Adaptación y extensión de un SIG

Albert Gavarró Rodríguez

PID_00174756

Universitat Oberta de Catalunya
www.uoc.edu

Índice

Introducción

En el módulo anterior, nos hemos introducido en el mundo de la programación orientada a objetos de la mano del lenguaje Java. El objetivo del módulo era claro: proporcionar los conocimientos mínimos necesarios para realizar pequeñas implementaciones para, posteriormente, abordar con mínimas garantías la programación de un SIG.

Introducidos ya en el mundo de la programación, en este módulo aprenderemos a implementar un SIG de escritorio o –lo que es lo mismo– un SIG que se ejecuta en la máquina del usuario de la misma forma en que lo hace cualquier otra aplicación.

Como la programación desde cero de un SIG es muy compleja y nos llevaría muchas horas de dedicación (en caso de que poseyéramos los suficientes conocimientos teóricos y técnicos como para llevarla a cabo), lo cual no es nuestro objetivo, usaremos como punto de partida un SIG ya implementado, que adaptaremos a nuestras necesidades mediante el desarrollo de complementos. El SIG del que partiremos será gvSIG.

Objetivos

Una vez finalizado el módulo, deberíamos ser capaces de:

1. Extender gvSIG mediante complementos para adaptarlo a nuestras necesidades.

2. Agregar capas vectoriales a una vista de gvSIG.

3. Señalar lugares del mapa, trazar rutas o delimitar áreas mediante el uso de capas vectoriales.

4. Transformar las coordenadas del mapa en coordenadas geográficas y viceversa.

5. Determinar los límites visibles del mapa (*viewport*).

6. Implementar herramientas que interactúen con el mapa.

1. gvSIG

gvSIG es una herramienta orientada al manejo de información geográfica. Permite acceder a información vectorial y rasterizada georreferenciada, así como a servidores de mapas que cumplan las especificaciones que impone el OGC (*Open Geospatial Consortium* —Consorcio Abierto Geoespacial—).

Las razones para usar gvSIG son variadas:

- Está programado en Java, lo que significa que puede ejecutarse virtualmente en casi cualquier máquina.
- Es software libre*, lo que nos da la libertad de modificarlo y distribuirlo a nuestro antojo. El coste de este software es cero.
- Su funcionalidad se puede ampliar fácilmente mediante complementos que se pueden programar en Java o en Jython**.
- Está probado por una base amplia de usuarios, lo que asegura una buena estabilidad del código.

*** Está sometido a una licencia GNU GPL 2.0. Se pueden leer los términos de la licencia en: http://www.gnu.org/licenses/gpl-2.0.html**

**** Jython es una implementación del lenguaje Python para la plataforma Java: http://www.jython.org/**

1.1. Arquitectura de gvSIG

gvSIG se estructura alrededor de un núcleo de funcionalidad relativamente pequeño. A este núcleo, que contiene las partes esenciales del sistema, se le pueden añadir complementos para dotarlo de nuevas funcionalidades. Esta arquitectura permite descomponer el sistema en partes pequeñas, más fáciles de implementar y mantener. Actualmente, gran parte de la distribución de gvSIG son complementos.

Aunque los complementos pueden ser de naturaleza muy variopinta, siempre se muestran al usuario de forma homogénea. Es decir, el aspecto gráfico y la forma de trabajar con un complemento u otro serán más o menos parecidos.

Los complementos se integran en el marco de gvSIG mediante las llamadas "extensiones", que no son más que un conjunto de clases Java que añaden nuevas funcionalidades a la aplicación.

El núcleo de gvSIG está formado por tres subsistemas:

- ***FMap***. Agrupa la lógica SIG. Contiene las clases que se encargan de generar los mapas, gestionar las capas, transformar coordenadas y realizar búsquedas, consultas, análisis, etc.

- ***gvSIG***. Agrupa las clases encargadas de gestionar la interfaz gráfica de la aplicación y representar en pantalla los mapas generados por el subsistema *FMap*. En otras palabras, este subsistema se encarga de proporcionar una interfaz de usuario que proporciona acceso a las funcionalidades de gvSIG.

- ***Subdriver***. Contiene las clases que se encargan de dar acceso a los datos, tanto locales como remotos (por ejemplo, servicios OGC). Además, permite escribir datos geográficos en los formatos más comunes.

Para llevar a cabo su cometido, un complemento puede acceder a uno o más subsistemas de gvSIG.

1.2. Anatomía de un complemento

Como mínimo, un complemento está constituido por:

- un fichero XML que describe el complemento, y
- el código del complemento.

El fichero XML, que debe llamarse *config.xml*, proporciona a gvSIG la información necesaria sobre el complemento para que pueda cargarlo e integrarlo en la interfaz gráfica. Entre otras cosas, especifica el directorio que contiene el código del complemento, las dependencias que tiene con otros complementos y los elementos de interfaz de usuario (menús, barras de herramientas, etc.) que añade a la interfaz de gvSIG.

El código del complemento es un conjunto de clases Java que implementan la funcionalidad aportada por el complemento. Probablemente, el código irá acompañado de ficheros de configuración, imágenes, datos geográficos o cualquier otro dato que sea necesario para la ejecución de éste.

A lo largo de este módulo, vamos a implementar, a modo de ejemplo, un complemento llamado “Construcciones” que mostrará elementos constructivos (edificios y caminos) sobre una base topográfica. En los apartados siguientes, aprenderemos a crear un complemento y, por medio de él, a:

- agregar una capa de información geográfica a una vista,
- dibujar líneas y polígonos georeferenciados sobre un mapa, y
- crear una herramienta que muestre información sobre los elementos dibujados.

1.3. El fichero *config.xml*

El fichero *config.xml* contiene los datos de configuración del complemento. Este fichero, que obligatoriamente debe acompañar al código, define básica-

mente los elementos de interfaz de usuario que se van a añadir a la aplicación (herramientas, menús y controles de la barra de estado), a la vez que los asocia con clases del código. Cada una de estas asociaciones es una "extensión".

El fichero *config.xml* debe escribirse en lenguaje XML (de *eXtensible Markup Language* —'lenguaje de marcas extensible'—) y ha de seguir unas pautas determinadas.

Un documento XML está constituido por elementos que forman una jerarquía. Estos elementos se denotan mediante el uso de etiquetas de inicio y fin. Las etiquetas tienen la forma <*nombre*>, </*nombre*> o <*nombre* />, en donde *nombre* es el nombre del elemento. La primera es una etiqueta de inicio, la segunda de fin, y la última una combinación de ambas. Las dos primeras se utilizan cuando un elemento puede contener otros elementos, y sirven para marcar el inicio y el fin de dicho elemento. La última está reservada a elementos que no contienen otros. Además, los elementos pueden tener atributos cuyos valores se establecen de forma muy parecida a las asignaciones de los lenguajes de programación.

