IBM ILOG JViews TGO V8.6

Getting started

© Copyright International Business Machines Corporation 1987, 2009
US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Copyright

Copyright notice

© Copyright International Business Machines Corporation 1987, 2009.

US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Trademarks

IBM, the IBM logo, ibm.com, WebSphere, ILOG, the ILOG design, and CPLEX are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at *http://www.ibm.com/legal/copytrade.shtml*

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Notices

For further copyright information see ***<installdir>*** **`/license/notices.txt`**.

Table of contents

Meeting IBM® ILOG® JViews TGO....5
What is IBM® ILOG® JViews TGO?....6
Graphic components....7
Business objects and data sources....11
Look and feel....13
Cascading style sheets....15
Tutorial: getting started with IBM® ILOG® JViews TGO....17
Running the tutorial....19
Executing the tutorial....20
Creating a basic network component....22
Creating a basic tree....25
Creating a basic table....28
Configuring the network component....30
Using custom business objects....33
Updating existing objects....37
Showing equipment details....40
Adding interactors....44
Handling selection....48

© Copyright IBM Corp. 1987, 2009

Index..53

Meeting IBM® ILOG® JViews TGO

Describes the basic concepts that underlie the development of JViews TGO GUI applications.

In this section

What is IBM® ILOG® JViews TGO?
Introduces the main features of IBM® ILOG® JViews TGO.

Graphic components
Introduces the four types of ready-to-use graphic components that are provided with JViews TGO and that are all based on the Model-View-Controller (MVC) architecture.

Business objects and data sources
Introduces the role of business objects and data sources.

Look and feel
Describes the default and custom look and feel of JViews TGO.

Cascading style sheets
Describes how the CSS mechanism is used in JViews TGO.

© Copyright IBM Corp. 1987, 2009

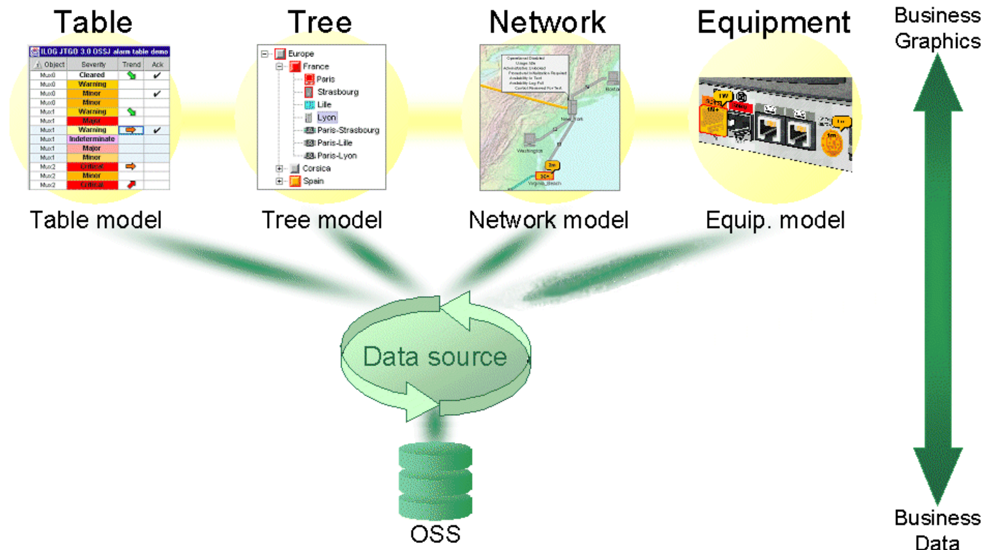
What is IBM® ILOG® JViews TGO?

IBM® ILOG® JViews TGO is a set of Java™ components for rapidly developing high-performance, extendable GUIs for Operation Support Systems (OSS) that seamlessly integrate with back-end applications written in XML or based on JavaBeans™.

JViews TGO supplies a complete range of ready-to-use, highly customizable, Java graphic components and styling services for the display of domain-specific data with a unified, consistent look and feel across a variety of representations: table, tree, network, and equipment.

JViews TGO takes data from a variety of back-end sources, typically Operation Support Systems (OSS) applications, and transforms this data into consistent, high-quality graphics that display across various graphic components. JViews TGO can be integrated with almost any back-end application that is capable of exporting data. For example, the data can be obtained from an XML stream, as Java objects, from a relational database, through a Corba interface, or in other ways.

JViews TGO provides four types of predefined graphic component: network, equipment, table, and tree. Graphic components connect to the back end through a data source that transforms data into business objects. Graphic components render these objects graphically by retrieving associated graphic properties (such as foreground or background color, font, line pattern) from a style sheet. One or several style sheets can be applied to the graphic components. The style sheets control the data-to-graphics mapping through rules that conform to CSS2 syntax.

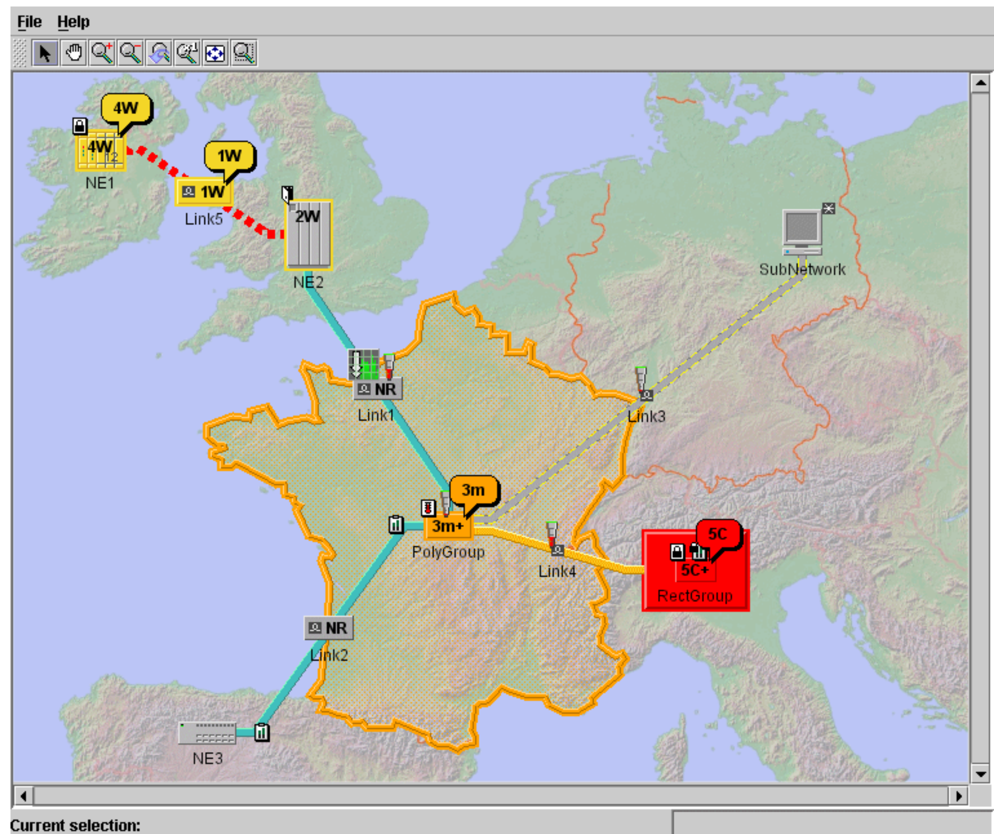


Graphic components

JViews TGO graphic components decouple back-end application objects from their graphic representations while providing powerful mapping capabilities to translate application objects and data into high-quality graphics. These components come fully-fledged with comprehensive built-in behavior and interactors. They benefit from runtime access to high-level functionality such as filtering, sorting, and search.

The network component

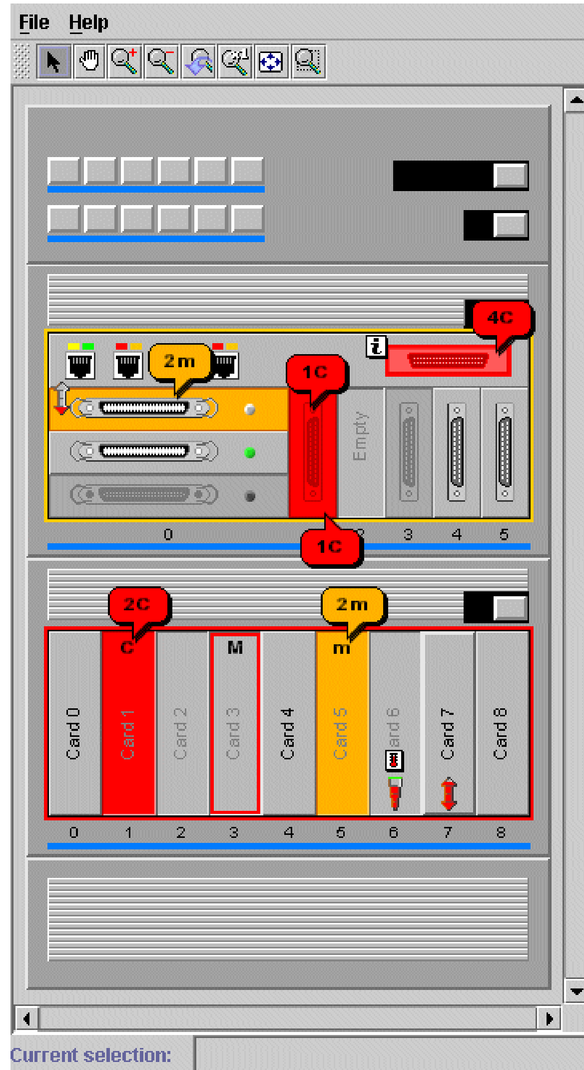
The network component is based on the IBM® ILOG® JViews grapher. It shows network nodes interconnected by links. The network component has support for editing the network, navigation, automatic layout of nodes and links, and background maps. It also supports pop-up menus and tooltips.



