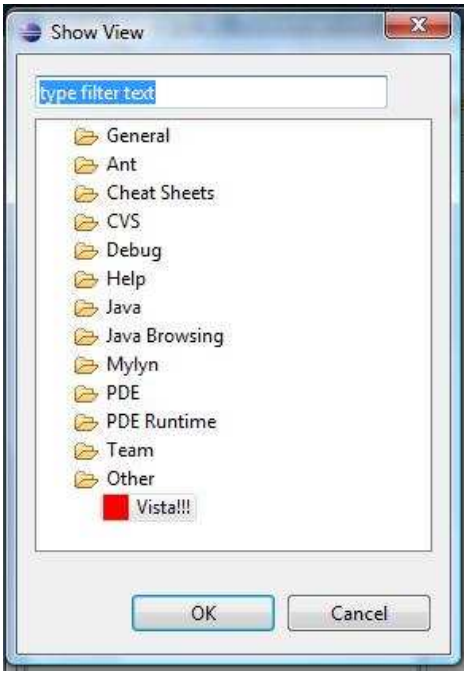
```
			return null;
		}

		public boolean hasChildren(Object element) {
			if (element instanceof File)
				return ((File)element).listFiles() != null;
			return false;
		}

		public Object[] getElements(Object inputElement) {
			if (inputElement instanceof File)
				return ((File)inputElement).listFiles();
			return null;
		}

		public void dispose() {
		}

		public void inputChanged(Viewer viewer, Object oldInput, Object
newInput)
		{ }
}
```

7. Ya podemos probar nuestro nuevo plugin, para ello en plugin.xml vamos a la pestaña **Overview** y en la parte de Testing pulsamos sobre **Launch an Eclipse application**, se lanzará una nueva ventana de Eclipse donde podremos visualizar el nuevo plugin.

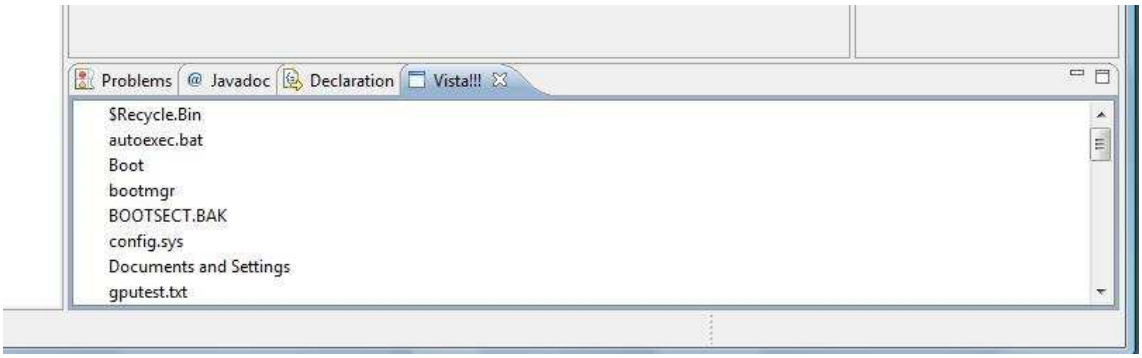
8. En la nueva ventana de Eclipse pulsamos en el menú Window->ShowView->Other



9. Aparecerá una ventana como esta donde podemos ver la nueva vista creada:



10. Pulsamos sobre ella y aparecerá la nueva vista



11. Podemos arrastrar la nueva vista para ver mejor su contenido