Veamos el fichero de configuración que usará nuestro complemento:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<plugin-config>
  <depends plugin-name="com.iver.cit.gvsig" />
  <libraries library-dir="."/>
  <extensions>
    <extension class-name=
      "edu.uoc.postgradosig. construcciones.Construcciones"
      description="Complemento que dibuja construcciones"
      active="true"
      priority="50">
      <menu text="Postgrado SIG/Construcciones"
       action-command="MENU_CONSTRUCCIONES" />
    </extension>
  </extensions>
</plugin-config>
```

La primera línea, llamada prólogo, describe la versión del lenguaje XML utilizada para escribir el documento y su codificación. Como el prólogo no forma parte de la configuración del complemento, no debemos preocuparnos demasiado por su contenido. Simplemente, siempre utilizaremos la misma fórmula.

El significado del resto de elementos es el siguiente:

- ***plugin-config***. Es una etiqueta que engloba todas las opciones de configuración. Es la raíz de la jerarquía de elementos.

- ***depends***. Enumera los complementos de los que depende nuestro complemento para funcionar correctamente. El nombre del complemento se especifica mediante el atributo "plugin-name". Debe haber un elemento "depends" por cada complemento. Un complemento depende de otro cuando el código del primero utiliza alguna de las clases del código del segundo. Normalmente, todos los complementos dependerán del complemento gvSIG ("com.iver.cit.gvsig"), que es la aplicación en sí.

- ***libraries***. Especifica el directorio en el que reside el código de nuestro complemento. El nombre del directorio se especifica mediante el atributo "library-dir". El punto como nombre de directorio significa que el código se encuentra en el mismo directorio que el fichero de configuración.

- ***extensions***. Marca el inicio de la lista de extensiones del complemento.

- ***extension***. Declara una extensión de gvSIG. La declaración incluye el nombre de la clase Java que implementa la extensión y una lista de elementos de interfaz de usuario que deben añadirse a la aplicación y que están asociados con la extensión. El elemento "extension" tiene los atributos siguientes:

  – ***class-name***: especifica el nombre de la clase que implementa la extensión.

  – ***description***: describe la funcionalidad que aporta la extensión. Esta etiqueta es meramente informativa.

  – ***active***: especifica si la extensión está activa o inactiva. Si el valor de este atributo es "false", gvSIG no cargará la extensión.

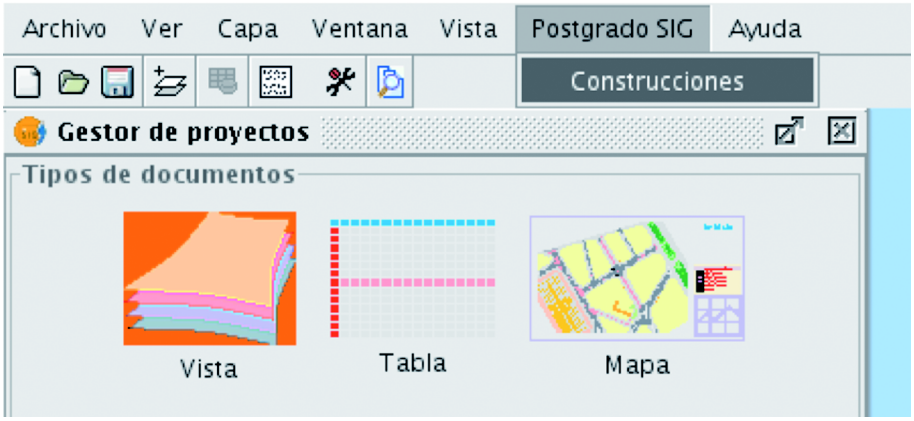
  – ***priority***: establece la prioridad de carga de la extensión respecto al resto de extensiones declaradas en el fichero de configuración. Cuanto menor sea el número, antes se cargará la extensión. En nuestro caso, sólo hay una extensión, por lo que el valor de la prioridad no es fundamental.

- ***menu***. Define una nueva entrada del menú de gvSIG y la asocia a la extensión. El elemento "menu" tiene dos atributos:

  – ***text***: establece el nombre y la ubicación del menú. Puede usarse la barra ("/") para establecer diversos niveles de menú.

  – ***action-command***: especifica el identificador de la acción. Cada vez que se active el menú, se llamará a la clase asociada pasándole el valor de este atributo. Esto nos permite asociar varios menús a una sola clase.

En el ejemplo, se declara la opción de menú *Postgrado SIG* → *Construcciones* con el identificador asociado "MENU_CONSTRUCCIONES". El resultado será el siguiente:



1.4. La clase "Extensión"

Una vez creado el fichero XML de configuración de la extensión, en cuyo contenido hemos definido la clase que la implementa y los elementos de interfaz de usuario que nos permitirán acceder a su funcionalidad, llega el momento de implementarla.

Para ser consideradas como tales, las extensiones deben extender (valga la redundancia) la clase "Extension"* y sobrecargar sus funciones. Al reconocer la clase como una extensión, gvSIG se comunicará con ella por medio de estas funciones, que actuarán a modo de interfaz entre gvSIG y la extensión.

*** La documentación de la clase se puede consultar en: API gvSIG/com/iver/andami/plugins/Extension.html**

A continuación, se muestra una posible implementación de la extensión "Construcciones":

```
package edu.uoc.postgradosig.construcciones;

import com.iver.andami.plugins.Extension;
import javax.swing.JOptionPane;

public class Construcciones extends Extension {
  public void initialize() {
    // aquí se realizan las tareas de inicialización
  }