The network component can be configured either through an XML file, where you define all its associated settings (map displayed in the background, display of toolbar or overview window, zoom policy, and so on), or through an API.

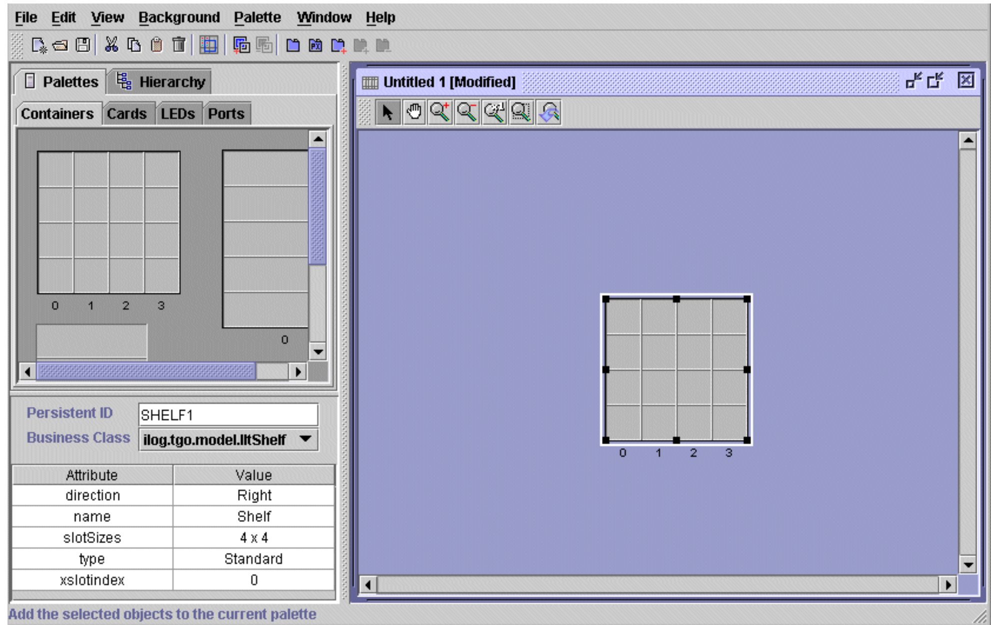
The equipment component

Like the network component, the equipment component is based on IBM® ILOG® JViews. It allows you to display items of equipment such as cards, shelves, ports, and LEDs. It also supports pop-up menus and tooltips.



Like the network component, the equipment component can be configured either through an XML file, where you define all its associated settings (map displayed in the background, toolbar, and so on), or through an API.

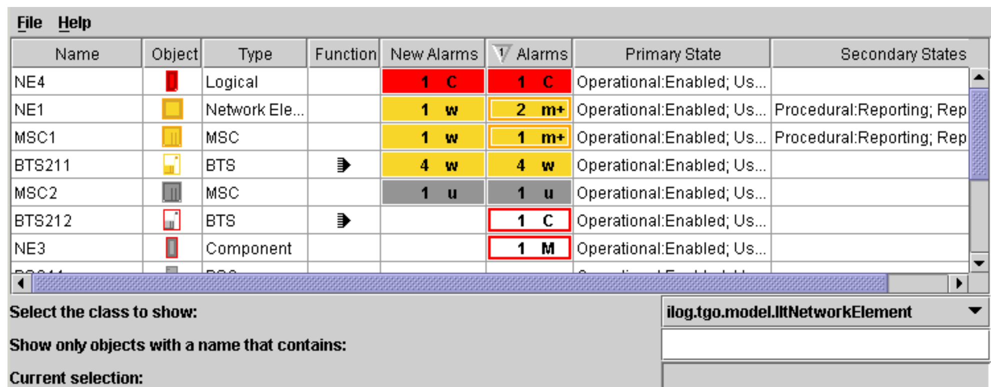
JViews TGO includes an equipment editor, a graphical user interface (GUI), that allows you to build an equipment component in a very easy and user-friendly manner.



The table component

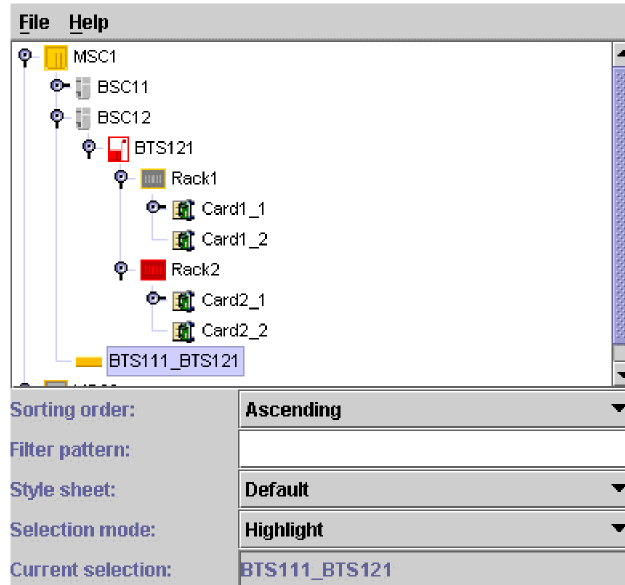
The table component is based on the Swing table component. It allows you to display data in a two-dimensional table format. Business objects are displayed in table rows, while their associated properties appear in separate columns.

The table component features smart resizing modes, multiple selection, and sorting, as well as filtering and searching capabilities. It also supports pop-up menus and tooltips.



The tree component

The tree component is based on the Swing tree component. It allows you to display data in a hierarchical representation. It features an efficient tiny look and feel, smart selection modes, sorting capabilities, and load-on-demand.



Business objects and data sources

JViews TGO graphic components communicate with the back-end application from which they obtain data to be displayed through a data source. The role of the data source is to transform data retrieved from the back end to objects that JViews TGO can handle. These objects, known as business objects, can be represented in any of the JViews TGO graphic components.

Business model and business classes

Business objects are instances of business classes, which are described by a business model. The business model translates back-end data into classes that JViews TGO can easily manipulate. It describes inheritance between these classes, along with their associated attributes.

Business classes and their instances are dynamic, which means that you can modify them and add new attributes at run time. As a consequence, you do not have to recompile your application for modifications to be taken into account.

Business classes can be defined directly in XML.

Data sources

Data sources are connecting objects that form a bridge between the back end and the front-end application or GUI. They transform data retrieved from the back end into business objects that will then be rendered as graphic objects at the level of the graphic components.

JViews TGO provides a default implementation of the data source that directly plugs in to XML files or streams or to JavaBeans™. On the other side, the same data source can be connected to multiple graphic components, allowing you to have different views of the same data with a consistent appearance.

You can specialize the data source to connect to other types of back end, such as Java™ Naming and Directory Interface (JNDI), or any other kind of back-end application.

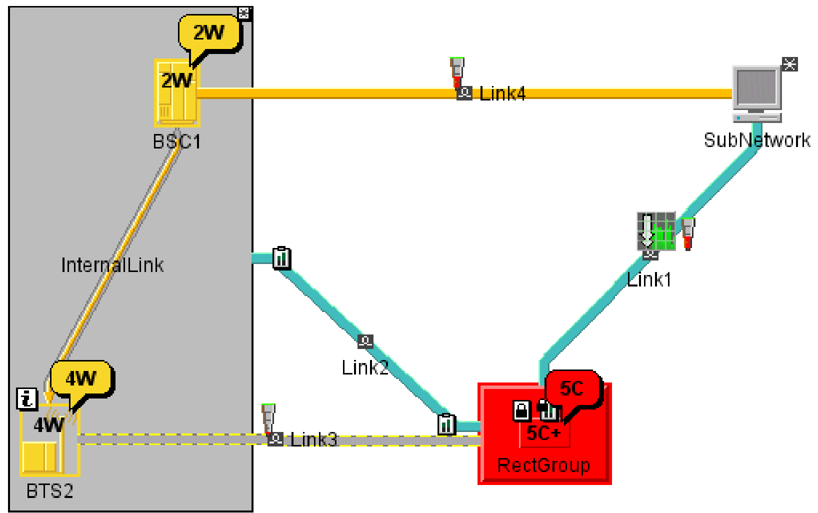
Predefined business objects

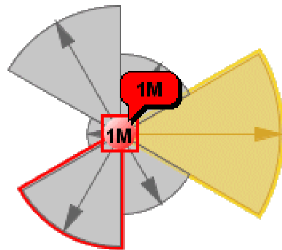
JViews TGO comes with a library of predefined business objects that are specifically designed to help you build, with a minimum of effort, high-quality and user-friendly user interfaces in the domain of telecom network management.

Examples of these objects are network elements, Base Transceiver Stations (BTS), links, cards, shelves, ports, and many more. JViews TGO also supplies a complete library of graphic symbols, icons, and decorations for representing changes in telecommunication business object states and alarms. State and alarm representations comply with the most widespread telecommunication standards, such as OSI, Bellcore, SONET, and SNMP.

Predefined business objects can be defined either in an XML file or through a specific API. You simply add these objects to the data source to display them with a homogeneous, high-quality look and feel and with associated decorations (such as alarm balloons or status icons) across all the predefined JViews TGO components.

The following figures show a few examples of the way some of the JViews TGO predefined business objects are graphically rendered in components.


2W
2W
BSC1
Link4
SubNetwork
InternalLink
Link1
Link2
4W
4W
Link3
5C
5C+
RectGroup
BTS2


1M
1M

Look and feel

JViews TGO is able to turn business data into specialized graphical representations. High-level views, such as tree, topology, chassis or table views, are automatically populated with graphic objects whose icons, decorations and annotations are driven by business-specific logic. However, the appropriate representation can differ a lot, depending on the context and purpose of the display. One can distinguish two main domains: Network Management and Network Visualization. JViews TGO offers both ready-to-use and highly configurable solutions to translate information into pictures.

