



IBM ILOG JViews TGO V8.6

**Business objects and data
sources**

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

Introducing business objects and data sources.....	9
Overview.....	10
Business model and business classes.....	11
Data sources.....	12
Predefined business objects.....	13
Network elements.....	14
Links.....	16
Groups.....	17
Subnetworks.....	19
Shelves, cards, ports, and LEDs.....	20
Base Transceiver Stations (BTS).....	23
Off-page connectors.....	24
Alarms.....	25
States.....	26
Predefined business classes.....	29
Overview of the predefined business classes.....	30
Attributes of predefined business objects.....	34
Computed attributes based on the object state.....	38
Extending predefined business classes.....	39
The business model.....	43
Business model, business classes, and business objects.....	44

Integrating the business model with the back end.....	45
Defining the business model in XML.....	47
Defining a dynamic class in XML.....	48
Extending a predefined business class in XML.....	54
Loading the business model.....	56
Business model API.....	57
Class overview.....	58
Business class API.....	59
Business object API.....	60
Attribute API.....	62
Business class manager API.....	66
Defining the business model from JavaBeans classes.....	68
Defining the business model with dynamic classes.....	71
Defining a dynamic class using the API.....	72
Extending a predefined business class using the API.....	75
Data sources.....	79
About data sources.....	81
Data source API.....	82
Adding business objects from XML.....	85
Reading an XML file into a data source.....	86
Writing the data source content to XML.....	95
Adding predefined business objects.....	96
Adding business objects from JavaBeans.....	97
Adding dynamic business objects.....	98
Defining business object relationships.....	101
Grouping changes in batches.....	106
Advanced parsing and writing of a data source.....	108
Implementing a new data source.....	110
Network elements.....	115
Network element class.....	116
Loading a network element defined in XML.....	118
Creating a network element with the API.....	120
Representation of network elements in a network.....	121
Network element types.....	122
Network element functions.....	135
Network element families.....	139

Partial network elements.....	140
Shortcut network elements.....	141
Network element sizes.....	142
Representation of network elements in a table and in a tree.....	143
Links.....	145
Classes overview.....	147
Links.....	148
Link sets.....	151
Link bundles.....	154
Representation of links in a network.....	158
Representation of links in a table and in a tree.....	162
Link connection ports.....	163
Link programming examples.....	167
Groups.....	171
Group class.....	172
Group shapes.....	173
Loading a group defined in XML.....	175
Creating a group with the API.....	177
Representation of groups in a table and in a tree.....	179
Subnetworks.....	181
About subnetworks.....	182
Loading a subnetwork defined in XML.....	183
Creating a subnetwork with the API.....	185
Representing alarms in expanded subnetworks.....	187
Shelves and cards.....	191
Overview of classes.....	192
Shelves.....	193
Overview of shelves.....	194
Shelf class.....	195
Loading a shelf defined in XML.....	196
Creating a shelf with the API.....	197
Shelf items.....	199
Cards.....	201
Overview of cards.....	202
Card class.....	203

Loading a card defined in XML.....	204
Creating a card with the API.....	206
Empty slots.....	209
Overview of empty slots.....	210
Empty slot class.....	211
Loading an empty slot defined in XML.....	212
Creating an empty slot with the API.....	214
Card carriers.....	217
Overview of card carriers.....	218
Card carrier class.....	219
Loading a card carrier defined in XML.....	220
Creating a card carrier with the API.....	222
Card items.....	225
Overview of card items.....	226
Card item class.....	227
LEDs.....	229
Overview of LEDs.....	230
LED class.....	231
Loading an LED defined in XML.....	232
Creating an LED with the API.....	234
Predefined LED types.....	235
Ports.....	237
Overview of ports.....	238
Port class.....	239
Loading a port defined in XML.....	240
Creating a port with the API.....	242
Predefined port types.....	243
Representation of shelves and cards in a table and in a tree.....	247
BTS (Base Transceiver Station).....	249
BTS Class.....	250
Loading a BTS object defined in XML.....	252
Creating a BTS object with the API.....	254
Representation of BTS objects in a table and in a tree.....	256
Off-page connectors.....	257
Off-page connector class.....	258
Loading an off-page connector defined in XML.....	259
Creating an off-page connector with the API.....	260
Representation of off-page connectors in a network.....	261
Representation of off-page connectors in a table and in a tree.....	262

- Alarms.....263**
 - Alarm object class.....264**
 - Loading an alarm defined in XML.....267**
 - Creating an alarm with the API.....269**
 - Representation of alarms in a network.....270**
 - Representation of alarms in a table and in a tree.....271**
- Lookup tables for state visuals.....273**
 - The OSI state dictionary visuals.....275**
 - Graphical representation of the OSI primary states.....276
 - Graphical representation of OSI secondary states.....278
 - The Bellcore state dictionary visuals.....283**
 - Graphical representation of the Bellcore primary states.....284
 - Graphical representation of the Bellcore secondary states.....285
 - The SNMP state dictionary visuals.....293**
 - Graphical representation of SNMP primary states.....294
 - Graphical representation of SNMP secondary states.....295
 - The Misc state dictionary visuals.....305**
 - Graphical representation of Misc secondary states.....306
 - The Performance state dictionary visuals.....309**
 - Graphical representation of Performance secondary states.....310
 - The SAN state dictionary visuals.....315**
 - Graphical representation of SAN secondary states.....316
 - The SONET state dictionary visuals.....319**
 - Graphical representation of SONET primary states.....320
 - Graphical representation of SONET secondary states.....323
- States.....325**
 - Graphical representations of predefined business object states.....327**
 - State dictionaries: an overview.....329**
 - The OSI state dictionary.....331**
 - The Bellcore state dictionary.....333**
 - The SNMP state dictionary.....334**
 - Miscellaneous states: the Misc state dictionary.....336**
 - Performance states: the Performance state dictionary.....337**
 - SAN states: the SAN state dictionary.....338**
 - Link states: the SONET state dictionary.....339**

Alarm states	341
Graphical representation of alarm conditions.....	342
Setting the alarm counters.....	348
Defining alarm states with the API.....	350
Loading alarm states in XML.....	352
Trap states	353
Managing states	357
State values, state classes, and state systems.....	358
Object states.....	361
The object state classes.....	364
Modifying states and statuses.....	366
Accessing and removing states.....	368
Defining states in XML	369
Overview.....	371
OSI states.....	374
Bellcore states.....	378
SNMP states.....	381
Miscellaneous states.....	386
Performance states.....	388
SAN states.....	390
SONET states.....	392
BiSONET states.....	394
Alarm states.....	396
Trap states.....	399
Information window	401
System window	403
Customizing the representation of states and alarms	404
Index	405

Introducing business objects and data sources

Presents the concepts of business model and data sources as well as the predefined business objects supplied with JViews TGO.

In this section

Overview

Introduces the relationship between a back-end application and JViews TGO objects such as data sources, business objects, graphic components.

Business model and business classes

Defines the concept of business model.

Data sources

Defines the concept of data source.

Predefined business objects

Introduces the predefined business objects provided by JViews TGO.

States

Provides a basic description of the visual aspect of states.

Predefined business classes

Shows class diagrams of the predefined business classes and describes their attributes and how to extend them.

Overview

IBM® ILOG® 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.

JViews TGO provides a set of predefined business classes that you can use directly in your applications. These classes are specifically designed for easing the development and leveraging the overall graphic quality and the ergonomics of user interfaces in telecommunication applications. All you have to do is insert instances of these predefined business classes for them to be translated into high-quality graphic representations with a common look and feel in all the JViews TGO graphic components. Predefined business objects include a default graphic renderer that maps them to graphic representations automatically, thus significantly minimizing coding efforts.

In addition, JViews TGO furnishes 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 widely-spread telecommunication standards, such as OSI, Bellcore, and SNMP. For detailed information on graphical representations of states and alarms, see *States* and *Lookup tables for state visuals*.

Business model and business classes

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 objects are instances of business classes, which are described by a business model.

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

Business classes can also be defined directly in XML.

Data sources

Data sources are connecting objects that bridge the back-end with 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 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 original 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

Introduces the predefined business objects provided by JViews TGO.

In this section

Network elements

Presents the different kinds of network elements and their graphical representation.

Links

Provides some examples of links and their graphical representation.

Groups

Explains what a group is and presents the different kinds of groups and their graphical representation.

Subnetworks

Defines the concept of subnetwork and shows the two possible ways to display a subnetwork.