  public boolean isEnabled() {
    return true; // está habilitada
  }
```

```
  public boolean isVisible() {
    return true; // es visible
  }

  public void execute(String actionCommand) {
    // muestra un mensaje por pantalla
    JOptionPane.showMessageDialog(null,
      "Esta extensión muestra elementos constructivos.");
  }
}
```

Como se puede observar, la clase "Construcciones" extiende la clase "Extension" e implementa (sobrecarga) cuatro funciones: *initialize*, *isEnabled*, *isVisible* y *execute*. El cometido de cada una de ellas es el siguiente:

- ***void initialize()***. Se utiliza para inicializar la extensión. En esta función emplazaremos llamadas a funciones que se encarguen de reservar recursos, leer configuraciones, etc. Debemos fijarnos en que esta función desempeña un papel muy parecido al del constructor de la clase. Sin embargo, por coherencia con gvSIG, reservaremos las tareas más arduas para esta función, limitando el cometido del constructor —si lo hay— a la inicialización de variables.

- ***boolean isEnabled()***. Indica si la extensión debe considerarse habilitada o no. Cuando una extensión no está habilitada, los elementos de interfaz de usuario (botones, menús o controles de la barra de estado) que tiene asociados no son accesibles. En la mayoría de sistemas, estos elementos se mostrarán en tonos grisáceos. gvSIG llama periódicamente a esta función respondiendo a los eventos de la interfaz gráfica.

- ***boolean isVisible()***. Indica si los elementos de interfaz de usuario que tiene asociados la extensión deben estar visibles o no. Si esta función devuelve *false*, no se mostrará ninguno de sus elementos. gvSIG llama periódicamente a esta función respondiendo a los eventos de la interfaz gráfica.

- ***void execute(String actionCommand)***. gvSIG llama a esta función cada vez que se activa la extensión, es decir, cada vez que el usuario accede a alguno de los elementos de interfaz de usuario que la extensión ha definido. El parámetro "actionCommand" contiene el valor del atributo "action-command" (definido en el fichero de configuración) del elemento al que se ha accedido.

Todas las extensiones están obligadas a implementar estas cuatro funciones. Al tratarse de funciones sobrecargadas, debemos prestar especial atención a respetar tanto los modificadores de visibilidad como el tipo de la función y de los parámetros. Opcionalmente, si se desea un mayor control sobre los proce-

sos de inicialización y descarga de la extensión, también se pueden implementar las funciones siguientes:

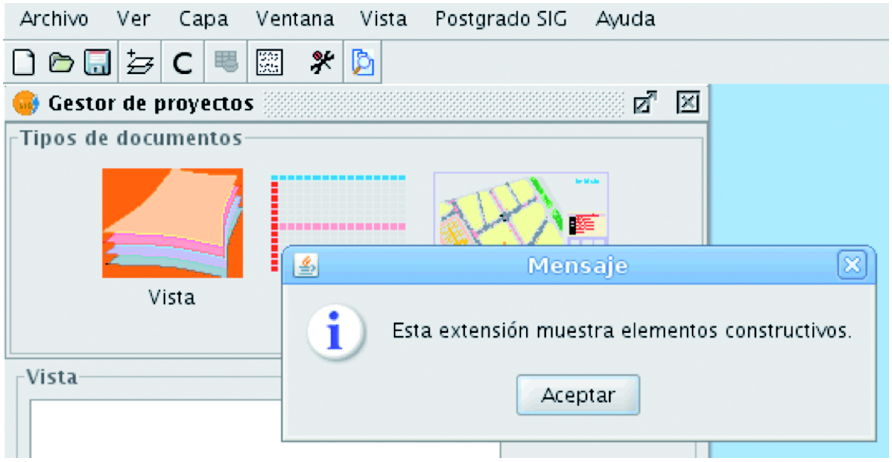
- ***void postInitialize()***. Se llama a esta función después de haberse llamado a la función *initialize* de todas las extensiones. Se utilizará para realizar tareas de inicialización que requieran una interacción con otras extensiones.
- ***void terminate()***. Se llama a esta función al terminar la ejecución de gvSIG. Típicamente, en esta función almacenaremos las preferencias del usuario, liberaremos recursos, etc.

Volvamos al ejemplo: la extensión mostrará una ventana con un mensaje descriptivo cada vez que se active. Este código, combinado con el fichero de configuración que ya habíamos definido, dará lugar a que se muestre el mensaje cada vez que se seleccione la opción de menú *Postgrado SIG* → *Construcciones*. Cabe destacar que, en este caso, el contenido del parámetro "actionCommand" será "MENU_CONSTRUCCIONES", tal y como habíamos definido en el fichero de configuración. También es importante observar que las funciones *isEnabled* e *isVisible* devuelven *true*, lo que indica en todo momento que la extensión está habilitada y es visible.

Debemos observar también que el mensaje informativo se muestra mediante la función *showMessageDialog* de la clase "JOptionPane"*.

* http://java.sun.com/javase/6/docs/api/javax/swing/JOptionPane.html

El resultado será el siguiente:



1.5. Servicios prestados a las extensiones

Hasta ahora hemos visto cómo gvSIG se comunica con las extensiones para determinar su estado o acceder a su funcionalidad, pero hemos ignorado cómo las extensiones acceden al contexto de gvSIG. Entendemos como contexto todas aquellas clases y objetos que representan las ventanas, las vistas, los mapas y los demás elementos que conforman el entorno de ejecución de gvSIG. El acceso a este contexto se logra mediante la clase "PluginServices"*.

* Todas las clases que componen la versión 1.1 de la API gvSIG están explicadas en detalle en la documentación de la API que se entrega conjuntamente con estos materiales. La página principal de la documentación se encuentra en: API gvSIG/index.html

Si continuamos con la implementación de nuestra extensión, es deseable que la extensión (y, en consecuencia, la opción del menú) sólo esté habilitada si se

trabaja sobre una vista. Para conseguir este efecto, utilizaremos la clase "PluginServices" para obtener la ventana activa, comprobaremos si se trata de una ventana de vista e informaremos a gvSIG sobre si está habilitada o no la extensión mediante la función *isEnabled*:

```
package edu.uoc.postgradosig.construcciones;

import com.iver.andami.plugins.Extension;
import javax.swing.JOptionPane;
import com.iver.andami.PluginServices;
import com.iver.cit.gvsig.project.documents.view.gui.View;
import com.iver.andami.ui.mdiManager.IWindow;

public class Construcciones extends Extension {

  public void initialize() {
  }

  public boolean isVisible() {
    return true; // es visible
  }

  public boolean isEnabled() {
    IWindow ventanaActiva;

    // obtiene la ventana activa
    ventanaActiva = PluginServices.getMDIManager().
      getActiveWindow();

    // estará habilitada sólo si la ventana activa es una
    // ventana de vista
    return ventanaActiva instanceof View;
  }

  public void execute(String actionCommand) {
    // muestra un mensaje por pantalla
    JOptionPane.showMessageDialog(null,
      "Esta extensión muestra elementos constructivos.");
  }
}
```

Como se puede observar, utilizamos la clase "PluginServices" para obtener, en primer lugar, el gestor de ventanas de gvSIG*, y, después, la ventana activa, mediante las funciones *getMDIManager* y *getActiveWindow* respectivamente:

*** Un gestor de ventanas es un software que se encarga de gestionar un entorno de ventanas. Proporciona los mecanismos necesarios para crear, alterar, ordenar, presentar y destruir ventanas, entre otros.**

```
ventanaActiva = PluginServices.getMDIManager().
  getActiveWindow();
```

La ventana activa es de la clase "IWindow". De hecho, todas las ventanas de gvSIG extienden esta clase. Por ejemplo, las clases "View" y "Table" extienden "IWindow" e implementan las ventanas de vista y de tabla respectivamente.

Obtenida la ventana activa, sólo nos queda determinar si se trata de una ventana de vista o no. Esto lo hacemos mediante el operador *instanceof*, que nos permite comprobar si un objeto es instancia de una clase determinada*. Como ya hemos visto, el objeto "ventanaActiva" será instancia de la clase "View" si se trata de una ventana de vista. En consecuencia, la extensión estará activa sólo si "ventanaActiva" es una instancia de la clase "View":

* Dada una clase *C* y un objeto *o*, *o instanceof C* devolverá *true* si *o* es una instancia de *C* o de alguna de las clases que extienden *C*. En cualquier otro caso se devolverá *false*.