Default look and feel

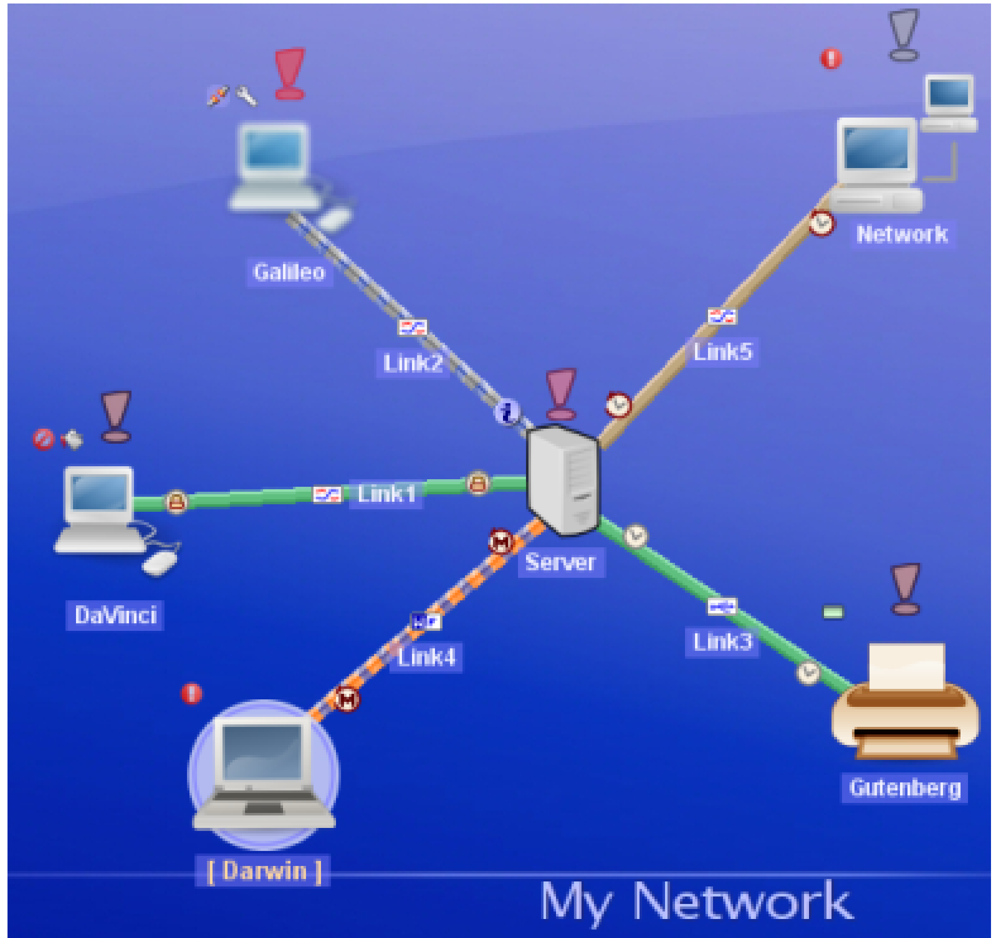
The default look and feel, which comes natively with JViews TGO, is intended to be used by professional network managers. It includes extremely precise ways to display interconnected assets (such as nodes, networks and equipments), and management information (such as alarms and states). The underlying look and feel is based on international and industrial standards, as well as the long-term expertise of IBM® ILOG® in this domain. For example, a single node usually aggregates a lot of information such as its name, type, function, family, operational state, different kinds of alarms, primary and secondary states, and so on. Based on various preferences and settings specified by the programmer, JViews TGO is able to automatically build an appropriate visual representation. The aim is to display as much relevant information as possible while, at the same time, reducing the risk of mistakes. Here is a typical example of the default look and feel:

This look and feel is often too precise for pure network visualization needs. For this reason, JViews TGO offers other flexible ways to represent business information.

Custom look and feel

For domains where the graphical representation should be less constrained and possibly more attractive, JViews TGO offers different ways to enrich the default look and feel. For example, it is very easy to define new icons for some kinds of node or state, or to define a specific representation for a given router when it does not work properly. In contrast with demanding network management activities, this is particularly suitable for network and asset visualization for which there is less precise information to display but in a more attractive way. This type of configuration can be performed through Java™ code using the

open API of JViews TGO, or through configuration files based on the Cascading Style Sheets (CSS) syntax. So it is possible to describe the mapping between information and its desired visual representation, and allow developers to reuse and share the newly-defined look and feel between many different applications. Here is a typical example of a custom look and feel:



For a working example, run the Customizing Business Objects Sample located in ***<installdir>* /samples/integration/customization**.

Cascading style sheets

Style sheets consist of collections of graphic settings, such as color, font, and icon, that are used to render objects and associated attributes in graphic components. Cascading Style Sheets (CSS) provide a powerful mechanism for customizing HTML rendering in a Web browser. The CSS specification originates from the World Wide Web Consortium (W3C) and has the status of a W3C Recommendation.

The CSS mechanism is a great improvement over the `.Xdefault` resource mechanism of the X Window System. The basic idea remains the same: matching a pattern and setting resource values. The CSS language is intended for HTML rendering: matching HTML tags, and setting style values. XML is also a CSS target, especially in the context of the Scalable Vector Graphic (SVG) specification of W3C.

In JViews TGO, the CSS level 2 (CSS2) Recommendation is transposed for the Java™ language and used to set Bean properties according to the Java object hierarchy and state.

JViews TGO graphic components use CSS to customize the object rendering as well as the graphic component configuration. JViews TGO supplies a large number of predefined ready-to-use CSS properties that apply to both predefined and custom business classes, objects, and attributes of these objects. The predefined representation can be fully customized to achieve the graphic representation that best suits your needs.

Tutorial: getting started with IBM® ILOG® JViews TGO

Steps you through the typical tasks that you want to do with JViews TGO.

In this section

Running the tutorial
Describes the start up conditions.

Executing the tutorial
Describes how you initialize JViews TGO and store the context, and how to execute the tasks.

Creating a basic network component
Shows you how to create a network component to model and represent a telecommunications network.

Creating a basic tree
Shows you how to create a tree component to show the containment hierarchy

Creating a basic table
Shows you how to create a table component.

Configuring the network component
Describes how to configure the background, behavior, and other aspects of the network component.

Using custom business objects
Shows you how to read in a file containing alarms.

© Copyright IBM Corp. 1987, 2009

Updating existing objects

Shows you how to read in a file containing data for updating and removing objects.

Showing equipment details

Shows you how to create a dialog for showing equipment details.

Adding interactors

Shows you how to configure a pop-up menu that is consistent across all the components, how to add double-click behavior for network elements and how to show an overview window with a specific keystroke.

Handling selection

Explains how to handle the tree selection and the single selection model.

Running the tutorial

This sample application is used as a tutorial to show you how to create some graphic components, configure a network component, use custom business objects, update objects, create a dialog to show equipment details, add some interactors, and handle selection.

This tutorial is available online at **<installdir>/tutorials/gettingStarted** where `<installdir>` is the directory where you have installed JViews TGO.

If you click the file name **<installdir>/tutorials/gettingStarted**, you will display information about the tutorial, including how to install and run it.

Executing the tutorial

Describes how you initialize JViews TGO, store the context, and execute the different tasks. It assumes that you are familiar with writing Java™ code. Therefore, it does not describe the Java import statements or data member declarations that are used to set up the application environment for this tutorial.

Initializing JViews TGO

To run the tutorial, you first have to specify a container or frame to contain what is to be displayed and initialize JViews TGO by reading a deployment descriptor file, `deploy.xml`, to initialize the application context:

```
void doSample (Container container) {
  try {
    if (isApplet())
      IltSystem.Init(this, "deploy.xml");
    else
      IltSystem.Init("deploy.xml");
```

In this code extract, `container` is a `java.awt.Container` class.

For more information about the content and purpose of the deployment descriptor file, see The application context.

Executing the steps in the tutorial

The remainder of this code refers to the main steps of the tutorial, which are described in detail in the rest of this section. These steps are called and executed in sequence. Finally, the code catches and logs any errors.

```
    // Create a simple network component
    step1();

    // Create a simple tree
    // Add subnodes to show containment in tree and network
    step2();

    // Create a simple table component
    step3();

    // Configure the network component
    step4();

    // Load alarms and show them in a separate table
    step5(container);

    // Update and remove objects
    step6();
```

```
      // Create a new window to show details of equipment
      step7();

      // Interactors:
      // - configure consistent pop-up menu on all components
      // - add double-click behavior for network elements
      // - show overview window on special keystroke
      step8();

      // Synchronize selection between components
      step9();
    }
    catch(Exception e){
      log.severe("Exception caught while running sample: " +
                  e.getMessage());
      e.printStackTrace();
    }
  }
```

Creating a basic network component

Shows you how to create a network component to model and represent a telecommunications network. The data for this part of the tutorial is in the file:

<installdir>/tutorials/gettingStarted/data/regions.xml.

To allow the network component to function, you must create a data source to supply the business data in the form of JViews TGO business objects. See Data sources.

You can then create a network component and connect it to the data source. See Creating a network component: a sample .

Next you read in the data from the XML file.

Finally, you add the network component to the container that you defined during initialization. This delimits the physical area in which the network view will be displayed.

This part of the code is referred to as Step 1.

```
void step1() throws Exception {
```

1. Create the data source.

   ```
     mainDataSource = new IltDefaultDataSource();
   ```

   The data source will contain the business objects used in the application. JViews TGO supplies `IltDefaultDataSource` as a predefined data source. The new instance of `IltDefaultDataSource` created here is called `mainDataSource`. This data source is not specific to the network component and will be used to supply business objects to other graphic components.

2. Create the network component.

   ```
   networkComponent=new IlpNetwork();
   ```

   JViews TGO supplies `IlpNetwork` as a predefined network component. The new instance of `IlpNetwork` that you create is called `networkComponent`.