Shelves, cards, ports, and LEDs

Defines and illustrates the concepts of shelves, cards, ports, and LEDs.

Base Transceiver Stations (BTS)

Defines the concept of base transceiver station.

Off-page connectors

Describes what off-page connectors are used for.

Alarms

Describes the specificity of the JViews TGO alarm object.

Network elements

Network elements include any kind of shelf-based telecom or data-communications equipment (a switch, a multiplexer, a cross-connect, and so on), outside plant equipment (coax node), and peripheral equipment (terminal or printer).

Network element representations

A network element can be represented by a pictorial representation (bitmap image or vector drawing), a symbol, or a shape. Not all physical details of the element are visible in the representation.

- ◆ **Pictorial representation.** The network element base is a bitmap image or vector drawing. This drawing is meant to be realistic. Several predefined bases are available for shelf-based equipment, terminals, and mobile phone access network elements. New bases can easily be introduced by providing bitmap images.



Pictorial representations of shelf-based equipment and terminal

- ◆ **Symbolic representation.** The network element base has a square and the network element function is denoted by a symbol containing ITU/ANSI or traditional symbols. The default type corresponding to the default symbolic network element representation is called NE (Network Element). The following figure illustrates an NE type network element: here, an add-drop multiplexer with a capacity of OC192.



Symbolic representation of NE type network element

- ◆ **Shape representation.** The network element base has a geometric shape that symbolizes the network element type (or function class). The center of the base may contain an icon that further refines the network element function. Several predefined shapes are provided as types of the network elements. The following figure illustrates a Mux shape network element.



Shape representation of mux network element

Partial network elements

A *partial network element* is an abstraction which denotes a network element that is only part of the real-world network element. Partial network elements can be used in several situations, for example:

- ◆ To represent distributed clusters where parts of a cluster need to be divided across different subnetworks.
- ◆ To allow one network element to be used by different service providers. In this case, the network element needs to be divided in several parts. Each part is represented as a partial network element and its state reflects only the elements that are interesting for the service provider that is using it.

Partial network elements are graphically represented by an icon located at the bottom left of the network element base.



Partial network elements expanded and collapsed

Shortcuts

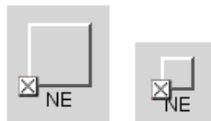
A *shortcut* network element is an abstraction denoting an object that is only a reference to an existing network element.

Shortcuts can be either *standard* or *dangling*. In the first case, the network element is a shortcut to another object that is managed by the system. In the second case, the network element is a shortcut to an object that is currently not available, which means that the shortcut is dangling and needs to be validated by the management system.

Shortcut network elements are graphically represented by an icon located at the bottom left of the network element base.



Standard shortcuts



Sample links

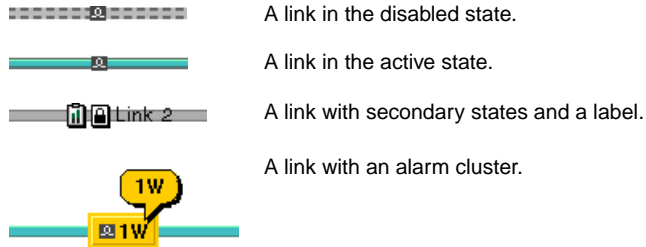
For more information about network elements, see *Network elements*.

Links

Links are used to display the transmission elements making up the network lines.

Links feature the same dynamic display as network elements. Each type of link has its own graphical representation, and different link states are represented graphically by changes in color or internal pattern design. Links can also carry decorations, in particular to represent alarms. Links can be directional.

Some sample links are shown in the following figure.



The drawing of a link between two nodes is, by default, automatically calculated by JViews TGO. The link calculation is performed using a layout optimizer.

The default layout optimizer provided draws direct links without intermediate points and according to a given angle. The links are attached to nodes using small horizontal or vertical segments.

For more information about links, see *Links*.

Links can also be grouped in a *link bundle*, that is, a set of links that have the same destination node. Links in a bundle can be collapsed to a single overview link.

Groups

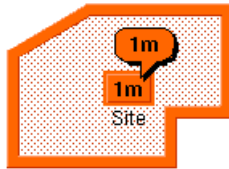
Groups are containers that logically group network resources. They are used to display a geographic or a functional region by grouping together either network elements (such as multiplexers, switches, and so on) or links (such as transport, access, and so on).

Groups support the whole set of state and alarm representations: alarm balloons, severity color, primary state representation graphics, status or secondary state icons, and so forth.

Groups have three kinds of visual representations: polygonal, rectangular, and linear.

Polygonal groups

Polygonal groups are flexible containers that generally represent a group of network elements at a regional level. Polygonal groups usually do not represent a physical object, but rather a user-defined collection of objects that are not necessarily located in the same place. Polygonal groups are represented by a screened transparent polygon with a relief border as shown in the following figure.



Polygonal group with single minor alarm

In this figure the group label, the alarm counter, and the alarm balloon are displayed as a cluster. The drawing of this cluster is organized around a central rectangle called a plinth which is drawn by default at the polygon center.

Rectangular groups

Rectangular groups hold network elements that are located in the same place such as a site, a building, or a city. They can be resized to create any kind of rectangular container. Rectangular groups look like an opaque relief rectangle, as shown in the following figure.



Rectangular group with single warning alarm

Rectangular groups with alarms contain an information cluster located by default at the center of the rectangle.

Linear groups

Linear groups are bendable, “pipe-like” containers. They are used to hold a group of network elements and links or they may represent the backbone transport system in the network. The graphical representation of linear groups evokes a linear collection of objects. For example, linear groups can be used to represent all the repeaters between two line termination network elements.

The following figure shows an example of a linear group on which a critical alarm has been detected.



Linear group with single critical alarm

When an alarm is displayed on a linear group, an information cluster appears at the center of the median segment. The median segment is the segment containing the midpoint of the link.

Shortcuts

A *shortcut* group is an abstraction denoting an object that is only a reference to an existing group.

Shortcuts can be either *standard* or *dangling*. In the first case, the group is a shortcut to another object that is managed by the system. In the second case, the group is a shortcut to an object that is currently not available, which means that the shortcut is dangling and needs to be validated by the management system.

Shortcut groups are graphically represented by an icon located at the bottom left of the group plinth.



Standard shortcut



Dangling shortcut

For more information about groups, see *Groups*.

Subnetworks

Subnetworks allow you to create applications that display a network inside another network. They are not pure predefined business objects in the sense that they are created automatically by the JViews TGO network component when you define a containment relationship between objects in the data source.

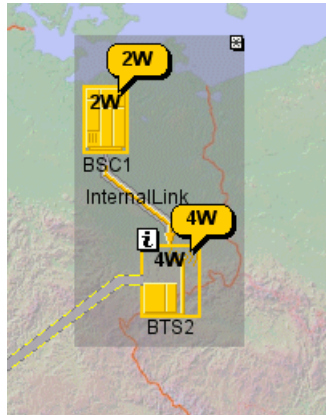
A subnetwork can be defined as any business object with child objects in the network component. You can display it either collapsed or expanded in the network component.

- ◆ In the *collapsed* state, the subnetwork is represented as a single object.



Collapsed subnetwork

- ◆ In the *expanded* state, the subnetwork is displayed with all the objects contained in it.



Expanded subnetwork

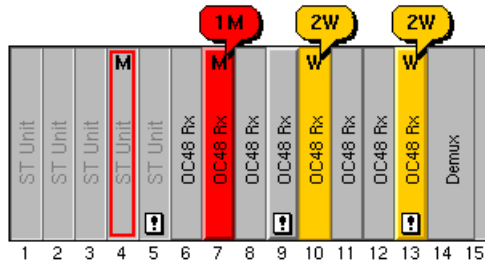
For more information about subnetworks, see *Subnetworks*.

Shelves, cards, ports, and LEDs

Shelves are telecom objects that can be made up of a certain number of slots of different widths in which cards can be stored. By default, slot numbers are displayed at the bottom of the slots.

Cards are represented by rectangles that support the same base states as network elements. Like network elements, they can carry alarm and status icons. They are used to display details of modifications that have been made to the states and alarms of an item of equipment at the level of the physical card.

The following figure shows a shelf with empty slots (the first five) and cards in various states, some of them carrying alarms.



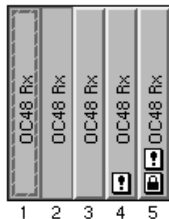
Shelf with cards in various states

By default, status and secondary state icons are displayed at the bottom of the card, and alarms at the top of the card. Only the letter corresponding to the highest outstanding alarm is displayed. Other counting information such as figures and + signs do not appear.