```
return ventanaActiva instanceof View;
```

La clase "PluginServices" proporciona otras muchas funciones que dan soporte a las extensiones. A continuación, comentamos las más relevantes:

- ***String[] getArguments()***. Devuelve un vector con los parámetros de arranque de la aplicación.

- ***String getArgumentByName(String name)***. Devuelve el valor del parámetro de arranque especificado. Para que funcione correctamente, el parámetro debe tener la forma "-<nombre_parámetro>=<valor>".

- ***Logger getLogger()***. Devuelve un objeto "Logger" mediante el que se pueden escribir trazas en el fichero de *log** de gvSIG. El objeto proporciona (entre otras) tres funciones (*error*, *warn* e *info*) que permiten escribir distintos tipos de mensajes según su criticidad (de error, de aviso y de información respectivamente).

* Un fichero de *log* es aquel que almacena información sobre los sucesos que ocurren en un sistema. Es como un cuaderno de bitácora en el que se escriben evidencias que más tarde servirán par detectar posibles anomalías en el software.

- ***MainFrame getMainFrame()***. Devuelve el objeto que representa a la ventana principal de gvSIG. Este objeto se utiliza principalmente para cambiar la herramienta activa.

- ***MDIManager getMDIManager()***. Devuelve un objeto de la clase "MDIManager" que representa al gestor de ventanas de gvSIG. Utilizaremos el gestor cuando queramos añadir nuevas ventanas o gestionar las existentes.

- ***void setPersistentXML(XMLEntity entity)***. Permite almacenar información sobre el estado de la extensión que podrá ser recuperada más tarde. Puede ser interesante guardar información de estado al finalizar la sesión de gvSIG.

- ***XMLEntity getPersistentXML()***. Devuelve un objeto "XMLEntity" que contiene el último estado guardado de la extensión. Es la función complementaria de "setPersistentXML".

1.6. Agregar una capa vectorial

Para representar elementos constructivos sobre un mapa, utilizaremos líneas y polígonos. Las líneas nos permitirán dibujar caminos, y los polígonos, edificios.

El primer paso será crear un mapa vectorial en memoria. Una vez creado, dibujaremos sobre él los elementos constructivos, que previamente habremos georeferenciado, y les asociaremos datos. Terminado el mapa, lo sobrepondremos a una base topográfica mediante el uso de capas y obtendremos así un nuevo mapa.

1.6.1. Crear una capa vectorial en memoria

Para crear un mapa vectorial en memoria, usaremos la clase "ConcreteMemoryDriver". Las líneas siguientes:

```
ConcreteMemoryDriver mapa;

mapa = new ConcreteMemoryDriver();
mapa.setShapeType(FShape.LINE + FShape.POLYGON);
mapa.getTableModel().setColumnIdentifiers(new String[] {
  "ID", "Descripcion"});
```

crean un mapa vectorial llamado "mapa". Además, se establecen, mediante la función *setShapeType*, los tipos de figuras que contendrá, "FShape.LINE" (por líneas) y "FShape.POLYGON" (por polígonos), y mediante la función *setColumnIdentifiers*, los atributos asociados a cada una de ellas, "ID" y "Descripcion". Los atributos se pueden definir arbitrariamente según nuestras necesidades.

1.6.2. Dibujar líneas y polígonos

Una vez creado el mapa, podemos empezar a añadirle figuras. Supongamos que, en una fase previa de captación de datos, hemos localizado una vivienda y un camino cuyos vértices son los siguientes:

Objeto	Vértices (UTM 31N/ED50)	
	x	*y*
Vivienda	356.270,7	4.727.507,1
	356.273,0	4.727.503,3
	356.278,5	4.727.505,8
	356.275,5	4.727.509,8
Camino	356.242,7	4.727.498,8
	356.268,0	4.727.509,8
	356.281,0	4.727.519,1

La vivienda la dibujaremos mediante el objeto "FPolygon2D". El código siguiente:

```
GeneralPathX ruta;
FPolygon2D edificio;

// definimos la geometría del edificio
ruta = new GeneralPathX();
ruta.moveTo(356270.7, 4727507.1); // mueve el cursor al inicio
ruta.lineTo(356273.0, 4727503.3); // dibuja el primer segmento
ruta.lineTo(356278.5, 4727505.8); // dibuja el segundo segmento
ruta.lineTo(356275.5, 4727509.8); // dibuja el tercer segmento
ruta.lineTo(356270.7, 4727507.1); // dibuja el cuarto segmento

// creamos el objeto edificio a partir de la geometría definida
// previamente
edificio = new FPolygon2D(ruta);

// agregamos el edificio al mapa
mapa.addShape(edificio, new Object[] {
  // valor del atributo "ID"
  ValueFactory.createValue(1),
  // valor del atributo "Descripcion"
  ValueFactory.createValue("Casa unifamiliar.") });
```

dibuja la vivienda y la agrega al mapa. Como se puede observar, primero usamos un objeto de la clase "GeneralPathX" para definir la geometría de la figura, y después creamos el polígono mediante la clase "FPolygon2D". En un último paso, lo agregamos al mapa y especificamos el valor de sus atributos. Observad que la función *createValue* está sobrecargada, lo que permite pasarle por parámetro tanto enteros como texto. También debemos notar que trabajamos directamente con coordenadas geográficas.

El camino lo dibujaremos de forma muy parecida. La única diferencia es que utilizaremos la clase "FPolyline2D" en lugar de "FPolygon2D":

```
GeneralPathX ruta;
FShape camino;

// definimos la geometría del edificio
ruta = new GeneralPathX();
ruta.moveTo(356242.7, 4727498.8);
ruta.lineTo(356268.0, 4727509.8);
ruta.lineTo(356281.0, 4727519.1);
```

```
// creamos el objeto camino a partir de la geometría
// definida previamente
camino = new FPolyline2D(ruta);

// agregamos el camino al mapa
mapa.addShape(camino, new Object[] {
     ValueFactory.createValue(2),
     ValueFactory.createValue("Camino de herradura.") });
```

1.6.3. Crear una capa

Una vez dibujado el mapa, ya sólo nos queda agregarlo a la vista actual como una capa. Esta tarea se reparte entre las clases "LayerFactory", que crea la capa, y "MapControl", que la agrega a la vista actual:

```
FLayer capa;
MapControl controlMapa;
View vistaActual;

// obtiene la ventana de vista activa
vistaActual = (View)PluginServices.getMDIManager().
 getActiveWindow();

// obtiene el objeto que controla el mapa
controlMapa = vistaActual.getMapControl();

// crea una capa a partir de un mapa
capa = LayerFactory.createLayer("Construcciones", mapa,
 controlMapa.getProjection());

// hace visible la capa
capa.setVisible(true);

// agrega la capa a la vista actual
controlMapa.getMapContext().getLayers().addLayer(capa);
```

Como se puede observar, la capa se crea mediante la función *createLayer* de la clase "LayerFactory". Esta función espera tres argumentos: el nombre de la capa ("Construcciones"), el mapa que contendrá y la proyección con la que se representará (la misma que se utiliza en la vista). En un segundo paso, se agrega la capa a la vista mediante la función *addLayer*. Como podemos ver, para poder llamar a la función necesitamos obtener, en primer lugar, el contexto del mapa ("getMapContext"), y en segundo lugar, la colección de las capas que contiene ("getLayers").

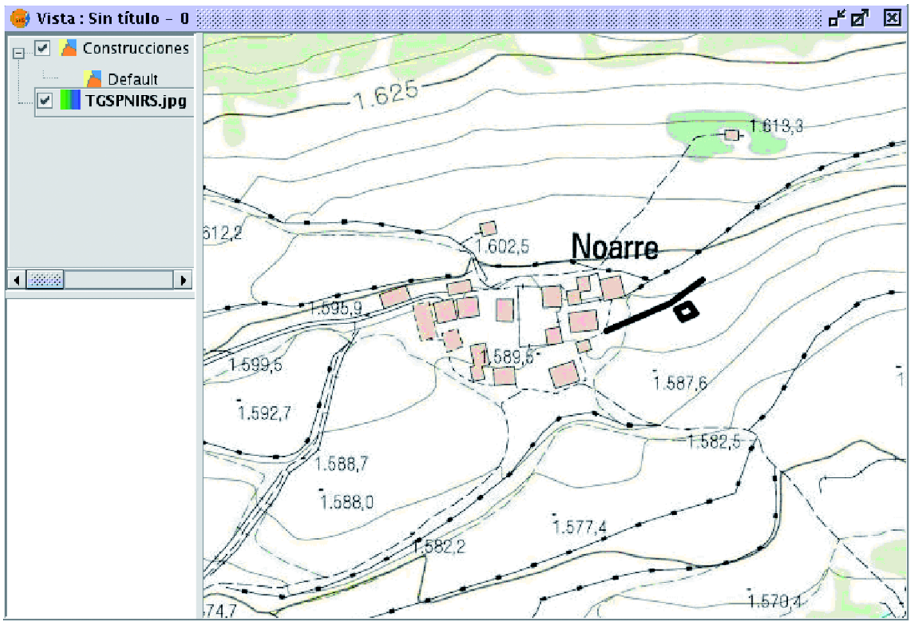
Recapitulando, el código de la extensión nos queda así:

```
package edu.uoc.postgradosig.construcciones;

import com.hardcode.gdbms.engine.values.ValueFactory;
import com.iver.andami.PluginServices;
import com.iver.andami.plugins.Extension;
import com.iver.andami.ui.mdiManager.IWindow;
import com.iver.cit.gvsig.fmap.MapControl;
import com.iver.cit.gvsig.fmap.core.FPolygon2D;
import com.iver.cit.gvsig.fmap.core.FPolyline2D;
import com.iver.cit.gvsig.fmap.core.FShape;
import com.iver.cit.gvsig.fmap.core.GeneralPathX;
import com.iver.cit.gvsig.fmap.drivers.
  ConcreteMemoryDriver;
import com.iver.cit.gvsig.fmap.layers.FLayer;
import com.iver.cit.gvsig.fmap.layers.LayerFactory;
import com.iver.cit.gvsig.project.documents.view.gui.View;

public class Construcciones extends Extension {

  public void execute(String actionCommand) {
    ConcreteMemoryDriver mapa;
    GeneralPathX ruta;
    FShape edificio, camino;
    FLayer capa;
    MapControl controlMapa;
    View vistaActual;

    // crea un mapa vectorial en memoria
    mapa = new ConcreteMemoryDriver();
    mapa.setShapeType(FShape.LINE + Fshape.POLYGON);
    mapa.getTableModel().setColumnIdentifiers(
      new String[] { "ID", "Descripcion"});

    // definimos la geometría del edificio
    ruta = new GeneralPathX();
    ruta.moveTo(356270.7, 4727507.1);
    ruta.lineTo(356273.0, 4727503.3);
    ruta.lineTo(356278.5, 4727505.8);
    ruta.lineTo(356275.5, 4727509.8);
    ruta.lineTo(356270.7, 4727507.1);

    // creamos el objeto edificio a partir de la geometría
    // definida previamente
    edificio = new Fpolygon2D(ruta);
```

```
    // agregamos el edificio al mapa
    mapa.addShape(edificio, new Object[] {
      ValueFactory.createValue(1),
      ValueFactory.createValue("Casa unifamiliar.") });

    // definimos la geometría del camino
    ruta = new GeneralPathX();
    ruta.moveTo(356242.7, 4727498.8);
    ruta.lineTo(356268.0, 4727509.8);
    ruta.lineTo(356281.0, 4727519.1);

    // creamos el objeto camino a partir de la geometría
    // definida previamente
    camino = new Fpolyline2D(ruta);

    // agregamos el camino al mapa
    mapa.addShape(camino, new Object[] {
      ValueFactory.createValue(2),
      ValueFactory.createValue("Camino de herradura.") });

    // crea la capa
    vistaActual = (View)PluginServices.getMDIManager().
      getActiveWindow();
    controlMapa = vistaActual.getMapControl();

    capa = LayerFactory.createLayer("Construcciones", mapa,
      controlMapa.getProjection());
    capa.setVisible(true);

    // agrega la capa a la vista actual
    controlMapa.getMapContext().getLayers().addLayer(capa);
  }

  public boolean isEnabled() {
    IWindow ventanaActiva;

    // obtiene la ventana activa
    ventanaActiva = PluginServices.getMDIManager().
      getActiveWindow();

    // estará habilitada sólo si la ventana actual es una
    // ventana de vista
    return ventanaActiva instanceof View;
  }
```

```
  public boolean isVisible() {
    // siempre estará visible
    return true;
  }

  public void initialize() {

  }
}
```

Si ejecutamos la extensión sobre una ventana de vista, el resultado será parecido al siguiente:

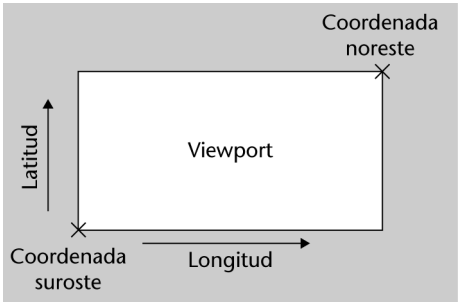


Como podemos apreciar, la vista contiene dos capas: la base topográfica ("TGSPNIRS.jpg") y la capa "Construcciones" que acabamos de crear. Mediante las clases "GPolygon2D" y "GPolyline2D" hemos conseguido dibujar un edificio y un camino al este del pueblo de Noarre.

1.7. El *viewport*

Cuando se muestra un mapa por pantalla, no siempre se muestra en toda su extensión. Que veamos una parte mayor o menor del mismo depende del nivel de *zoom*. De hecho, el área visible y el nivel de *zoom* se rigen por una regla de proporcionalidad inversa: a mayor *zoom*, menor área, y a menor *zoom*, mayor área.

El recuadro que delimita el área visible de un mapa se llama *viewport*. Normalmente, se define por las coordenadas geográficas de sus esquinas inferior izquierda (suroeste) y superior derecha (noreste). Latitud y longitud crecen desde la coordenada suroeste hasta la coordenada noreste.



gvSIG nos da acceso al *viewport* mediante la función *getViewPort* de la clase "MapControl".

El *viewport* es un objeto importante en gvSIG, pues nos proporciona mecanismos para transformar las coordenadas de la pantalla en coordenadas geográficas y viceversa. Esto nos servirá, por ejemplo, para determinar las coordenadas de un punto seleccionado mediante el ratón.

Suponiendo que las variables *x* e *y* (de tipo *int*) son las componentes de un punto del *viewport*, el código siguiente:

```
View vistaActiva;
MapControl controlMapa;
ViewPort viewport;
Point2D geoPoint;

// obtiene el viewport
vistaActiva = (View)PluginServices.getMDIManager().
 getActiveWindow();
controlMapa = vistaActiva.getMapControl();
viewport = controlMapa.getViewPort();

// transforma las coordenadas del viewport en coordenadas
// geográficas
geoPoint = viewport.toMapPoint(x, y);