3. Connect the data source and the network component.

   ```
   networkComponent.setDataSource(mainDataSource);
   ```

   Once the data source is set, the network component is notified of any modifications to the business objects contained in the data source. Changes include adding or removing objects, or changing the attribute values or structure of an object.

4. Read the XML file `regions.xml` containing the network business objects.

   ```
   mainDataSource.parse("regions.xml");
   ```

The data source you have just created reads in an XML file. The method used is `parse (org.xml.sax.InputSource)`.

You can pass as an argument to this method either a filename or a valid `org.xml.sax.InputSource` that you have previously created.

The data in the file adds three business objects, `Britain` and `Benelux` and `France`. They belong to the business object class `IltPolyGroup` and have the attributes `name` and `position`. A position must be an instance of a class that implements `IlpPosition`. Here, the type is `IlpPolygon` which contains the (x,y) coordinates of the points that define the shape of the region.

For example:

```
<addObject id="Britain">
  <class>ilog.tgo.model.IltPolyGroup</class>
  <attribute name="name">Britain</attribute>
  <attribute name="position" javaClass="ilog.cpl.graphic.views.IlpPolygon">

    <point>  <x>167.0</x>       <y>106.0</y> </point>
    <point>  <x>160.0</x>       <y>84.0</y>  </point>
    <point>  <x>196.0</x>       <y>78.0</y>  </point>
    <point>  <x>186.0</x>       <y>24.0</y>  </point>
    <point>  <x>142.0</x>       <y>18.0</y>  </point>
                            ...
    <point>  <x>191.0</x>       <y>104.0</y> </point>
  </attribute>
</addObject>
```

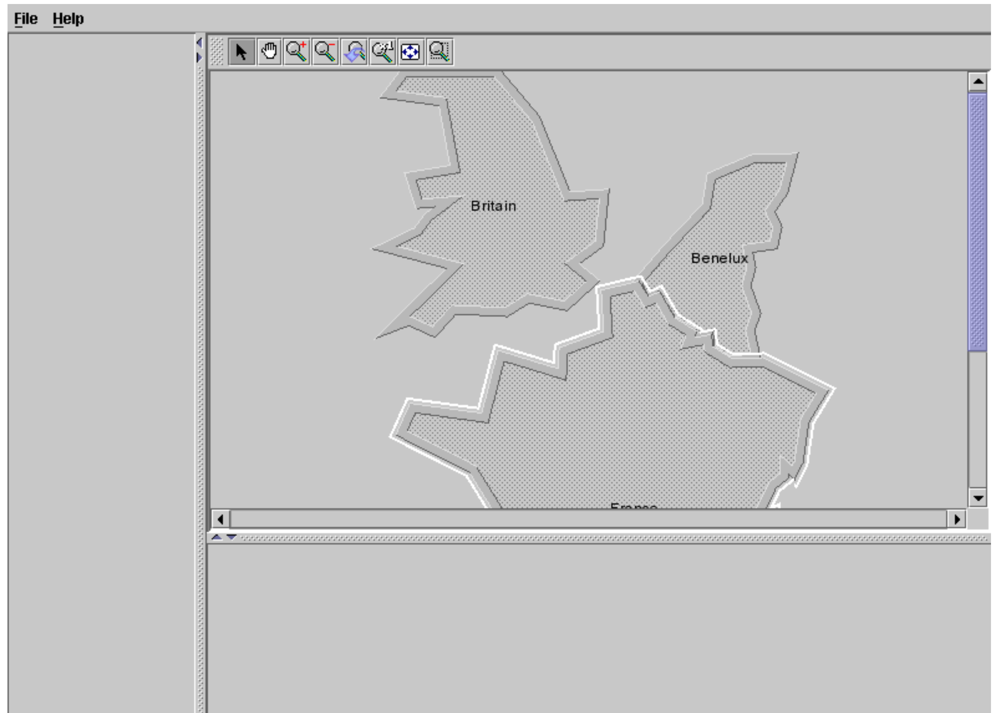
Note that all the data in an JViews TGO XML file must be defined in the element `<cplData>`. See Data sources .

5. Add the network component to the container or frame.

```
  networkArea.add(networkComponent, BorderLayout.CENTER);
}
```

This code causes the network component to be displayed in the corresponding area of the main frame.

The network component should look as shown in the following figure.


File
Help
Britain
Benelux

Creating a basic tree

Shows you how to create a tree component to show the containment hierarchy. The data for this part of the tutorial is in the file **<installdir>/tutorials/gettingStarted/data/elements.xml**.

To allow the tree component to function, you must create a tree component and connect it to a data source. This example uses the same data source that was created for the network component to show the same data in the tree and the network. A new XML data file is read into the data source. This file includes the child elements to be contained under the top level objects (`Britain` and `Benelux`) read in Step 1. See *Creating a basic network component*.

You must also add the tree component to the container or frame.

This part of the code is referred to as Step 2:

```
void step2() throws Exception {
```

To do Step 2:

1. Create the tree component.

   ```
   treeComponent = new IlpTree();
   ```

2. Connect the data source `mainDataSource` to the tree component.

   ```
   treeComponent.setDataSource(mainDataSource);
   ```

   The data source shows the tree with containment by default in line with the data that will be read in from the XML file.

3. Read in the data to the data source from the file `elements.xml`.

   ```
   mainDataSource.parse("elements.xml");
   ```

   The data concerns named Base Station Controller (BSC) and Base Station Transceiver (BTS) objects and a link between the BTS subnodes. This data is added to the main data in the data source.

   Note: Here top-level BTS nodes only are created. JViews TGO also supports a more complete representation of BTS nodes and their antennas through the class `IltBTS`.

   The BSC network elements, `Cardiff` and `London`, are both contained under `Britain`. The BTSs are child objects of the BSCs.

   Each node object belongs to the business class `IltNetworkElement`, which has the following attributes among others:

- `name`
- `type`
- `family`
- `function`
- `position`

  The position is not meaningful for the tree, but it is used to display the elements in the network component. The tutorial uses pixel (x,y) coordinates, as supported by the Java™ class `IlpPoint`. You can also give geographic positions as latitude and longitude values through the class `IlpGeographicPosition`. In this case, you need to provide a corresponding `IlpGeographicPositionConverter` to the network component. See Creating a network component: a sample in the *Graphic Components* documentation for details.

The following is an example of how a node is described in an JViews TGO XML data file:

```
<addObject id="London">
   <class>ilog.tgo.model.IltNetworkElement</class>
   <parent>Britain</parent>
   <attribute name="name">London</attribute>
   <attribute name="type">BSC_Image</attribute>
   <attribute name="position"> javaClass="ilog.cpl.graphic.IlpPoint">
     <x>269.0</x>
     <y>149.0</y>
   </attribute>
</addObject>
```

The links belong to the class `IltLink`. They are defined in terms of their start and end nodes. The links are named by their start and end node IDs. The start node is defined within a `<from>` element and the end node is defined within a `<to>` element. The file provides values for the following link attributes:

- `name`
- `media`

  The media defines the material of the link, such as fiber optic.

The following is an example of how a link is described in an JViews TGO XML data file:

```
<addObject id="BTS11_BTS22">
   <class>ilog.tgo.model.IltLink</class>
   <parent>Britain</parent>
   <attribute name="name">BTS11_BTS22</attribute>
   <link>
     <from>BTS11</from>
     <to>BTS22</to>
   </link>
```

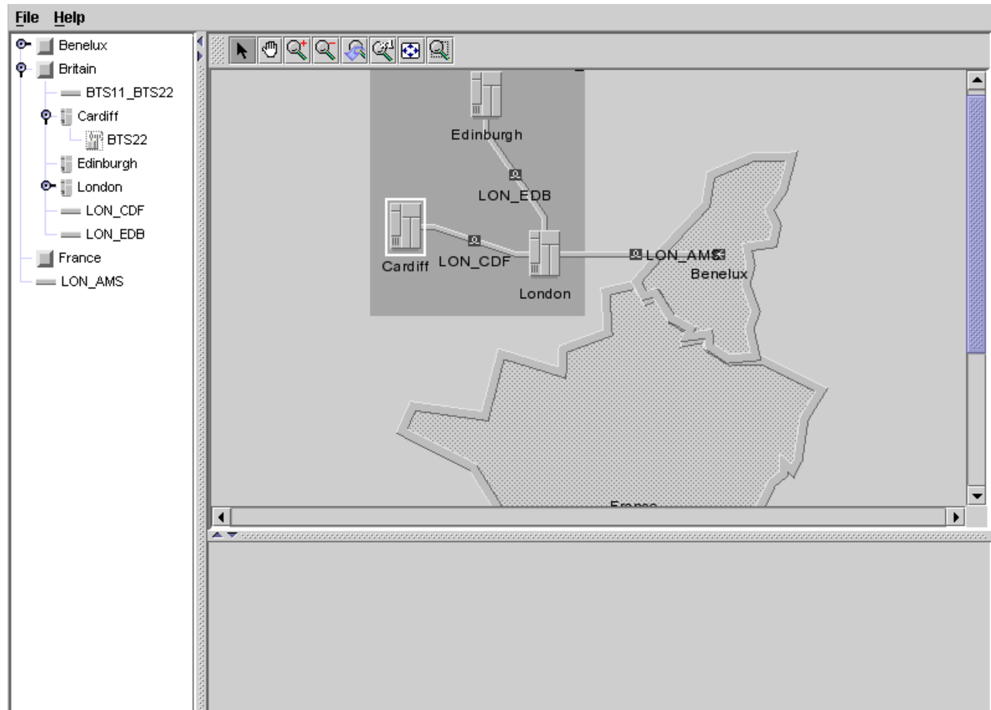
```
   <attribute name="media">Fiber</attribute>
</addObject>
```

4. Add the tree component to the container or frame.

```
  treeArea.add(treeComponent,BorderLayout.CENTER);
}
```

This causes the tree component to be displayed in the corresponding area of the main frame.

The sample with the tree component should look as shown in the following figure.