There are three different types of card:

- ◆ *Standard cards* correspond to the description given above. Such cards can take up one or several slots on the shelf and do not necessarily extend across an entire shelf section. You can, for example, place a card starting from slot #3 of the shelf and occupying this slot, the next slot and 50% of slot 5. Each card has its own label.

An example of a set of standard cards in various states (from disabled on the left to busy and shutting down on the right) is provided in the following figure.



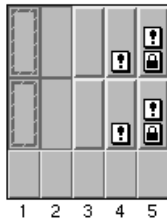
Standard cards in various states

- ◆ *Empty slots* correspond to an empty portion of the shelf for which you want to display certain characteristics, either static (a label) or dynamic (statuses and alarms). The following figure shows a set of empty slots, where the basic design does not vary.



Empty slots

- ◆ *Card-carrier cards* hold other cards in a linear arrangement. All cards contained in the card carrier are the same size and can contain states, statuses, and alarms. The following figure shows a set of five card carriers, each of which carries two cards (in various states). The card carrier itself can carry statuses and alarms. The area at the bottom of the card carrier is used to display a graphic representation of its state.



Several card carriers with cards

Cards can contain ports and LEDs, which are called card items.

Ports are the physical interfaces of pieces of equipment and are usually located on cards. Ports can also be connectors.



Example of ports

LEDs (Light Emitting Diode) are used to represent the state of an item of equipment through a color. Most types of equipment use LEDs as interfaces to provide the user with information on hardware and software conditions.



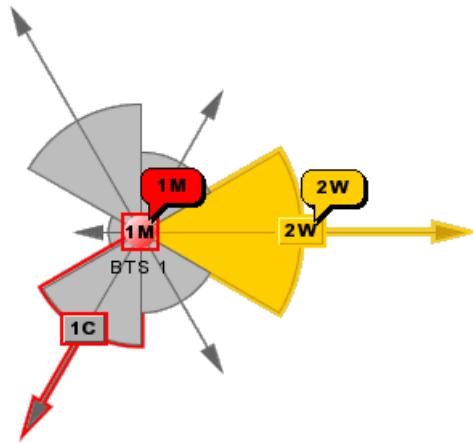
Examples of LEDs

For more information about shelves and cards, see *Shelves and cards*.

Base Transceiver Stations (BTS)

Base Transceiver Stations (BTS) are base stations composed of antennas that relay (receive and transmit) radio messages within cells of a cellular phone system.

Each antenna has an orientation and a beam width that are graphically represented. Each antenna can have its own state and graphical characteristics.



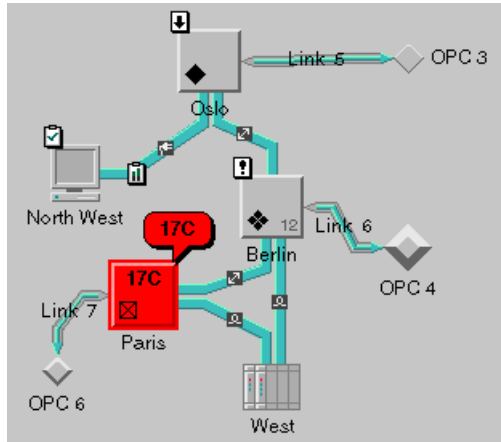
A BTS with three antennas

For more information about base transceiver stations, see *BTS (Base Transceiver Station)*.

Off-page connectors

Off-page connectors usually come in pairs and are used to show the continuation of a link from one network to another. They can be used in place of nodes (either groups or network elements) and can have links connected to them.

The following figure shows a partial network with three off-page connectors, each with a different graphic representation.









Network with different representations of off-page connectors

For more information about off-page connectors, see *Off-page connectors*.

Alarms

The JViews TGO alarm object is based on the alarm object defined by the Java™ Specification Requests (JSR)-90 workgroup. This alarm object has been designed to be used in the development of OSS/J Quality of Service APIs (telecom management applications).

Alarms are of two kinds: *raw* and *impact*.

Alarm	Notification	Severity	Ack	Date Raised	Managed Object Instance	Probable Cause
 alarm 1		Warning	✓	06:00:12 GMT-03:00		Indeterminate
 alarm 2		Minor	✓	06:24:52 GMT-03:00		Bandwidth reduction
 alarm 3		Cleared		06:32:28 GMT-03:00		Excessive bit error rate
 alarm 4		Minor!	✓	08:48:02 GMT-03:00		Indeterminate
 alarm 5		Cleared	✓	09:07:05 GMT-03:00		Unavailable
 alarm 6		Minor	✓	09:09:22 GMT-03:00		Excessive bit error rate

An alarm table with raw (balloons) and impact (clouds) alarms

A raw alarm is an alarm reported by a network element and carried by this element.

An impact alarm corresponds to a propagated alarm that is reported by a network element but carried by another element.

Alarms can be represented as individual objects in a table and tree. For more information, see *Alarms*.

Note: Alarms can also be represented as part of the managed object (in a network or equipment view). In this case, they are considered as part of the object state and not as individual objects. For more information, see *Alarm states*.

States

Base element states

JViews TGO provides a comprehensive and user-friendly graphical environment for network states and components, as well as for the alarms detected by the supervisory system (network management application).

For full details about states, refer to *States*.

In most cases, telecom equipment is in one of the following three fundamental states:

- ◆ Out Of Service (OOS)
- ◆ In service but carrying No Traffic (NT)
- ◆ In service and Carrying Traffic (CT)

These states may be named differently depending on the telecom standard used, but these three fundamental states appear in the main state models.

States are represented graphically through a base element. The following figure shows a network management workstation in the three fundamental states.



Network management workstation in different states

State modifiers

The other states introduced by the various state models are represented by icons placed at the base of the telecom object. These icons, called *modifiers*, enhance the visual representation provided by the base state. The following figure illustrates how state modifiers placed on a network element and on a link are displayed.



Network element and link with state modifiers

Alarm states

A number of graphical properties have been developed to notify the operator of the presence of alarms. When a new alarm is registered on an element, four visual cues are added to the graphical representation of the element:

- ◆ An alarm counter is displayed in the network element base.
- ◆ An alarm balloon appears above the network element displaying another alarm count.

- ◆ The object base turns a vibrant color (red, orange, or yellow) according to the severity of the alarm.
- ◆ A colored outline is associated with the object base.

The following figure shows a network element in three different alarm states illustrating the colored base, the outline, the alarm balloon, and the alarm count.



Network element with outstanding alarms

Details of the alarm color coding scheme and other alarm characteristics are provided in *Alarm states*.

Predefined business classes

Shows class diagrams of the predefined business classes and describes their attributes and how to extend them.

In this section

Overview of the predefined business classes

Presents the relationship between the predefined business classes.

Attributes of predefined business objects

Describes the attributes of the `ItObject` class and its subclasses.

Computed attributes based on the object state

Shows how to define a new computed attribute.

Extending predefined business classes

Shows the two ways to dynamically extend a predefined business class.

Overview of the predefined business classes

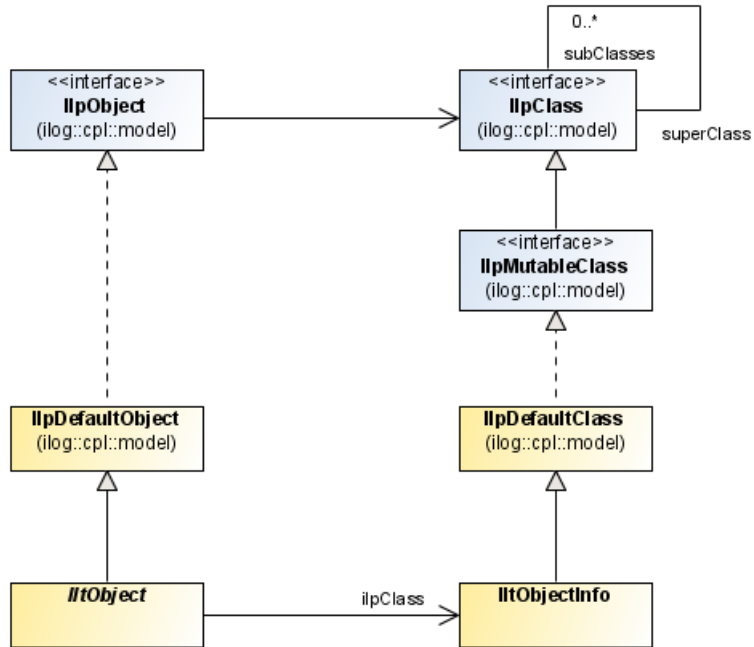
Predefined business objects include telecommunication network managed objects such as:

- ◆ Network elements of the class `IltNetworkElement`
- ◆ Links of the class `IltLink`
- ◆ Groups of links of the classes `IltLinkBundle`, `IltLinkSet`
- ◆ Groups of the class `IltGroup`
- ◆ Shelves, cards, ports, and LEDs of the classes `IltShelf`, `IltCard`, `IltPort` and `IltLed`
- ◆ Dedicated wireless representations of BTS and antennas of the classes `IltBTS` and `IltBTSAntenna`
- ◆ Off-page connectors of the class `IltOffPageConnector`

Inheritance tree of predefined business classes provides the complete hierarchy for business object classes. For a detailed description of each of these object classes, see the subsequent sections in this documentation.