JOptionPane.showMessageDialog(null, "Latitud: " +
 geoPoint().getX() + " Longitud: " +
 geoPoint.getY());
```

muestra por pantalla las coordenadas geográficas que se corresponden con este punto. Como se puede observar, la función que hace la transformación es *toMapPoint*. Esta función devuelve un objeto "Point2D" que contiene las coordenadas geográficas resultantes, cuyas componentes se pueden consultar mediante las funciones *getY* (latitud) y *getX* (longitud).

Otras funciones interesantes que nos ofrece la clase *viewport* son las siguientes:

- ***fromMapPoint***. Hace la transformación inversa: de coordenadas geográficas a coordenadas del *viewport*.

- ***fromMapDistance***. Transforma una distancia geográfica en una distancia del *viewport*.
- ***toMapDistance***. Transforma una distancia del *viewport* en una distancia geográfica.
- ***getAdjustedExtend***. Devuelve los límites geográficos del *viewport*, es decir, las coordenadas de las esquinas.

1.8. Herramientas

Hasta ahora, hemos visto cómo agregar datos al mapa de manera estática, es decir, desde el mismo código del complemento, pero en ningún momento hemos abordado la interacción con el usuario. Sin embargo, es rara la aplicación SIG que no ofrezca al usuario algún tipo de mecanismo interactivo para consultar los datos relevantes de los objetos que aparecen en el mapa.

Durante el proceso de creación del mapa vectorial, hemos definido una serie de atributos para todos los elementos del mapa. Más adelante, para cada línea y polígono, hemos asignado valores concretos a estos atributos. Ahora nuestro objetivo será crear una herramienta que permita al usuario seleccionar un elemento y consultar la información que tenga asociada.

El mecanismo de acción será el siguiente:

- El usuario activará la herramienta mediante un botón de la barra de herramientas.
- El usuario seleccionará un elemento del mapa haciendo clic sobre él.
- Aparecerá una ventana en la que se mostrarán los valores de los atributos del elemento.

Veamos paso a paso cómo implementar esta funcionalidad.

1.8.1. Agregar un botón a la barra de herramientas

El primer paso será agregar un botón a la barra de herramientas de la aplicación. Los botones, como todo elemento gráfico, pueden añadirse mediante el fichero de configuración *config.xml*. Las líneas siguientes:

```
<tool-bar name="Postgrado SIG" position="2">
  <action-tool icon="seleccion.png"
    tooltip="Localizador" position="1"
    action-command="BOTON_LOCALIZADOR"/>
</tool-bar>
```

añaden un botón a la barra de herramientas de gvSIG. El significado de cada uno de los elementos se detalla a continuación:

- ***tool-bar***. Define una barra de herramientas. Este elemento tiene dos atributos:
  - ***name***: especifica el nombre de la barra de herramientas.
  - ***position***: especifica la posición que ocupará la barra de herramientas entre las demás.

- ***action-tool***. Define un botón de la barra de herramientas. Este elemento, que debe ir siempre dentro de un elemento "tool-bar", tiene cuatro atributos:
  - ***icon***: especifica el nombre del fichero que contiene la imagen que aparecerá en el botón.
  - ***tooltip***: especifica el texto que se mostrará para describir la herramienta.
  - ***position***: especifica la posición del botón dentro de la barra de herramientas.
  - ***action-command***: al igual que el atributo "action-command" de los menús, define un identificador para la acción (este identificador se utilizará en las llamadas a la función *exec* de la extensión).

El elemento "tool-bar" debe ir siempre dentro de un elemento "extension". Si incorporamos estas líneas al fichero de configuración de la extensión "Construcciones", el resultado será el siguiente:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<plugin-config>
  <depends plugin-name="com.iver.cit.gvsig" />
  <libraries library-dir="."/>
  <extensions>
    <extensionclass-name=
      "edu.uoc.postgradosig.construcciones.Construcciones"
      description="Complemento que dibuja construcciones"
      active="true"
      priority="50">
      <menu text="Postgrado SIG/Construcciones"
        action-command="MENU_CONSTRUCCIONES" />
    <tool-bar name="Postgrado SIG" position="2">
      <action-tool icon="consulta.png"
        tooltip="Consultar construcciones"
        position="1"
        action-command="BOTON_CONSTRUCCIONES"/>
    </tool-bar>
    </extension>
  </extensions>
</plugin-config>
```

Es importante observar que el atributo "action-command" del menú y del botón son diferentes: en el primer caso es MENU_CONSTRUCCIONES, mientras que en el segundo caso es BOTON_CONSTRUCCIONES. Esto permite a la extensión (mediante el parámetro que recibe en la función *execute*) averiguar por medio de qué elemento de la interfaz de usuario ha sido llamada: si por medio del menú o por medio del botón de la herramienta.

El efecto sobre la interfaz de usuario de gvSIG será el siguiente:

Como se puede observar, se añade un botón "C" a la barra de herramientas.

1.8.2. Escuchar los eventos del ratón

Una vez configurada la extensión, debemos prepararla para que sea capaz de escuchar los eventos del ratón. Concretamente, nos interesa el evento "clic", que se produce cuando se hace clic sobre un punto del mapa. Esto se hace implementando la interfaz* "PointListener" y registrando la clase como escucha de dichos eventos.

*** Una interfaz es como una especie de clase cuyas funciones están definidas, pero carecen de código. Implementar una interfaz es similar a extender una clase, aunque nos veremos obligados a proporcionar una implementación para todas y cada una de las funciones que ésta defina. Generalmente, se usa para obligar a determinadas clases a implementar un conjunto de funciones que se utilizarán para comunicarse con ella. Una clase puede implementar una o más interfaces.**

La interfaz "PointListener" define las siguientes funciones, que deben ser implementados obligatoriamente por la extensión:

- ***void point(PointEvent)***. Se llama a esta función cada vez que se hace clic sobre un mapa. El objeto "PointEvent" contiene las coordenadas sobre las que se ha desatado el evento.

- ***void pointDoubleClick(PointEvent)***. Esta función tiene el mismo cometido que la anterior, pero para el doble clic. El objeto "PointEvent" también contendrá las coordenadas sobre las que se ha desatado el evento.

- ***boolean cancelDrawing()***. Indica a gvSIG si debe cancelarse el dibujado en curso. Un entorno estable puede ser deseable para determinadas herramientas.

- ***Cursor getCursor()***. Devuelve el cursor de la herramienta. Mientras la herramienta está activa, el ratón mostrará este cursor.

Las líneas siguientes reflejan una posible implementación de estas funciones por parte de la extensión "Construcciones", que mostraría las coordenadas del punto seleccionado cada vez que se hiciese clic sobre el mapa. Observad que no se reproduce la clase al completo:

```
public class Construcciones extends Extension
  implements PointListener {

  (...)

  public void point(PointEvent event)
    throws BehaviourException {
    Point2D puntoViewport;

    // obtiene el punto seleccionado
    puntoViewport = event.getPoint();

    // muestra las coordenadas del punto
    JOptionPane.showMessageDialog(null,
      "x=" +puntoViewport.getX() +
      "y=" +puntoViewport.getY());
  }

  public void pointDoubleClick(PointEvent event)
    throws BehaviourException {
    // ante un doble clic no hace nada
  }

  public boolean cancelDrawing() {
    return false; // no nos interesa cancelar el dibujado
  }

  public Cursor getCursor() {
    return null; // no queremos cambiar el cursor
  }
}
```

Como podemos ver, la función *point* obtiene el punto seleccionado por medio de la función *getPoint* del objeto "event" que recibe por parámetro. Observad el uso de la palabra clave *implements* en la declaración de la clase, seguida del nombre de la clase (o clases) que implementa (en este caso, "PointListener"). Observad también el uso de la fórmula *throws BehaviourException* en la declaración de la función *point*, que permite al código indicar un estado de excepción*.