Creating a basic table

Shows you how to create a table component. The table in the sample uses the same data source as the network component. Unlike a tree or a network, a table can only meaningfully display objects of a single class or objects with a common base class. The reason is that the table can show only attributes (columns) that are common to all of the objects (rows) displayed.

In this example, the objects to be displayed are restricted by class to the network elements. Therefore, neither the regions added in Step 1 (*Creating a basic network component*) nor the alarms to be added later in Step 5 (*Using custom business objects*) will be shown in the table.

To allow this table component to function, you must create a table component, connect the data source to the table component, restrict the content of the table to network elements, and add the table to the container.

This part of the code is referred to as Step 3.

```
void step3() {
```

1. Create the table component.

   ```
   tableComponent = new IlpTable();
   ```

   JViews TGO supplies `IlpTable` as a predefined table component. The new instance of `IlpTable` that you create is called `tableComponent`.

2. Connect the data source to the table component and filter the objects to be put into the table.

   ```
   tableComponent.setDataSource(mainDataSource,
                               ilog.tgo.model.IltNetworkElement.GetIlpClass()
   );
   ```

   The objects to be displayed are restricted to the class `IltNetworkElement`. The method `GetIlpClass()` is called to access the dynamic business class corresponding to the `IltNetworkElement` Java™ class.

3. Add the table component to the container.

   ```
     tableArea.add(tableComponent, BorderLayout.CENTER);
   }
   ```

The sample with the table component should look as shown in the following figure.

File Help

- Benelux
- Britain
  - BTS11_BTS22
  - Cardiff
    - BTS22
  - Edinburgh
  - London
    - LON_CDF
    - LON_EDB
- France
- LON_AMS

Edinburgh
LON_EDB
Cardiff
LON_CDF
London
LON_AMS
Benelux
France

Obj...	Name	Type	Family	Funct...	New ...	Alarms	Primary State	Secondary States
	BTS11	BTS						
	BTS22	BTS						
	Brussels	BSC_Im...						
	London	BSC_Im...						
	Cardiff	BSC_Im...						

Configuring the network component

You can configure the background, behavior, and other aspects of the network component in a cascading style sheet file. See Configuring a network component through a CSS file in the *Graphic Components* documentation for detailed information on the content of this type of file. This section shows you how to read in one or more network configuration files.

Three network configuration files are loaded:

- **<installdir>/tutorials/gettingStarted/data/networkConfiguration.css**

  This file specifies the configuration of the default interactors. See Interacting with the network view in the *Graphic Components* documentation for details of the default network interactors.

- **<installdir>/tutorials/gettingStarted/data/networkBackground.css**

  This file specifies the name and type of a background map to load. See Background support in the *Graphic Components* documentation for details concerning background maps.

- **<installdir>/tutorials/gettingStarted/data/network.css**

  This file specifies the configuration of the objects displayed in this network component.

This part of the code is referred to as Step 4:

```
void step4() {
```

Read in the network configuration.

```
String[] css = new String[] { "network.css",
  "networkConfiguration.css", "networkBackground.css" };
try {
  networkComponent.setStyleSheets(css);
} catch (Exception e) {
  e.printStackTrace();
}
```

The names of the network configuration files are passed as string arguments of the `setStyleSheets` method of the network component class. This method reads a configuration into an existing network, replacing the previous configuration. See Configuring a network component through a CSS file in the *Graphic Components* documentation for a detailed description of the properties used in this type of file.

This example shows you how to configure the toolbar of the network component and how to modify the behavior of the interactors that it invokes. In the configuration file, the toolbar is defined by a number of buttons. The complete list of supported buttons can be found in the `ilog.cpl.network.action.toolbar` package; the names found in the CSS file correspond to the class name of the button, without the `IlpNetwork` prefix and with the `Button` suffix. When invoked, each button sets its associated interactor to the network component.

You can customize the behavior of each interactor through the CSS selectors that correspond to the toolbar buttons. Many Bean properties of the interactor class can be set in this way.

For a complete list, see the reference documentation of the toolbar button classes in the `ilog.cpl.network.action.toolbar` package.

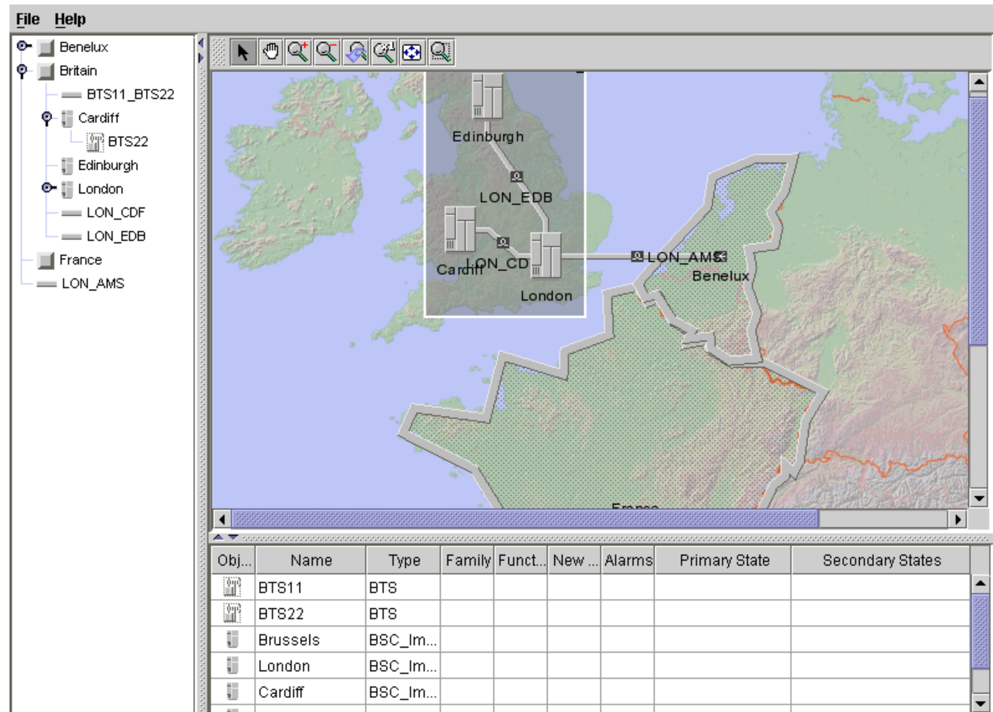
The `networkConfiguration.css` file is nearly the same as the default configuration of the network component toolbar, except that it disables the moving of nodes. Moving nodes is disabled by setting the `moveAllowed` property of the `Select` interactor to `false`. In real life, this feature could be useful for preventing accidental changes to the network layout. See Selection interactor in the *Graphic Components* documentation for more details.

The corresponding CSS file looks like this:

```
// Sample network configuration file

// This file copies the default toolbar configuration, with one
// modification: the moving of objects is disabled.

// See ilog.cpl.network.renderer package for additional options.

Network {
  toolbar: true;
  interactor: true;
}

ToolBar {
  enabled: true;
  button[0]: @+SelectButton;
  button[1]: @+PanButton;
  button[2]: @+ZoomInButton;
  button[3]: @+ZoomOutButton;
  button[4]: @+ZoomBackButton;
  button[5]: @+ZoomResetButton;
  button[6]: @+FitToContentsButton;
  button[7]: @+ZoomViewButton;
}

Subobject#SelectButton {
  actionType: "Select";
  usingObjectInteractor: true;
  opaqueMove: true;
  moveAllowed: false;
}

Subobject#PanButton {
  actionType: "Pan";
  usingObjectInteractor: false;
}

Subobject#ZoomInButton {
  actionType: "ZoomIn";
}

Subobject#ZoomOutButton {
  actionType: "ZoomOut";
}
```

```
Subobject#ZoomBackButton {
  actionType: "ZoomBack";
}

Subobject#ZoomResetButton {
  actionType: "ZoomReset";
}

Subobject#FitToContentsButton {
  actionType: "FitToContents";
  margins: "5";
}

Subobject#ZoomViewButton {
  actionType: "ZoomView";
  usingObjectInteractor: false;
}

Interactor {
  name: "Select";
}
```

The sample with a configured network component should look as shown in the following figure.



Using custom business objects

Shows you how to read in a file containing alarms, which are instances of a custom business class. Custom business classes are dynamic classes that you define for yourself, as opposed to those that are predefined in JViews TGO. They can be used to represent any type of business object.

The data for this part of the tutorial is in the file:

<installdir>/tutorials/gettingStarted/data/alarms.xml.

A custom style sheet is used to create an attractive display of the instances of the alarm class. For example, the style sheet defines the labels used in the table column headers, the background color of the table cells, and whether to use an icon instead of a string value.

A style sheet is read in from a CSS file. In the example, the alarm configuration is defined in the file:

<installdir>/tutorials/gettingStarted/data/alarm.css.

This CSS file is imported by the style sheet (CSS) file of each graphic component interested in the alarm business class. This is illustrated in the file

<installdir>/tutorials/gettingStarted/data/table.css by the following line:

```
@import "alarm.css"
```

Then the style sheet of the graphic component is loaded with the method `setStyleSheets`, as shown in *Configuring the network component*.

This part of the code is referred to as Step 5:

```
void step5(Container container) throws Exception{
```

To do Step 5:

1. Read in the file `alarms.xml` that contains the declaration of the custom dynamic class `Alarm`, and a number of its instances.

   ```
   mainDataSource.parse("alarms.xml");
   ```

   Note: You read the data into the same data source as used for all the previous data. The custom class declarations can also be read at startup time by declaring them in the deployment descriptor.

   The `Alarm` class is described as follows in the XML data file:

   ```
   <classes>
     <class>
   ```

```
    <name>Alarm</name>
    <attribute>
      <name>identifier</name>
      <javaClass>java.lang.String</javaClass>
    </attribute>
    <attribute>
      <name>perceivedSeverity</name>
      <javaClass>java.lang.Integer</javaClass>
    </attribute>
    <attribute>
      <name>acknowledged</name>
      <javaClass>java.lang.Boolean</javaClass>
    </attribute>
    <attribute>
      <name>creationTime</name>
      <javaClass>java.util.Date</javaClass>
    </attribute>
  </class>
</classes>
```

Each attribute takes its type from its Java™ class type, such as `java.lang.String` for the `id` attribute.