Predefined business object classes are subclasses of `IltObject`, which is itself a subclass of `IlpDefaultObject`. For each class deriving from `IltObject`, there is an instance of `IltObjectInfo` that is a subclass of `IlpDefaultClass`.

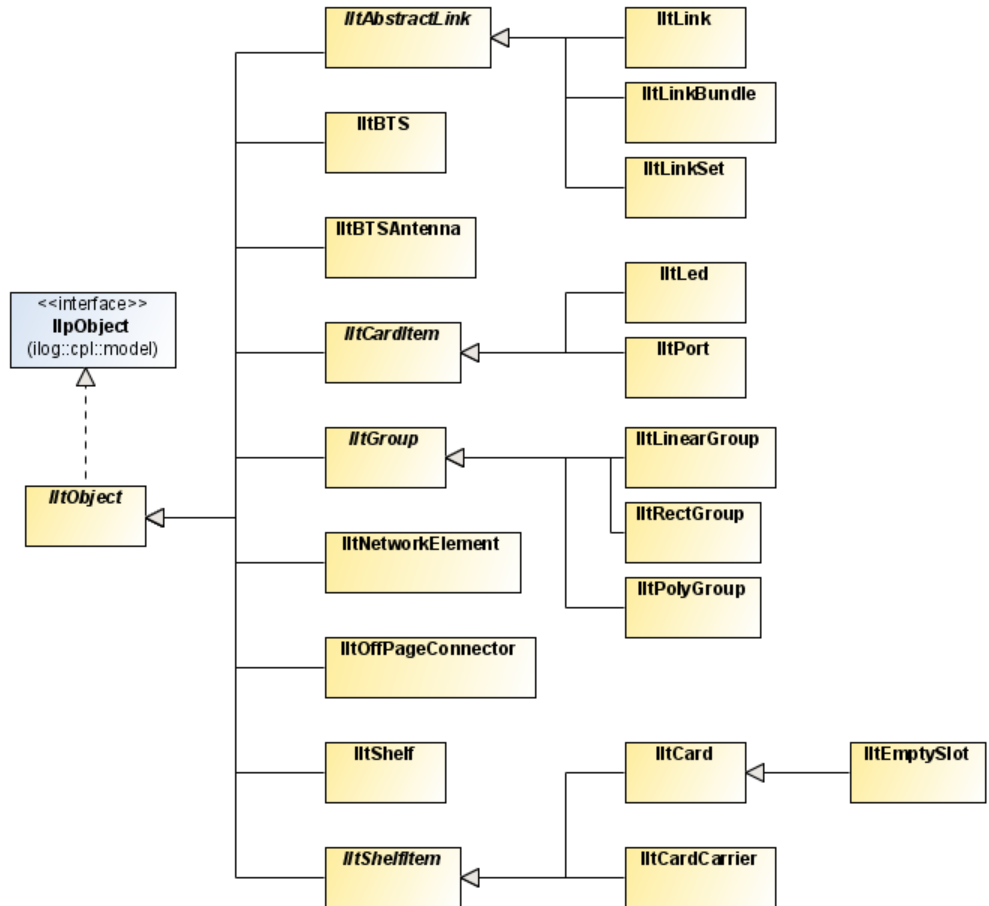
The following diagram shows the relationship between these classes.



IltObject inheritance path

For details on `IlpDefaultObject` and `IlpDefaultClass`, see *Business model API*.

The following figure shows the inheritance tree of predefined business classes.



Inheritance tree of predefined business classes

All predefined business object classes can be retrieved using the method `GetItpClass`, which is declared in each one of these classes; for example, `GetItpClass()`. Every business class contains specific attributes that you can set using their particular API, for example, `IltNetworkElement.setFamily(value)`, or the generic `ItpObject` API, for example, `setAttributeValue(IltNetworkElement.FamilyAttribute, value)`. See *Business model API*.

Instances of predefined business classes hold two types of data: structural data and states and alarms.

◆ Structural data remains constant while the application is running. It includes:

- Characteristics of the element (for example, its name, Toronto-C10).
- The key properties of the element that have an impact on its own representation, regardless of its states or alarms. The network element function and family (such as, ATM, OC192) are examples of structural data that can be displayed permanently.

- ◆ State and alarm data describes the most recently known or inferred state of the managed object. A state can have several different aspects that depend on the type of network management used. For example, in the OSI state system, there are three categories of states: operational states, usage states, and administrative states. On top of these values, a set of statuses can further qualify the managed object. In other standards, such as Bellcore, all states are either primary or secondary states.

For an introduction to state and alarm visuals, refer to *States*.

Attributes of predefined business objects

By default, the `IltObject` class defines several attributes that are present in all its subclasses. These attributes are the following:

- ◆ `Name` indicates the name of the business object and is represented graphically by a label.
 - Name: `name`
 - Value class: `String`
 - Attribute: `IltObject.NameAttribute`
- ◆ `Graphic Representation` is a computed attribute used to make it possible to display the entire business object in one column of a table. This attribute is computed from all the other attributes attached to the object and cannot be set to a value directly.
 - Name: `graphicRepresentation`
 - Value class: `ilog.tgo.model.attribute.IltGraphicRepresentationAttributeType`
 - Attribute: `IltObject.GraphicRepresentationAttribute`
- ◆ `Object State` defines the state of the business object. See *States* for a complete description of object states. By default, the object state is not displayed in the table. This attribute is used as a base for all computed attributes that display the state of the telecom object. For more information, see *Computed attributes based on the object state*.
 - Name: `objectState`
 - Value class: `ilog.tgo.model.IltObjectState`
 - Attribute: `IltObject.ObjectStateAttribute`
- ◆ `Position` indicates the geometric position or the shape of the business object in the network and equipment components. See *Positioning* for more information. By default, the object position is not displayed in the table.
 - Name: `position`
 - Value class: `ilog.cpl.graphic.IlpPosition`
 - Attribute: `IltObject.PositionAttribute`
- ◆ `New Alarm Count` indicates the number of new raw alarms or traps of the business object. This string is displayed in the alarm balloon of the object. This count is computed from the object state and should not be set to a value directly.
 - Name: `newAlarmCount`
 - Value class: `ilog.tgo.model.IltAlarmCountAttributeType`
 - Attribute: `IltObject.NewAlarmCountAttribute`

- ◆ **New Alarm Count Number** indicates the number of new raw alarms or traps of the business object. This alarm count is computed from the object state and should not be set to a value directly.
 - **Name:** `newAlarmCountNumber`
 - **Value class:** `java.lang.Integer`
 - **Attribute:** `IltObject.NewAlarmCountNumberAttribute`
- ◆ **Alarm Count** indicates the number of outstanding raw alarms or traps of the business object. Outstanding refers to both acknowledged and new alarms or traps. This string displays on the object base. The count is computed from the object state and should not be set to a value directly.
 - **Name:** `alarmCount`
 - **Value class:** `ilog.tgo.model.IltAlarmCountAttributeType`
 - **Attribute:** `IltObject.AlarmCountAttribute`
- ◆ **Alarm Count Number** indicates the number of outstanding raw alarms or traps of the business object. Outstanding refers to both acknowledged and new alarms or traps. This integer represents the number of outstanding raw alarms or traps. The count is computed from the object state and should not be set to a value directly.
 - **Name:** `alarmCountNumber`
 - **Value class:** `java.lang.Integer`
 - **Attribute:** `IltObject.AlarmCountNumberAttribute`
- ◆ **New Alarm Highest Severity** indicates the highest severity of the new raw alarms or traps raised on the business object. This attribute determines the color of the alarm balloon. It is computed from the object state and should not be set to a value directly.
 - **Name:** `newAlarmHighestSeverity`
 - **Value class:** `ilog.tgo.model.IltAlarmSeverity`
 - **Attribute:** `IltObject.NewAlarmHighestSeverityAttribute`
- ◆ **Alarm Highest Severity** indicates the highest severity of the outstanding raw alarms or traps raised on the business object. Outstanding refers to both acknowledged and new alarms or traps. This attribute determines the color of the alarm border. It is computed from the object state and should not be set to a value directly.
 - **Name:** `alarmHighestSeverity`
 - **Value class:** `ilog.tgo.model.IltAlarmSeverity`
 - **Attribute:** `IltObject.AlarmHighestSeverityAttribute`
- ◆ **Ack Alarm Highest Severity** indicates the highest severity of the acknowledged raw alarms or traps of the business object. By default, this attribute is not displayed in the table. It is computed from the object state and should not be set to a value directly.