* Las excepciones son un mecanismo muy potente de control de errores. Sin embargo, su estudio va más allá de los objetivos de esta asignatura.

Implementar la interfaz "PointListener" capacita a la clase para recibir los clics del ratón, pero esto no significa que los reciba de inmediato: es necesario de-

clarar la voluntad de recibir dichos eventos. Esto se lleva a cabo mediante la función *addMapTool* de la clase "MapControl":

```
controlMapa.addMapTool("Construcciones",
  new PointBehavior(this));
```

En el ejemplo, "controlMapa" es un objeto de la clase "MapControl". Lo que hacemos realmente es registrar nuestra clase como una herramienta de la ventana de vista que actúa en respuesta a los clics del ratón. Como se puede observar, la función *addMapTool* espera dos parámetros, el nombre de la herramienta y un objeto "PointBehavior", que es el que representa el comportamiento de la extensión. De este segundo parámetro, que no variará de una extensión a otra, sólo hay que destacar el uso de la palabra *this* para referirse al objeto actual.

Para activar la herramienta, utilizaremos la sentencia siguiente:

```
controlMapa.setTool("Construcciones");
```

Una vez hecho esto, cualquier evento relacionado con un punto del mapa será notificado a nuestra extensión.

1.8.3. Obtener un elemento del mapa

Obtenidas las coordenadas de un punto, nos interesa averiguar qué elemento del mapa se encuentra en el punto seleccionado. Normalmente encontraremos alguno de los elementos que hemos incorporado al mapa (líneas y polígonos), aunque es posible que no hallemos ninguno si el usuario ha hecho clic sobre un área vacía.

La consulta se realiza mediante la función *queryByPoint* de la clase "FLyrVect":

```
elementos = (FLyrVect)capa.queryByPoint(puntoGeo,
  mapControl.getViewPort().toMapDistance(1));
```

donde "puntoGeo" es un objeto de la clase "Point2D" y "capa" uno de la clase "FLyrVect", que representan el punto y la capa que se desea consultar respectivamente. La función devuelve un objeto "FBitSet" que contiene los datos asociados al elemento encontrado.

La función *queryByPoint* espera un segundo parámetro que es el radio de la consulta. En nuestro caso, consideramos suficiente la distancia equivalente a un píxel* del *viewport*. Como la función espera una distancia real, usamos la

*** Un píxel es la unidad mínima de representación de un dispositivo gráfico. Corresponde a un punto de color.**

función *toMapDistance* del *viewport* para trasladar la distancia de un píxel a su equivalente en la realidad.

Los datos obtenidos los recuperaremos mediante la función *getFieldValue* del objeto devuelto. Para determinar si hay datos y, en consecuencia, si la consulta ha encontrado algún elemento en la posición indicada, utilizaremos la función *isEmpty*.

Dado un objeto "capa" de la clase "FLyrVect" y un objeto "elementos" de la clase "FBitSet", el código siguiente:

```
DataSource fuenteDatos;
Value id, descripcion;

if (!elementos.isEmpty()) { // si hay datos
  // obtiene el objeto que contiene los datos
  fuenteDatos = ((AlphanumericData)capa).getRecordset();

  // empieza la lectura de los datos
  fuenteDatos.start();

  // lee el atributo "ID"
  id = fuenteDatos.getFieldValue(elementos.nextSetBit(0),
    fuenteDatos.getFieldIndexByName("ID"));

  // lee el atributo "Descripcion"
  descripcion = fuenteDatos.getFieldValue(
    elementos.nextSetBit(0),
    fuenteDatos.getFieldIndexByName("Descripcion"));

  // finaliza la lectura de los datos
  fuenteDatos.stop();
}
```

obtiene los valores de los campos "ID" y "Descripcion" del elemento encontrado y los almacena en las variables "id" y "descripcion" respectivamente. Observad que la función *getFieldValue* espera dos parámetros: la fila y la columna del elemento que se desea leer. Debemos imaginarnos "fuenteDatos" como una tabla en la que las filas son los elementos del mapa y las columnas, sus atributos. La fila la obtenemos del objeto "elementos" mediante la función *nextSetBit*, que nos devuelve la fila del primer objeto encontrado. Por su parte, la columna la obtenemos mediante la función *getFieldIndexByName*, que nos devuelve el número de columna de un atributo a partir de su nombre*.

*** El número de columna de los atributos depende del orden en el que se hayan definido. En el ejemplo, se definen los atributos "ID" y "Descripcion" por este orden. En consecuencia, el atributo "ID" estará en la columna 0, mientras que el atributo "Descripcion" estará en la columna 1.**

Juntándolo todo, el código de la extensión nos queda así:

```
package edu.uoc.postgradosig.construcciones;

import java.awt.Cursor;
import java.awt.geom.Point2D;

import javax.swing.JOptionPane;

import com.hardcode.gdbms.engine.data.DataSource;
import com.hardcode.gdbms.engine.values.Value;
import com.hardcode.gdbms.engine.values.ValueFactory;
import com.iver.andami.PluginServices;
import com.iver.andami.plugins.Extension;
import com.iver.andami.ui.mdiManager.IWindow;
import com.iver.cit.gvsig.fmap.MapControl;
import com.iver.cit.gvsig.fmap.ViewPort;
import com.iver.cit.gvsig.fmap.core.FPolygon2D;
import com.iver.cit.gvsig.fmap.core.FPolyline2D;
import com.iver.cit.gvsig.fmap.core.FShape;
import com.iver.cit.gvsig.fmap.core.GeneralPathX;
import com.iver.cit.gvsig.fmap.drivers.ConcreteMemoryDriver;
import com.iver.cit.gvsig.fmap.layers.FBitSet;
import com.iver.cit.gvsig.fmap.layers.FLayer;
import com.iver.cit.gvsig.fmap.layers.FLyrVect;
import com.iver.cit.gvsig.fmap.layers.LayerFactory;
import com.iver.cit.gvsig.fmap.layers.layerOperations.
  AlphanumericData;
import com.iver.cit.gvsig.fmap.tools.BehaviorException;
import com.iver.cit.gvsig.fmap.tools.Behavior.PointBehavior;
import com.iver.cit.gvsig.fmap.tools.Events.PointEvent;
import com.iver.cit.gvsig.fmap.tools.Listeners.
  PointListener;
import com.iver.cit.gvsig.project.documents.view.gui.View;

public class Construcciones extends Extension
  implements PointListener {
  private FLayer capa;

  public void accionMenu() {
    ConcreteMemoryDriver mapa;
    GeneralPathX ruta;
    FShape edificio, camino;
    MapControl controlMapa;
    View vistaActual;
```

```
    // crea un mapa vectorial en memoria
    mapa = new ConcreteMemoryDriver();
    mapa.setShapeType(FShape.LINE + FShape.POLYGON);

    // define los atributos de los elementos
    mapa.getTableModel().setColumnIdentifiers(new String[] {
      "ID", "Descripcion"});

    // agrega el edificio al mapa
    ruta = new GeneralPathX();
    ruta.moveTo(356270.7, 4727507.1);
    ruta.lineTo(356273.0, 4727503.3);
    ruta.lineTo(356278.5, 4727505.8);
    ruta.lineTo(356275.5, 4727509.8);
    ruta.lineTo(356270.7, 4727507.1);

    edificio = new FPolygon2D(ruta);
    mapa.addShape(edificio, new Object[] {
      ValueFactory.createValue(1),
      ValueFactory.createValue("Casa unifamiliar.") });

    // agrega el camino al mapa
    ruta = new GeneralPathX();
    ruta.moveTo(356242.7, 4727498.8);
    ruta.lineTo(356268.0, 4727509.8);
    ruta.lineTo(356281.0, 4727519.1);

    camino = new FPolyline2D(ruta);
    mapa.addShape(camino, new Object[] {
      ValueFactory.createValue(2),
      ValueFactory.createValue("Camino de herradura.") });

    // crea la capa...
    vistaActual = (View)PluginServices.getMDIManager().
      getActiveWindow();
    controlMapa = vistaActual.getMapControl();

    capa = LayerFactory.createLayer("Construcciones", mapa,
      controlMapa.getProjection());
    capa.setVisible(true);

    // ... y la agrega a la vista actual
    controlMapa.getMapContext().getLayers().addLayer(capa);
  }

  public void accionBoton() {
    View vistaActual;
    MapControl controlMapa;
```