Instances of the `Alarm` class are defined by giving specific values to the attributes of the class. For example:

```
<addObject id="alarm1">
  <class>Alarm</class>
  <parent>London</parent>
  <attribute name="identifier">Alarm 1</attribute>
  <attribute name="perceivedSeverity">5</attribute>
  <attribute name="acknowledged">true</attribute>
  <attribute name="creationTime">2001-12-12T15:42:17</attribute>
</addObject>
```

The parent of the alarm instance, which is the object on which the alarm is set, is also given.

2. Create a new table.

```
alarmTableComponent = new IlpTable();
```

The name of the new instance of `IlpTable` is `alarmTableComponent`. This table will be used to display the alarms.

3. Get the `Alarm` class.

```
final IlpClass alarmClass = context.getClassManager().getClass("Alarm");
```

The class manager is defined by the interface `IlpClassManager`. It handles a hierarchy of business classes. See Business class manager API in the *Business Objects and Data*

Sources documentation for details. The application context allows you to retrieve the class manager service.

The method `getClass(java.lang.String)` returns the specified class.

4. Connect the data source to the table component and filter the objects to be put into the table.

   This table will be used to display instances of the `Alarm` class only.

```
alarmTableComponent.setDataSource(mainDataSource, alarmClass);
```

   The data source that will supply the alarms is set as `mainDataSource`. The class of business objects to be displayed from this data source is specified as `alarmClass`.

5. Create a tabbed pane at the bottom of the window and add both tables to it.

```
initTableTab(container, alarmTableComponent);
```

   This method takes both `tableComponent` and `alarmTableComponent` and places them in a tabbed pane. It is written in pure Swing code.

6. Create a filter for the tree, so that it shows critical and major alarms only.

   The `IlpFilter` interface is implemented as `treeAlarmFilter`. The method `accept` is used to test the acceptability of the objects offered to this filter. The method returns `true` if the filter accepts an object.

```
IlpFilter treeAlarmFilter = new IlpFilter(){
  public boolean accept (Object object){
    IlpObject ilpObject = (IlpObject)object;
    if (ilpObject.getIlpClass().equals(alarmClass)) {
      IlpAttribute perceivedSeverityAttr =
                   alarmClass.getAttribute("perceivedSeverity");
```

```
      Object severityValue =
```

```
            ilpObject.getAttributeValue(perceivedSeverityAttr);
```

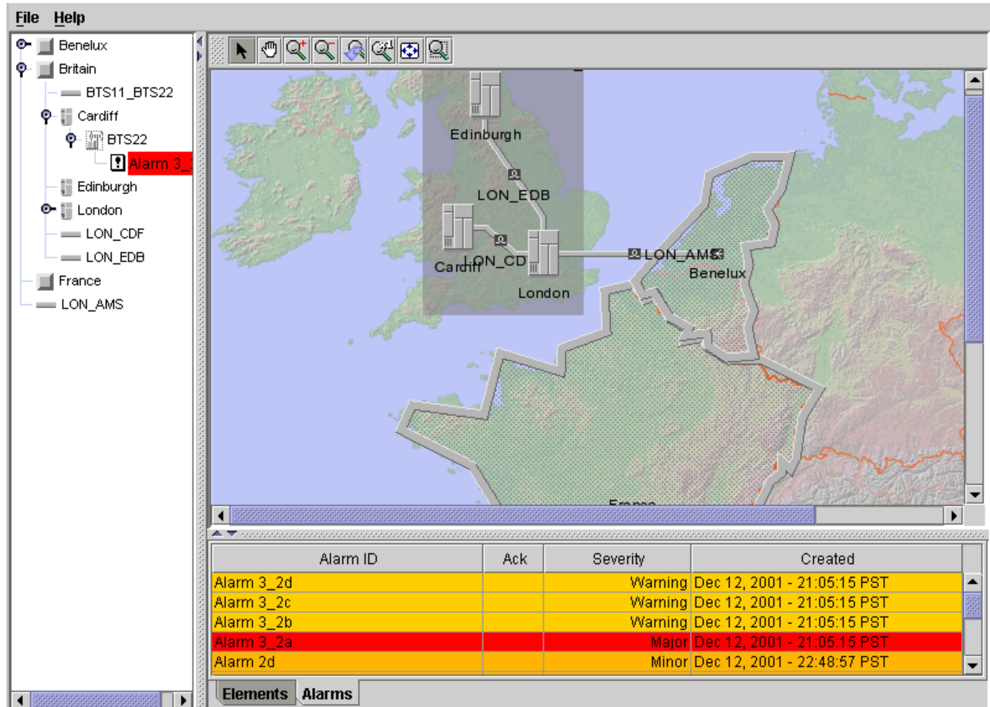
```
      return (new Integer(3).compareTo(alarmClass)<0);
    }
    return true;
  }
};
```

   The filter is refined to test the severity of the alarms. The `Alarm` class has the attributes `perceivedSeverity` and `acknowledged`. Only alarms with a severity greater than 3 and with the `acknowledged` attribute set to `false` will be included in the table.

   Then, the filter is set on the tree component:

```
  treeComponent.setFilter(treeAlarmFilter);
}
```

The sample should now look as shown in the following figure.



Updating existing objects

Shows you how to read in a file containing data for updating and removing objects. It also shows you how to update objects through the API.

The data for this part of the tutorial is in the file:
<installdir>/tutorials/gettingStarted/data/updates.xml.

This part of the code is referred to as Step 6.

```
void step6() throws Exception{
```

To do Step 6:

1. Read in the file containing the data on updating and removing objects.

   ```
   mainDataSource.parse("updates.xml");
   ```

   This file modifies some of the network elements described in the file `elements.xml`. The object to be updated is identified by its identifier. Then, the attribute to be updated is identified and the new values are given.

   For example:

   ```
   <updateObject id="BTS11">
     <attribute name="objectState"
                      javaClass="ilog.tgo.model.IltOSIObjectState">
       <state>
         <administrative>ShuttingDown</administrative>
         <operational>Enabled</operational>
         <usage>Active</usage>
       </state>
       <alarms>
         <new severity="Warning">4</new>
       </alarms>
       <procedural>Reporting</procedural>
       <repair>UnderRepair</repair>
       <performance state="Output">150</performance>
     </attribute>
   </updateObject>
   ```

   This XML description causes the given object state to be associated with the `BTS11` object.

   Another way to update business objects is through the API. The remaining steps show you how to do this.

2. Get a reference to an existing object from the data source.

   ```
   IltObject london = (IltObject)mainDataSource.getObject("London");
   ```

The Base Station Controller `london` is to be updated and is retrieved from the data source.

3. Create new state values for this object.

```
IltOSI.State osiState =
  new IltOSI.State(IltOSI.State.Operational.Enabled,
                   IltOSI.State.Usage.Idle,
                   IltOSI.State.Administrative.Locked);
```

The values `Enabled`, `Idle`, and `Locked` of specific OSI states are created for this object through inner classes of `IltOSI.State`. The value `Enabled` is attributed to `IltOSI.State.Operational`. The value `Idle` is attributed to `IltOSI.State.Usage`. The value `Locked` is attributed to `IltOSI.State.Administrative`

4. Create a new state, `objectState`, for this object with the primary state `osiState`.

```
IltOSIObjectState objectState = new IltOSIObjectState(osiState);
```

You create a new instance of the class `IltOSIObjectState`. An instance of this class represents the state of a telecom object as defined by the OSI SMF 10164-2 standard.

5. Add two alarms to the alarm state.

```
IltAlarm.State alarmState = (IltAlarm.State)objectState.getAlarmState();
alarmState.addNewAlarm(IltAlarm.Severity.Minor);
alarmState.addAcknowledgedAlarm(IltAlarm.Severity.Critical);
```

You retrieve the alarm state from `objectState` using the method `getAlarmState()`. `addNewAlarm(ilog.tgo.model.IltAlarmSeverity)` allows you to add an unacknowledged alarm with the severity `Minor`.

The method alarmState.`addAcknowledgedAlarm(ilog.tgo.model.IltAlarmSeverity)` allows you to add an acknowledged alarm with the severity `Critical`.

6. Update the state of the given object.

```
london.setObjectState(objectState);
}
```

Here, you assign the state `objectState` with its possible alarms to the object `london`. The data source and its attached graphic components are automatically notified of this change.

The sample should now look as shown in the following figure.

File Help

Benelux
Amsterdam
Brussels
Britain
BTS11_BTS22
Cardiff
BTS22
Alarm 3_
Edinburgh
London
LON_CDF
LON_EDB
France
LON_AMS

Edinburgh
1m
1C+
Cardiff
London
1m
Benelux
1C

Object	Name	Type	Family	Function	New Ala...	Alarms	Primary Stat
	BTS22	BTS			1 M+	1 M+	Operational:Enable
	London	BSC			1 m	1 C+	Operational:Enable
	BTS11	BTS			4 W	4 W	Operational:Enable

Elements Alarms

Showing equipment details

Shows you how to create a dialog for showing equipment details. To prepare for this dialog, you must:

- Create a new data source
- Create an equipment component
- Connect the equipment component to the data source
- Read an XML file containing the description of business objects into the data source
- Read a configuration file containing a background
- Read an XML file containing object state updates

Then you can create a dialog to show the equipment details and add the equipment component to the dialog.

The data for this part of the tutorial is in the files:

- **<installdir>/tutorials/gettingStarted/data/equipment_template.xml**
- **<installdir>/tutorials/gettingStarted/data/equipment_config.css**
- **<installdir>/tutorials/gettingStarted/data/equipment_state.xml**

This part of the code is referred to as Step 7:

```
void step7() throws Exception{
```

To do Step 7:

1. Create the equipment component.

   ```
   equipmentComponent = new IlpEquipment();
   ```

   The new instance of `IlpEquipment` is called `equipmentComponent`.