- **Name:** `ackAlarmHighestSeverity`
- **Value class:** `ilog.tgo.model.IltAlarmSeverity`
- **Attribute:** `IltObject.AckAlarmHighestSeverityAttribute`
- ◆ **New Impact Alarm Count** indicates the number of new impact alarms of the business object. This string is displayed in the alarm balloon of the object. The new impact alarm count is computed from the object state and should not be set to a value directly.
 - **Name:** `newImpactAlarmCount`
 - **Value class:** `ilog.tgo.model.IltAlarmCountAttributeType`
 - **Attribute:** `IltObject.NewImpactAlarmCountAttribute`
- ◆ **New Impact Alarm Count Number** indicates the number of new impact alarms of the business object. This integer represents the number of new impact alarms. The new impact alarm count is computed from the object state and should not be set to a value directly.
 - **Name:** `newImpactAlarmCountNumber`
 - **Value class:** `java.lang.Integer`
 - **Attribute:** `IltObject.NewImpactAlarmCountNumberAttribute`
- ◆ **Impact Alarm Count** indicates the number of outstanding impact alarms of the business object. Outstanding refers to both acknowledged and new alarms. This string displays on the object base. The impact alarm count is computed from the object state and should not be set to a value directly.
 - **Name:** `impactAlarmCount`
 - **Value class:** `ilog.tgo.model.IltAlarmCountAttributeType`
 - **Attribute:** `IltObject.ImpactAlarmCountAttribute`
- ◆ **Impact Alarm Count Number** indicates the number of outstanding impact alarms of the business object. Outstanding refers to both acknowledged and new alarms. This integer represents the number of outstanding impact alarms. The impact alarm count is computed from the object state and should not be set to a value directly.
 - **Name:** `impactAlarmCountNumber`
 - **Value class:** `java.lang.Integer`
 - **Attribute:** `IltObject.ImpactAlarmCountNumberAttribute`
- ◆ **New Impact Alarm Highest Severity** indicates the highest severity of the new impact alarms raised on the business object. This attribute determines the color of the alarm balloon. It is computed from the object state and should not be set to a value directly.
 - **Name:** `newImpactAlarmHighestSeverity`
 - **Value class:** `ilog.tgo.model.IltAlarmSeverity`

- **Attribute:** `IltObject.NewImpactAlarmHighestSeverityAttribute`
- ◆ **Impact Alarm Highest Severity** indicates the highest severity of the outstanding impact alarms raised on the business object. Outstanding refers to both acknowledged and new alarms. This attribute determines the color of the alarm border. It is computed from the object state and should not be set to a value directly.
 - **Name:** `impactAlarmHighestSeverity`
 - **Value class:** `ilog.tgo.model.IltAlarmSeverity`
 - **Attribute:** `IltObject.ImpactAlarmHighestSeverityAttribute`
- ◆ **Ack Impact Alarm Highest Severity** indicates the highest severity of the acknowledged impact alarms of the business object. By default, this attribute is not displayed in the table. It is computed from the object state and should not be set to a value directly.
 - **Name:** `ackImpactAlarmHighestSeverity`
 - **Value class:** `ilog.tgo.model.IltAlarmSeverity`
 - **Attribute:** `IltObject.AckImpactAlarmHighestSeverityAttribute`
- ◆ **Primary State** indicates the primary state of the business object. This attribute determines the base style. It is computed from the object state and should not be set to a value directly.
 - **Name:** `primaryState`
 - **Value class:** `String`
 - **Attribute:** `IltObject.PrimaryStateAttribute`
- ◆ **Secondary States** indicates the secondary states or statuses of the business object. This attribute determines the small icons displayed at the top left of the base. It is computed from the object state and should not be set to a value directly.
 - **Name:** `secondaryStates`
 - **Value class:** `String`
 - **Attribute:** `IltObject.SecondaryStatesAttribute`
- ◆ **Tiny Type** indicates the way the object will be displayed in the tiny representation. By default, this attribute does not appear in the table.
 - **Name:** `tinyType`
 - **Value class:** `ilog.tgo.model.IltObject.TinyType`
 - **Attribute:** `IltObject.TinyTypeAttribute`

Computed attributes based on the object state

The `IlpDefaultClass IltObject` business class includes a number of predefined attributes that make it possible to represent the state of the object in the table component. However, you might want to display other information that is contained in the object state in a table column. To do so, you can define a new computed attribute based on the object state.

The following example shows how to define a new computed attribute that returns the number of major new alarms.

How to define a new computed attribute

```
IlpAttribute NewMajorAlarmAttribute =
    new IltComputedAttribute("newMajorAlarmAttribute",
                            String.class) {
    public Object getValue (IlpAttributeValueHolder h) {
        IltObjectState oState =
            (IltObjectState)h.getAttributeValue(IltObject.ObjectStateAttribute);

        IltAlarm.State alarmState = oState == null
            ? null : (IltAlarm.State)oState.getAlarmState();
        if ( alarmState == null ) return null;
        return alarmState.getNewAlarmCount(IltAlarm.Severity.Major);
    }

    public boolean isDependentOn (IlpAttribute a) {
        return a.getName().equals(ObjectStateAttribute.getName());
    }
};
```

Extending predefined business classes

To extend a predefined business class dynamically, you can:

- ◆ Create a new instance of `IlpDefaultClass` directly or create it from an XML description of the class.
- ◆ Create a new Java™ class and its associated `IlpClass`.

Creating a subclass of a predefined business class dynamically

The following example demonstrates how to create a subclass of a predefined business class in XML.

How to create a subclass of a predefined business object in XML

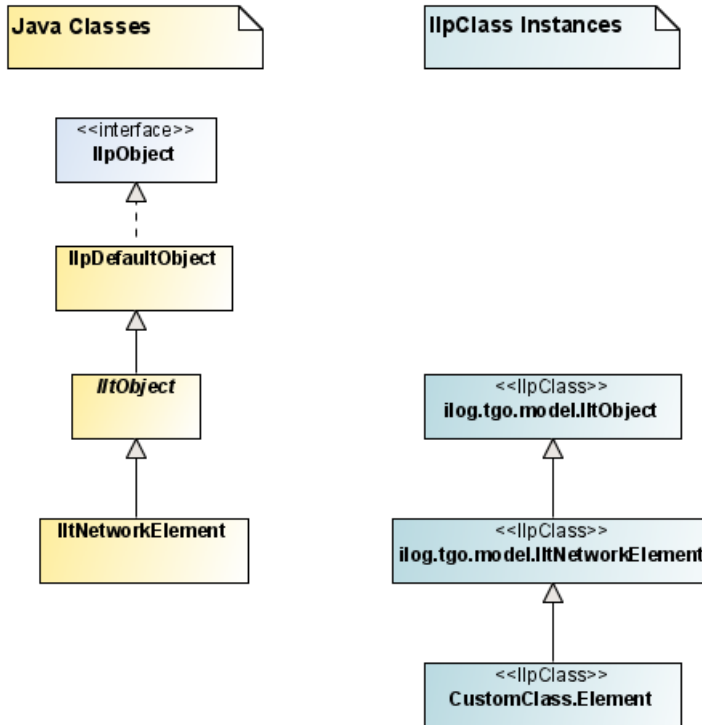
```
<class>
  <name>Element</name>
  <superClass>ilog.tgo.model.IltNetworkElement</superClass>
  <attribute>
    <name>throughput</name>
    <javaClass>java.lang.Integer</javaClass>
  </attribute>
</class>
```

The syntax is the same as for creating a regular class with XML. See *Defining the business model in XML*.