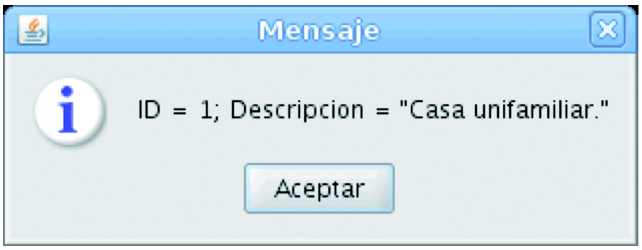
```
    // obtiene la vista actual
    vistaActual = (View) PluginServices.getMDIManager().
      getActiveWindow();
    controlMapa = vistaActual.getMapControl();

    // registra la escucha
    controlMapa.addMapTool("Construcciones",
      new PointBehavior(this));
    controlMapa.setTool("Construcciones");
  }

  public void execute(String actionCommand) {
    // realiza una accion u otra dependiendo de la opción
    // de menú seleccionada
    if (actionCommand.equals("MENU_CONSTRUCCIONES")) {
      accionMenu();
    }
    else if (actionCommand.equals("BOTON_CONSTRUCCIONES")) {
      accionBoton();
    }
  }

  public boolean isEnabled() {
    IWindow ventanaActiva;

    ventanaActiva = PluginServices.getMDIManager().
      getActiveWindow();

    // devuelve true en caso de que ventanaActiva sea de la
    // clase View
    return ventanaActiva instanceof View;
  }

  public void initialize() {

  }

  public boolean isVisible() {
    return true;
  }

  public void point(PointEvent event)
    throws BehaviorException {
    View vistaActiva;
    Point2D puntoGeo;
    ViewPort viewport;
    FBitSet elementos;
    MapControl controlMapa;
```

```
    DataSource fuenteDatos;
    Value id, descripcion;

    // obtiene el viewport
    vistaActiva = (View)PluginServices.getMDIManager().
      getActiveWindow();
    controlMapa = vistaActiva.getMapControl();
    viewport = controlMapa.getViewPort();

    // obtiene el punto en el que se ha producido el clic
    puntoGeo = viewport.toMapPoint(event.getPoint());
    try {
      elementos = ((FLyrVect) capa).queryByPoint(puntoGeo,
        viewport.toMapDistance(1));

      if (!elementos.isEmpty()) { // si hay datos
        // obtiene el objeto que contiene los datos
        fuenteDatos = ((AlphanumericData)capa).
          getRecordset();

        // empieza la lectura de los datos
        fuenteDatos.start();

        // lee el atributo "ID"
        id = fuenteDatos.getFieldValue(
          elementos.nextSetBit(0),
          fuenteDatos.getFieldIndexByName("ID"));

        // lee el atributo "Descripcion"
        descripcion = fuenteDatos.getFieldValue(
          elementos.nextSetBit(0),
          fuenteDatos.getFieldIndexByName("Descripcion"));

        // finaliza la lectura de los datos
        fuenteDatos.stop();

        // muestra los datos
        JOptionPane.showMessageDialog(null, "ID = " + id +
          "; Descripcion = \"" + descripcion + "\"");
      }
    }
    catch (Exception e) {

    }
  }

  public void pointDoubleClick(PointEvent arg0)
    throws BehaviorException {
    // no hace nada
  }
```

```
  public boolean cancelDrawing() {
    return false; // no nos interesa cancelar el dibujado
  }

  public Cursor getCursor() {
    return null; // sin cursor
  }
}
```

Hemos de destacar la presencia de un bloque *try – catch* en la función *point*. Este bloque es necesario para gestionar las excepciones que se pueden producir durante la llamada a las funciones *getRecorset*, *start*, *getFieldValue* y *stop*.

Si se selecciona la herramienta y se hace clic sobre el edificio que hemos creado, obtendremos el resultado siguiente:



Resumen

En este módulo hemos aprendido a extender una herramienta SIG para adaptarla a nuestras necesidades. Hemos aprendido a crear una capa con información vectorial y a rellenarla con elementos georreferenciados. Además, hemos visto cómo asociar datos a los elementos del mapa y cómo recuperarlos más tarde por medio de un simple clic del ratón.

Aunque en los ejemplos hemos trabajado exclusivamente con datos vectoriales almacenados en el código de la extensión, con unos pocos cambios estos se pueden leer de un fichero o de cualquier otra fuente de datos externa. Del mismo modo, las posibilidades de extensión de gvSIG van más allá de la representación estática de datos vectoriales, pudiendo representar datos a través de iconos o elementos en movimiento. En este sentido, las posibilidades que nos ofrece un lenguaje como Java, combinadas con la flexibilidad de gvSIG, son casi infinitas.

Bibliografía

Victor Acevedo, *"Guía de referencia para gvSIG 1.1"*, http://www.gvsig.org/web/docdev/reference/.

Francisco José Peñarrubia, *"Manual para desarrolladores gvSIG v1.1"*, http://www.gvsig.org/web/docdev/manual-para-desarrolladores-gvsig/.

Asociación gvSIG, Código fuente (Javadoc) de la versión 1.10 de gvSIG, http://www.gvsig.org/web/projects/gvsig-desktop/official/gvsig-1.10/.

Andres Herrera, *"Escribir un PlugIng para gvSIG "from scratch" en M$-Windows"*, http://t763rm3n.googlepages.com/PlugIng_gvSIG-fromScratch_Borrador.pdf.

Jorge Piera, *"Crear una extensión desde 0 en gvSIG"*, http://forge.osor.eu/docman/view.php/89/362/Crear_extensiones_en_gvSIG.pdf.