2. Create the data source for accepting the equipment details.

   ```
   equipmentDataSource = new IltDefaultDataSource();
   ```

   The new instance of `IltDefaultDataSource` used to read in the XML file of equipment details is called `equipmentDataSource`.

3. Read in an XML file containing the equipment objects, their positions, and their sizes.

   ```
   equipmentDataSource.parse("equipment_template.xml");
   ```

   The data source you have just created reads in the XML file by parsing the data. The method used is `IltDefaultDataSource.parse(org.xml.sax.InputSource)`.

The data in the file describes items of equipment with their IDs, classes, and attributes.

For example:

```
<addObject id="Shelf1">
  <class>ilog.tgo.model.IltShelf</class>
  <attribute name="name">Shelf1</attribute>
  <attribute name="slotSizes"
    javaClass="ilog.cpl.graphic.views.IlpSlotSizes">
    <width>
      <value>150</value>
      <value>30</value>
      <value>30</value>
      <value>30</value>
      <value>30</value>
      <value>30</value>
    </width>
    <height>
      <value>34</value>
      <value>27</value>
      <value>27</value>
      <value>27</value>
    </height>
  </attribute>
  <attribute name="position" javaClass="ilog.cpl.graphic.IlpPoint">
    <x>22</x>   <y>154</y>
  </attribute>
</addObject>
```

See Loading a shelf defined in XML in the *Business Objects and Data Sources* documentation for details of these XML elements.

This example uses the same static data file to display the detailed equipment view for each network element.

The template file does not contain the states of the objects. These will be loaded in the next step. Creating the objects is done separately from updating their state, because in a real application the same template file would probably be used for many instances of the same type of equipment. Therefore, this file could not contain the state of individual objects. A template could typically be generated by the back end or by the JViews TGO equipment editor. See the tutorial in **<installdir>/tutorials/browser** for a more complete example.

4. Apply a configuration from a CSS file.

```
String [] css = new String[] { "equipment.css" } ;
Try {
  equipmentComponent.setStyleSheets(css);
} catch (Exception e) {
  e.printStackTrace();
}
```

The equipment configuration specifies the background image to load, as well as the configuration of the objects to be displayed in the equipment component.

5. Load the state of the equipment objects from an XML file.

```
equipmentDataSource.parse("equipment_state.xml");
```

Load the current state of the equipment objects from a separate XML file. In a real life application, such a file could be periodically generated by the back end.

6. Connect the data source and the equipment component.

```
equipmentComponent.setDataSource(equipmentDataSource);
```

7. Create a dialog (using pure Swing code).

```
equipmentDialog = createDialog(false);
equipmentDialog.setSize(350, 550);
equipmentDialog.setLocation(700, 300);
```

8. Add the equipment view to the dialog.

```
equipmentDialog.getContentPane().add(equipmentComponent,
                                       BorderLayout.CENTER);
```

This line of code adds the equipment component to the dialog.

9. Fit the view to the pane such that all the contents of the view are displayed.

```
equipmentComponent.fitToContents();
```

The method `IlpEquipment.fitToContents()` modifies the zoom factor so that all the contents are visible in the view.

10. Hide the dialog at first.

```
equipmentDialog.setVisible(false);
}
```

The dialog will become visible in the next step, *Adding interactors*, when an interactor is added to the network elements to show the dialog.

The sample should now look as shown in the following figure.

Adding interactors

Shows you how to configure a pop-up menu that is consistent across all the components. It also shows you how to add double-click behavior for network elements and how to show an overview window with a specific keystroke.

This part of the code is referred to as Step 8.

```
void step8() {
```

To do Step 8:

1. Create a pop-up menu factory to use throughout the application.

```
IlpPopupMenuFactory popupMenuFactory = new IlpAbstractPopupMenuFactory()
{
```

The interface `IlpPopupMenuFactory` is used to create the pop-up menu. It is implemented by `IlpAbstractPopupMenuFactory`. The pop-up menu factory is invoked whenever an end user right-clicks (or clicks) in a view associated with pop-up menus by the system. The pop-up menu factory is expected to return a `JPopupMenu` appropriate to the context in which the end user right-clicked (or clicked) or `null` if no menu should be shown.

2. Add the identifier of each of the selected objects to the menu.

```
public JPopupMenu createPopupMenu (IlpObjectSelectionModel
                                   ilpSelectionModel)
{
```

The method `createPopupMenu(ilog.cpl.util.selection.IlpObjectSelectionModel)` is redefined to display a contextual menu that takes into account the business objects that are currently selected in the component where the pop-up menu is invoked.

3. Create an empty pop-up menu.

```
JPopupMenu menu = new JPopupMenu();
```

The new pop-up menu, `menu`, is an instance of the standard Java™ pop-up menu.

4. Access the selected objects from the selection model.

```
Collection selectedObjects = ilpSelectionModel.getSelectedObjects();
if (!selectedObjects.isEmpty()) {
```

If the `selectedObjects` collection is empty, no objects are selected.

5. If any objects are selected, add the identifier of each object to the pop-up menu.

```
      Iterator i=selectedObjects.iterator();
      while (i.hasNext()) {
        menu.add(((IlpObject)i.next()).getIdentifier().toString());
      }
    } else {
      menu.add("Nothing selected");
    }
    return menu;
  }
};
```

The code iterates through the list of business objects (`IlpObject`) getting the identifier of each object and adding it to the pop-up menu until no more objects are selected. Then it returns the completed pop-up menu.

In this example, the items added to the menu do nothing. In a real application, you would probably associate an implementation of the Swing `Action` interface with each menu item.

6. Set the pop-up menu factory as the interactor for all graphic components.

```
IlpViewInteractor networkInteractor = networkComponent.getViewInteractor
();
networkInteractor.setPopupMenuFactory(popupMenuFactory);
tableComponent.getViewInteractor().setPopupMenuFactory(popupMenuFactory)
;
alarmTableComponent.getViewInteractor().setPopupMenuFactory(
                                                       popupMenuFactory)
;
treeComponent.getViewInteractor().setPopupMenuFactory(popupMenuFactory);
```

First, use the method IlpNetwork. `getDefaultViewInteractor()`getViewInteractor. to get the current interactor for the network component. A network component can have multiple interactors, only one of which is active at any one time. All view interactors implement the `IlpViewInteractor` interface.

Then, use the reference to the interactor retrieved above to ask the interactor to use the pop-up menu factory created in step 1 of the current procedure.

Next, set the pop-up menu factory as the view interactor of the network element table, the alarm table, and the tree.

7. Show the equipment detail view by double-clicking a network element.

First, you create an action, `doubleClickAction`, to show the equipment detail view.

```
Action doubleClickAction = new AbstractAction("Show equipment details")
{
  public void actionPerformed(ActionEvent e) {
    if (e instanceof IlpViewActionEvent) {
      IlpViewActionEvent viewEvent = (IlpViewActionEvent)e;
      IlpObject ilpObj = viewEvent.getIlpObject();
      if (ilpObj!=null) {
```

```
        equipmentDialog.setVisible(true);
        return;
      }
     log.info("Double-click action detected at " + viewEvent.getPosition
()
               + ", not on IlpObject");
    } else
    log.info("Non-CPL action detected");
  }
};
```

The class `IlpViewActionEvent` is used to discover the context of the interaction that triggered the action. When an action is triggered by a view interactor that has recognized a gesture, the action event received by the action is a view action event.

Note that the equipment dialog created in Step 7, *Showing equipment details*, is set to be visible.

Then, you associate the action with a double-click gesture in the interactors of the various components.

```
networkInteractor.setGestureAction(IlpGesture.BUTTON1_DOUBLE_CLICKED,
                                   doubleClickAction);
tableComponent.getViewInteractor().setGestureAction(
                                   IlpGesture.BUTTON1_DOUBLE_CLICKED,
                                   doubleClickAction);
treeComponent.getViewInteractor().setGestureAction(
                                   IlpGesture.BUTTON1_DOUBLE_CLICKED,
                                   doubleClickAction);
```

The first parameter, `BUTTON1_DOUBLE_CLICKED`, identifies the gesture that should trigger the action. A gesture is a series of one or more graphic events triggered by the end user.

The class `IlpGesture` contains (as static fields) a number of gestures that are recognized by some or all interactors. Its subclasses contain additional gestures that are specific to individual interactors.

8. Show the overview window by pressing the key **o**.

First, you create an action, `overviewAction`, to show the overview window.

```
Action overviewAction = new AbstractAction("Show Overview") {
  public void actionPerformed (ActionEvent e) {
    networkComponent.setOverviewVisible(true);
  }
};
```

When this action is performed, the overview window becomes visible.