The newly created class extends the `IltNetworkElement` predefined business class and has an additional attribute (`throughput`).

This class is an instance of both the `IlpClass` `Element` and the Java class `ilog.tgo.model.IltNetworkElement`.

As shown in the following figure, an instance of `Element` derives from two class hierarchies: the dynamic class hierarchy and the Java class hierarchy.



Deriving instances from the dynamic and the Java class hierarchies

If you create an instance of the `IIPClass` `Element` with XML or with the API, the created object will be an instance of the Java class `IIPNetworkElement`.

For example:

How to extend predefined business object classes for use with the Java API

```

IIPClass elementClass =
    classManager.getClass("Element");
IIPObject element = elementClass.newInstance("element 1", true);
  
```

The method `newInstance` is available in business classes to create new instances. This method has two arguments:

- ◆ object identifier: a unique object identifier used by the new instance
- ◆ boolean `initializeAttributeValues`: a flag which indicates whether the new instance has its default attribute values initialized. It is important to note that predefined business objects and their subclasses always have their default attributes initialized when a new instance is created. In this case, the second parameter is ignored.

Creating a Java subclass of a predefined business class

Creating a Java subclass of a predefined business class is similar to what is described in *Adding predefined business objects* with the following differences:

- ◆ Methods `getIdentifier`, `getIlpClass`, `getAttributeValue` and `setAttributeValue` should not be overridden.
- ◆ You must implement the following constructors:

```
public MyClass (Object identifier) {
    super(identifier);
}
```

and

```
public MyClass (IlpClass ilpclass, Object identifier) {
    super(ilpclass, identifier);
}
```

- ◆ You must call the superclass in the constructor.
- ◆ You must create a static instance of `IltObjectInfo` that will store the business class information so that it is recognized as an JViews TGO business class.
- ◆ You must implement the method `IlpClass`. This method allows your new business class to be automatically recognized by the Class Manager service.
- ◆ Your new accessor methods (for example, `getThroughput/setThroughput`) should call `getAttributeValue` and `setAttributeValue` with the appropriate parameters. These methods already provide the mechanism to store the objects internally as well as notification support.

The following example illustrate the implementation of a new business object class that inherits from `IltNetworkElement`. This new business class contains a new attribute, called `THROUGHPUT`.

How to create a new business class from a predefined business class

```
public class CustomNetworkElement extends IltNetworkElement {

    // Create the business class
    static IltObjectInfo metainfo = new IltObjectInfo(CustomNetworkElement.class,

        "CustomNetworkElement");

    // Create the business attribute and register in the class
    public static final IlpAttribute THROUGHPUT = new IltAttribute("throughput",

                                                                    Integer.class,
```

```

                                                                    metainfo,
                                                                    new
                                                                    Integer(0));

// Register the new attribute in this business class
static {
    metainfo.addAttribute(THROUGHPUT);
}

// Implement method GetIlpClass so that this class is automatically
// recognized as a business class by the Class Manager service
public static IltObjectInfo GetIlpClass() {
    return metainfo;
}

// Implement the class constructor
public CustomNetworkElement (Object identifier) {
    super(identifier);
}

// Implement the class constructor
public CustomNetworkElement (IlpClass clazz, Object identifier) {
    super(clazz, identifier);
}

public int getThroughput() {
    Object v = getAttributeValue(THROUGHPUT);
    if (v == ilog.cpl.model.IlpAttributeValueHolder.VALUE_NOT_SET)
        return 0;
    else
        return ((Integer)v).intValue();
}

public void setThroughput(int throughput) {
    setAttributeValue(THROUGHPUT, new Integer(throughput));
}
}

```

The business model

Goes into the details of the business model, business classes and business objects.

In this section

Business model, business classes, and business objects

Summarizes the relationship between business model, business classes, and business objects.

Integrating the business model with the back end

Describes the various approaches to integrate a JViews TGO business model with the back-end application.

Defining the business model in XML

Explains how to create dynamic classes by describing them in an XML file that will be loaded at runtime.

Business model API

Describes the API of the components of a business model: business classes, business objects, and attributes.

Business class manager API

Describes the business class manager interface and its implementations.

Defining the business model from JavaBeans classes

Describes how to make use of the JavaBean wrappers provided by JViews TGO.

Defining the business model with dynamic classes

Describes how to define a business model from any Java™ class.

Business model, business classes, and business objects

The business model allows the back-end application and JViews TGO to share a common vocabulary. In other words, it translates the real application in the back end to a data model that JViews TGO can understand. The business model describes business classes, their inheritance and their attributes.

In the JViews TGO business model, classes are defined as instances of the `IlpClass` interface. You can retrieve the `IlpClass` corresponding to a given Java™ class. You can also create `IlpClass` instances dynamically, either by using the API or by loading a class description written in XML.

Instances of business classes or, in short, *business objects* are defined by the `IlpObject` interface.

The purpose of business objects is to allow you to map application data to JViews TGO graphic components in a flexible way and to make it possible to reuse the same data across multiple components, thus providing homogeneous graphical representations throughout an application.

JViews TGO provides a set of predefined business objects classes that you can use directly in your applications. These classes are described in detail in the following sections of this documentation.

Integrating the business model with the back end

There are different ways of integrating a JViews TGO business model with the back-end application. The approach depends on the requirements of a specific business model and on whether this model is based on Java™ classes or not.

The different possibilities are the following:

1. The easiest and recommended approach consists in mapping your business model with the JViews TGO predefined business classes. JViews TGO provides a set of predefined business classes that you can use directly in your applications. They have been specifically designed to ease the development and to leverage the overall graphic quality and ergonomics of user interfaces in telecommunication applications. Adapting predefined business classes to your own needs allows you to take advantage of the look and feel of these classes.

For a complete list of the predefined business classes, refer to *Introducing business objects and data sources*.

JViews TGO predefined business objects can be used either if your model is based on Java classes or if you prefer to describe the model and its data in XML. For more details, refer to *Introducing business objects and data sources* and to *Adding business objects from JavaBeans*.

Your business model may require more information than available in the predefined business classes. If this is the case, you can easily extend the predefined business classes with your own attributes. For details, refer to *Extending predefined business classes*.

If you cannot easily map your business classes with the JViews TGO predefined business classes, you can still envisage one of the other possibilities to integrate your business model with the back end.

2. You can describe your business model and its data in XML. For more information, refer to *Defining the business model in XML* and *Adding business objects from JavaBeans*.
3. You can describe your business model as Java classes that comply with the JavaBeans™ pattern. JViews TGO provides direct support for JavaBeans business classes. For more information, refer to *Defining the business model from JavaBeans classes* and *Adding business objects from JavaBeans*.
4. You can describe your business model as Java classes either by using the default business class (`IlpDefaultClass`) and business object (`IlpDefaultObject`) implementations, or by implementing the corresponding interfaces (`IlpClass`, `IlpObject`) directly. For more information, refer to *Defining the business model with dynamic classes* and *Adding dynamic business objects*.

Whatever the approach you choose to integrate an JViews TGO business model with the back-end application, all the objects will be represented with a default look and feel in the graphic components. If this look and feel does not suit you, you can customize it by using Cascading Style Sheets (CSS). For more information on how to customize the graphic representation of business objects, refer to *Using Cascading Style Sheets*.

Defining the business model in XML

Explains how to create dynamic classes by describing them in an XML file that will be loaded at runtime.

In this section

Defining a dynamic class in XML

Explains how to create a dynamic class in XML and describes the notions of inheritance, attribute types, and type conversion.

Extending a predefined business class in XML

Explains how to extend a predefined class in XML.

Loading the business model

Describes the two methods for loading a business model: at application start up or from a data source.

Defining a dynamic class in XML

The XML file may be loaded at the beginning of the program or at a later stage. In the same way as you can load a model from a file, you can also load the corresponding data from a file. You will find an example of a model and the corresponding data defined in the same file in *Adding business objects from JavaBeans*.

The example below shows how to describe the dynamic classes `Event` and `Alarm` in XML format. These classes will be created when the XML file is loaded.

The `Event` class has the `ID` attribute of type `string`. The `Alarm` class is a subclass of the `Event` class that has two attributes, `PerceivedSeverity`, of type `String`, and `Acknowledged`, a Boolean attribute that defaults to `false`. The `Alarm` class inherits the `ID` attribute from the `Event` class. For details, see *Inheritance*.

How to define dynamic classes in XML

```
<classes xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation = "ilog/cpl/schema/model.xsd">
<classes>
  <class>
    <name>Event</name>
    <attribute>
      <name>ID</name>
      <javaClass>java.lang.String</javaClass>
    </attribute>
  </class>
  <class>
    <name>Alarm</name>
    <superClass>Event</superClass>
    <attribute>
      <name>PerceivedSeverity</name>
      <javaClass>java.lang.String</javaClass>
    </attribute>
    <attribute>
      <name>Acknowledged</name>
      <javaClass>java.lang.Boolean</javaClass>
      <defaultValue>>false</defaultValue>
    </attribute>
  </class>
</classes>
```

The following table describes the elements that you can use to define your business model in XML format. You can also find a description of this format in the XML schema file **model.xsd**.

Elements in an XML business model

XML element	Attributes	Default	Description
<classes>	None		Delimits classes definition. This element is required. The file containing classes should necessarily start and end

XML element	Attributes	Default	Description
			with this element. This element can be used inside data (see <i>Elements in an XML data file</i>). This element contains <class> elements.
<class>	None		Contains a class definition. The class definition contains a name and possibly a <superClass> element and some <attribute> elements.
<superClass>	None		Contains the name of the superclass.
<name>	None		Contains the name of the class or of the attribute being defined.
<attribute>	None		Defines an attribute in the current <class>. The attribute contains a <name>, a <javaClass>, and optionally a <defaultValue>.
<javaClass>	None		Contains the name of the Java™ class of the attribute (for example <code>java.lang.String</code>).
<defaultValue>			Contains the default value of an attribute.
	javaClass	The Java class of the attribute.	Sometimes the Java class of the default value is not the Java class of the attribute (for example, if the Java class of the attribute is abstract). This attribute allows you to specify the Java class of the default value.
	null	false	If the default value is to be null, you can set this optional attribute to <code>true</code> .

Inheritance

The dynamic classes created from an XML file can inherit from existing classes, in the same way as the dynamic classes that you can create using the Java API, which are either classes created from JavaBean™ classes or custom `IlpClass` implementations retrieved by the static `GetIlpClass()` method.

The order in which classes are defined within the element <classes> has no impact when loading classes that have inheritance relationships. In other words, you can define a class before a superclass within the same <classes> element.

Attribute types

The JViews TGO schema for defining an XML business model provides a <javaClass> element that lets you assign an abstract class or interface type to an attribute (for example `java.lang.Number`) and provide a default value.

How to assign an abstract class or interface type and default value to an attribute

```
<attribute>
```

```
<name>number</name>
<javaClass>java.lang.Number</javaClass>
<defaultValue javaClass="java.lang.Integer">100</defaultValue>
</attribute>
```

You may want to specify that an attribute has a null default value. Sometimes it is difficult to distinguish between a null value and an empty value. For example, if you have an attribute of type `String`, its default value will be `""`.

How to specify a null default value for an attribute

```
<attribute>
  <name>emptyString</name>
  <javaClass>java.lang.String</javaClass>
  <defaultValue />
</attribute>
```

Therefore, there is another XML attribute for default values to specify that the default value is null:

```
<attribute>
  <name>nullString</name>
  <javaClass>java.lang.String</javaClass>
  <defaultValue null="true" />
</attribute>
```

Type conversion

The types assigned to attributes of dynamic classes defined in an XML file should be recognized by the XML parser in order to load data contained in these classes.

Since attribute types may be complex, JViews TGO supports both simple attribute types, which can be read from a single string, and complex attribute types which can be composed of several XML tags.

For converting simple types, the XML parser uses the type converter service of the current application context (see `Type converter`). It executes the type conversion by calling the methods `createJavaInstance` and `createStringValue`. These methods support all the types defined in `java.lang`, plus extra ones such as:

- ◆ Dates (instances of `java.util.Date`). The following format is supported:
"yyyy'-MM'-dd'T'HH':mm':ss"
- ◆ Colors (instances of `java.awt.Color`). Regular HTML formats are supported; for example:
"RED" or "#NNNNNN"
- ◆ Fonts (instances of `java.awt.Font`). To know the formats that are supported, see the `java.awt.Font.decode` method.
- ◆ Enumerated values (instances of `ilog.util.IEnum`).

You can extend the number of types supported by the default type converter by creating specific property editors or by subclassing the type converter.

Complex types

The interface `IlpSAXSerializable` supports complex types. This interface lets you read an instance from a SAX `XmlReader` and write it as a SAX event to a SAX `ContentHandler`. To be recognized, this interface should be implemented by a class named `<className>SAXInfo`; or, if you are using the default type converter implementation (`IlpDefaultTypeConverter`), you can register the value handler so that it is automatically taken into account by the XML parser, using the method `IlpDefaultTypeConverter.setAttributeValueHandler(java.lang.Class, ilog.cpl.storage.IlpSAXSerializable)`.

Most of the JViews TGO classes that can be used as attribute types have a corresponding SAXInfo class. For example, the `IlpPoint` class in the `ilog.cpl.graphic` package comes with the `IlpPointSAXInfo` class. When an `IlpPoint` is to be read, the default type converter checks whether there is a corresponding `IlpSAXSerializable` instance. If not, it will try to load the `IlpPointSAXInfo` class. If it succeeds, this class will be returned as the `IlpSAXSerializable` interface of the `IlpPoint` class.

As an example, here is the code of the class `IlpPointSAXInfo`.

How to use the `IlpSAXSerializable` interface with complex types

```
/**
 * Returns a SAX handler capable of reading the attribute.
 * To avoid an instance of a SAX handler being used simultaneously
 * by two concurrent threads, this method should either return a new
 * instance of a SAX handler each time it is called or take advantage of
 * the java.lang.ThreadLocal class to return a different
 * instance for each thread.
 * @return A SAX event handler.
 */
public IlpSAXAttributeValueHandler getSAXHandler() {
    return new IlpSAXPointHandler();
}

public static class IlpSAXPointHandler extends IlpSAXAttributeValueHandler
{
    protected float x;
    protected float y;
    /**
     * Indicates the beginning of an element.
     * By default, this method only clears the content of the element.
     */
    public void startElement(String namespaceURI,
                            String localName,
                            String qName,
                            Attributes atts) throws SAXException {
        if (localName.equals(finalTag)) {
            x = y = 0.0f;
        }
        super.startElement(namespaceURI, localName, qName, atts);
    }
}
```

```

/**
 * Notifies the end of an element.
 * When this method is called for the "attribute" element,
 * the <CODE>getAttributeValue</CODE> method is called, the object
 * is modified accordingly, and parsing continues.
 */
public void endElement(String namespaceURI,
                      String localName,
                      String qName) throws SAXException {
    if (localName.equals("x")) {
        x = Float.parseFloat(getContent());
    }

    if (localName.equals("y")) {
        y = Float.parseFloat(getContent());
    }

    super.endElement(namespaceURI, localName, qName);
}

/**
 * Is called at the end of the "attribute" element.
 */
protected Object getAttributeValue() {
    Object value = new IlpPoint(x,y);
    return value;
}

}

/**
 * Writes the <CODE>value</CODE> to a SAX ContentHandler.
 * The method translates the object as SAX ContentHandler method calls.
 * @param value The object to write.
 * @param typeConverter The type converter that may be needed to translate
 * values to strings.
 * @param outputHandler The SAX ContentHandler used by this method.
 * @see ilog.cpl.util.IlptypeConverter#createStringValue(Object,IlpKey)
 */
public void output(Object value,
                  IlpTypeConverter typeConverter,
                  ContentHandler outputHandler)
    throws SAXException {
    IlpPoint point = (IlpPoint)value;
    String valueStr = String.valueOf(point.x);
    outputHandler.startElement("", "x", "x", IlpSAXSerializable.EMPTY_ATTRS);
    outputHandler.characters(valueStr.toCharArray(), 0, valueStr.length());
    outputHandler.endElement("", "x", "x");
    valueStr = String.valueOf(point.y);
    outputHandler.startElement("", "y", "y", IlpSAXSerializable.EMPTY_ATTRS);
    outputHandler.characters(valueStr.toCharArray(), 0, valueStr.length());
    outputHandler.endElement("", "y", "y");
}

```

```
}  
}
```

Extending a predefined business class in XML

To extend a predefined business class dynamically, you can create a new instance of `IlpDefaultClass` from an XML description of the class.

The following example demonstrates how to create a subclass of a predefined business class in XML.