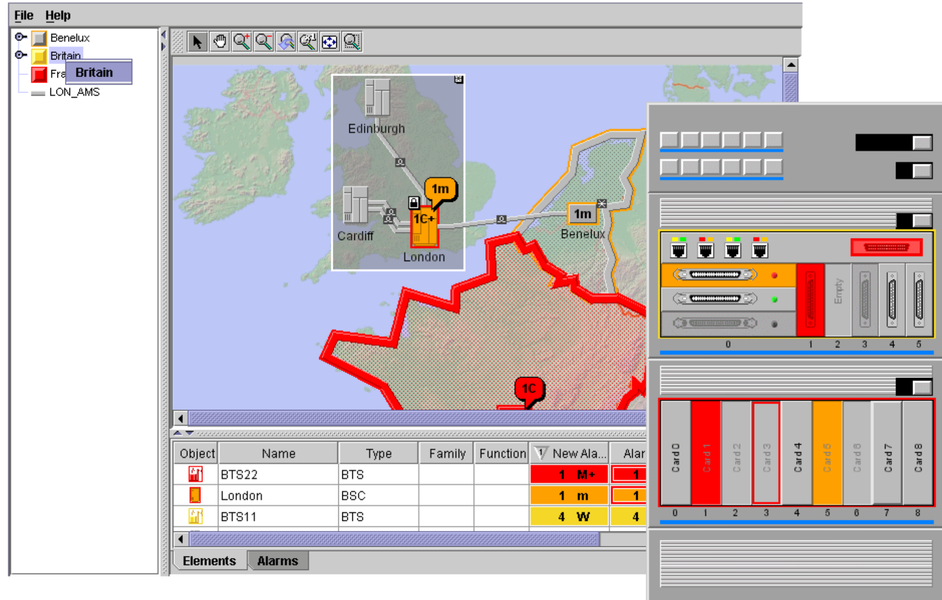
Then, associate the keystroke **o** with `overviewAction`.

```
networkInteractor.setKeyStrokeAction
```

```
                                   (KeyStroke.getKeyStroke('o'),overviewAction)
;
```

The default network view interactor has the method `setKeyStrokeAction(javax.swing.KeyStroke, javax.swing.Action)`, inherited from `IlpAbstractInteractor` to associate a keystroke with a specified action.

The sample should now look as shown in the following figure.



Handling selection

This topic shows you how to:

- ♦ Change the tree selection model to single selection
- ♦ Listen for selection changes in the tree
- ♦ Select an object in the network whenever the same object is selected in the tree
- ♦ Pan the network so that the selected object becomes visible

This part of the code is referred to as Step 9:

```
void step9() {
```

To do Step 9:

1. Set the tree selection mode to single selection.

```
final IlpTreeSelectionModel treeSelectionModel =
 (IlpTreeSelectionModel)treeComponent.getSelectionModel();
treeSelectionModel.setSelectionMode
                              (TreeSelectionModel.SINGLE_TREE_SELECTION);
```

The interface `IlpTreeSelectionModel` implemented here by `treeSelectionModel` extends `javax.swing.tree.TreeSelectionModel`. It is the selection model for the tree component (`IlpTree`). It also implements `IlpObjectSelectionModel`, which allows you to access the selection as business objects.

The method `setSelectionMode` is inherited from `javax.swing.tree.TreeSelectionModel`. Here you get the tree selection model `treeSelectionModel` and set its mode to `SINGLE_TREE_SELECTION`.

2. Create a selection listener.

```
TreeSelectionListener selectionListener = new TreeSelectionListener() {
  public void valueChanged(TreeSelectionEvent e) {
    IlpObject selectedObject = treeSelectionModel.getSelectedObject();
    if (selectedObject!=null) {
```

You create a new tree selection listener, `selectionListener`, to listen for tree selection events. When the value of the tree selection event changes, the selected business object (`selectedObject`), if there is one, is obtained by the tree selection model.

3. Check to see whether one of the network objects is selected.

```
    if (selectedObject.getIlpClass().isSubClassOf(IlpObject.GetIlpClass()
))
     {
```

```
        IlpNetworkSelectionModel networkSelectionModel =
                                     networkComponent.getSelectionModel();
```

If the `IlpClass` of the selected object is a subclass of `IltObject` (that is, it is one of the predefined JViews TGO objects and not an alarm), an attempt will be made to select the corresponding object in the network component.

First, get the network selection model, which implements `IlpNetworkSelectionModel`.

4. Remove the previous selection from the network.

```
networkSelectionModel.clearSelection();
```

5. Add a new object to the selection.

```
networkSelectionModel.addSelectionObject(selectedObject);
```

The inherited method `IlpObjectSelectionModel.addSelectionObject(ilog.cpl.model.IlpObject)` adds the selected business object `selectedObject` to the network selection model. This causes the corresponding graphic representation to appear as selected in the network component.

6. Pan the network view so that the selected object becomes visible.

```
          networkComponent.ensureVisible(selectedObject);
        }
      }
    }
  };
```

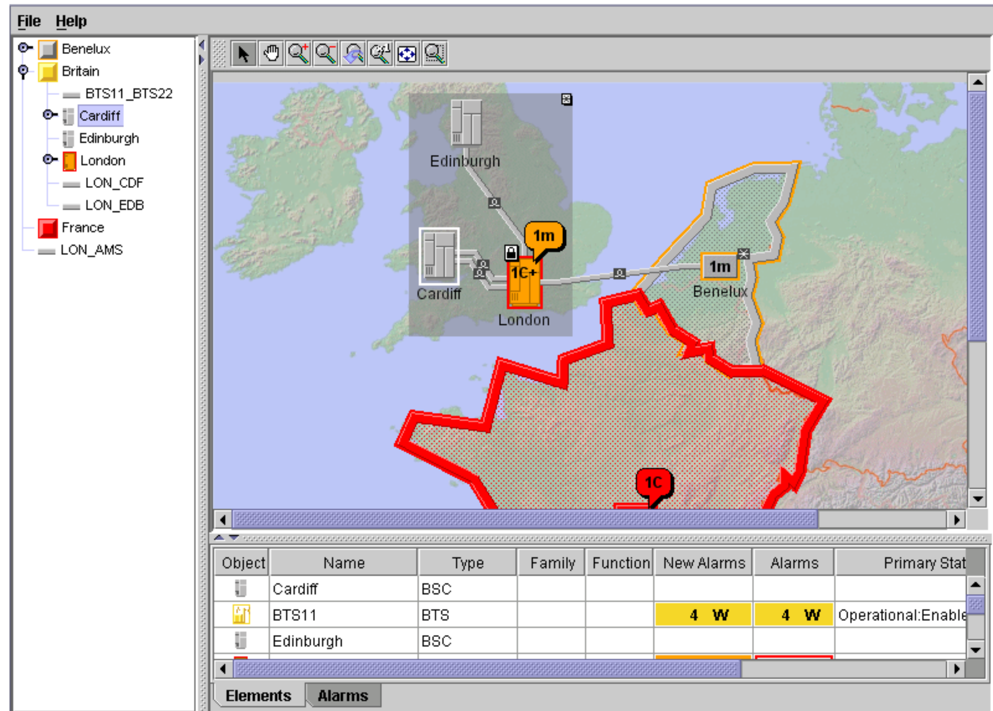
The method `IlpNetwork.ensureVisible(ilog.cpl.model.IlpObject)` pans the network component if necessary, so that the selected object becomes visible.

7. Add the selection listener to the tree component

```
  treeSelectionModel.addTreeSelectionListener(selectionListener);
}
```

The method `addTreeSelectionListener` is inherited from `javax.swing.tree.TreeSelectionModel`. The selection listener created above is added to the tree selection model to listen for selection changes.

The sample should now look as shown in the following figure.



The remaining code is based on Java™ and Swing facilities and will not be explained in detail.

```
/**
 * Executable entry point.
 */
public static void main(String[] args) {
  JFrame frame = new JFrame(sampleTitle);
  AbstractSample sample = new Main();
  sample.init(frame.getContentPane());
  frame.setSize(800,600);
  frame.setVisible(true);
}

/**
 * Common initialization function.
 * Is called either by the main function if this sample
 * is run as an application, or by the baseclass if it
 * is run as an applet.
 */
  public void init (Container container) {
    // Must call baseclass initializer
    super.init(container);
```

```
    // Initialize the subframes
    initSubFrames(container);

    doSample(container);
  }

/**
 * Create a number of subframes, separated by splitters.
 */
public void initSubFrames (Container container) {
    container.setLayout(new BorderLayout());

    // Create panels for the JTGO graphic components
    this.networkArea=new JPanel(new BorderLayout());
    this.treeArea=new JPanel(new BorderLayout());
    this.tableArea=new JPanel(new BorderLayout());

    // Split the main frame into three areas
    // Add the JTGO graphic components to the three areas
    this.rightSplitPane = new JSplitPane(JSplitPane.VERTICAL_SPLIT,
                                         this.networkArea, this.tableArea);
    this.rightSplitPane.setDividerLocation(400);
    this.rightSplitPane.setResizeWeight(.9);

    this.rightSplitPane.setOneTouchExpandable(true);
    this.rootSplitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
                                        this.treeArea, this.rightSplitPane);
    this.rootSplitPane.setDividerLocation(150);
    this.rootSplitPane.setOneTouchExpandable(true);

    // Add divided frames to main panel
    container.add(this.rootSplitPane, BorderLayout.CENTER);

  }
  /**
   * Create a dialog, transient for the main frame if possible
   */
  public JDialog createDialog (boolean modal) {
    // Create a new dialog
      JDialog dialog;
      // dialog should be transient for this view if in a regular
      // application
      if (isApplet())
        dialog = new JDialog();
      else
        dialog = new JDialog((Frame)getTopLevelAncestor(), modal);
      return dialog;
    }

    /**
     * Creates a tabbed pane for multiple tables
     * Inserts the existing table view into the first pane,
     * and the new view into the second
     */
```

```
    public void initTableTab (Container container, IlpTable
                                                   newTableComponent) {
      // Create a tabbed pane two contain two tables
      JTabbedPane tabbedPane=new JTabbedPane(JTabbedPane.BOTTOM);

      // Remove the existing table component from the table area
    tableArea.remove(tableComponent);

    // Add the existing table component to the first tab
    tabbedPane.add("Elements", tableComponent);

    // Add the new table view to the second tab
    tabbedPane.add("Alarms", newTableComponent);

    // Add the whole tabbed pane to the table area
    tableArea.add(tabbedPane);

    // Set the tabbed pane to show the element table
    tabbedPane.setSelectedComponent(tableComponent);
  }
}
```

Index

B

business
  objects **11**

C

Cascading Style Sheets (CSS) **15**
CSS2 Recommendation **15**
custom business objects **33**

D

data sources **11**

E

equipment component **7, 40**

G

graphic components **7**

N

network component **7**
  configuring **30**
  creating **22**

S

selection handling **48**

T

table component **8, 28**
tree component **9, 25**

© Copyright IBM Corp. 1987, 2009