How to create a subclass of a predefined business class in XML

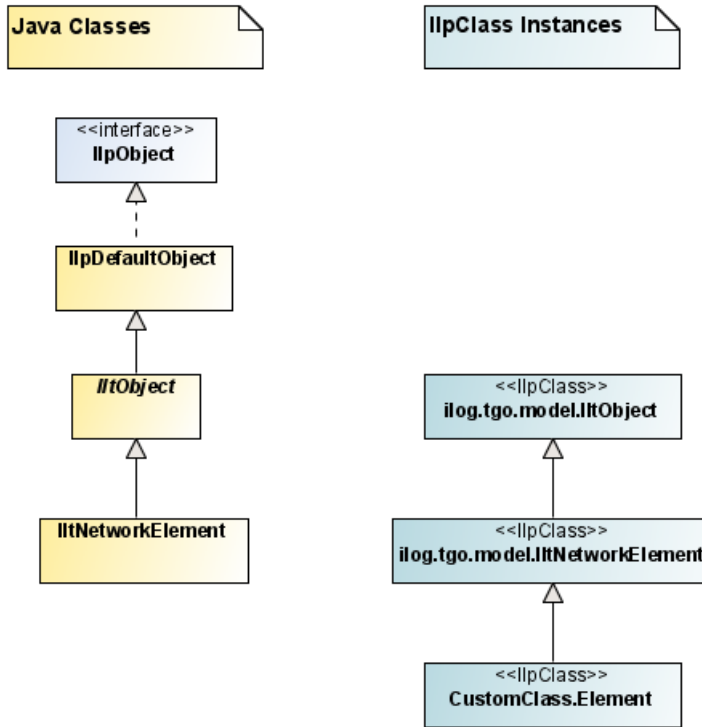
```
<class>
  <name>Element</name>
  <superClass>ilog.tgo.model.IltNetworkElement</superClass>
  <attribute>
    <name>throughput</name>
    <javaClass>java.lang.Integer</javaClass>
  </attribute>
</class>
```

The syntax is the same as for creating a regular class with XML. See *Defining the business model in XML*.

The newly created class extends the `IltNetworkElement` predefined business class and has an additional attribute (`throughput`).

This class is an instance of both the `IlpClass Element` and the Java™ class `ilog.tgo.model.IltNetworkElement`.

As shown in the following figure, an instance of `Element` derives from two class hierarchies: the dynamic class hierarchy and the Java class hierarchy.



Deriving instances from the dynamic and the Java class hierarchies

If you create an instance of the IIPClass Element with XML, the created object will be an instance of the Java class IIPNetworkElement.

Loading the business model

Dynamic classes can be loaded either when the application starts up, that is, when `Iltsystem.Init()` is called, or later when data is loaded in a data source. See *Adding business objects from JavaBeans*.

Loading at application start up

To load a model when the application is launched, you can modify the deployment descriptor by adding the following lines inside the `<deployment>` element:

```
<classManager>
  <file>mymodel.xml</file>
</classManager>
```

The `<classManager>` element can include a number of `<file>` tags. The path to these files is resolved by means of the URL access service, which is created and configured by the deployment descriptor.

For more information, see [Class manager](#).

Loading with the data in a data source

You can insert business model XML elements, that is, complete `<classes>` elements, in a data source file. For details, refer to *Adding business objects from JavaBeans*.

Business model API

Describes the API of the components of a business model: business classes, business objects, and attributes.

In this section

Class overview

Provides a diagram of the different classes involved in a business model.

Business class API

Describes the business class interfaces and their implementations.

Business object API

Describes the business object interfaces and their implementations.

Attribute API

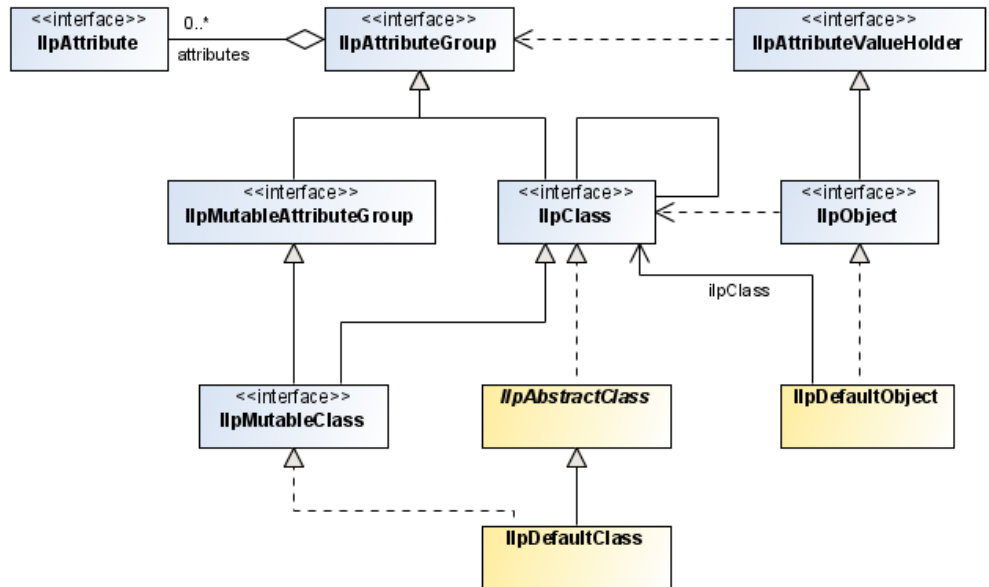
Describes the attribute-related interfaces and their implementations.

Class overview

The business model API is composed of the following APIs:

- ◆ *Business class API*
- ◆ *Business object API*
- ◆ *Attribute API*

The following figure illustrates the various elements that compose the business object model.



The business model API

Business class API

The interface `IlpClass` defines static classes. It provides methods to:

- ◆ Retrieve the class structure, that is, its superclass and subclasses
- ◆ Retrieve the associated attributes, that is, the attribute group

The interface `IlpMutableClass` defines dynamic classes. In addition to the methods listed above, it provides methods to:

- ◆ Add and remove attributes
- ◆ Aggregate attribute groups to the class
- ◆ Notify listeners about these modifications

JViews TGO provides a few convenience implementations of these interfaces that you can use directly or subclass when building an application:

- ◆ `IlpAbstractClass`—An abstract implementation that helps you create new `IlpMutableClass` implementations.
- ◆ `IlpDefaultClass`—A default implementation for a dynamic class.
- ◆ `IlpBeansClass`—A wrapper for an existing `JavaBean™` class that makes it possible to access it (through introspection) as if it were a dynamic class. Properties discovered during introspection are automatically translated to `IlpBeansAttribute` instances and inserted in the class attribute group.

Business object API

The interface `IlpObject` defines instances of business classes. These instances (business objects) contain values for the attributes defined by the corresponding class. They also have an attribute group that can be the `IlpClass` itself or an extension allowing you to define new attributes at the object level.

Each business object must be assigned to an identifier when it is created. The identifier must be unique (even across data sources); it is used to identify and retrieve the `IlpObject`. The object identifier can be of any type, as long as it satisfies the following constraints:

1. The identifier must be convertible to `String`, through an `IlpTypeConverter` installed in the `IlpContext`. The `IlpDefaultTypeConverter` uses the `Object.toString()` method to convert objects of unknown types to `String`. The `String` that results from this conversion must be unique.
2. It must be possible to create or retrieve the identifier `Object`, given the corresponding `String`, by using the `IlpTypeConverter` installed in the `IlpContext`. The `IlpDefaultTypeConverter` uses a constructor with a single `String` argument to create instances of unknown types.
3. The identifier `Object` itself must not be an instance of `IlpObject`.

Note the following recommendations:

- ◆ Ideally the identifier should be implemented as a simple object in order to perform a fast comparison. For the sake of simplicity, it is recommended to use simple objects, such as `Number` or `String`, which already satisfy the constraints listed above.
- ◆ If you are using your own Java™ class as identifier, the public `boolean equals(Object obj)` method of the identifier class should be overridden to perform an efficient comparison of identifier instances. The public `int hashCode()` method should also be overridden and must be consistent with the public `boolean equals(Object obj)` implementation.
- ◆ An identifier generator can be used to generate the object identifiers in a consistent manner. A predefined identifier generator is available in `ilog.cpl.util.IlpIDGenerator`.

Note that once the proper implementation for the `IlpTypeConverter` is created (normally you do this by subclassing `IlpDefaultTypeConverter`), you can register it with the `IlpContext` implementation as follows:

```
IlpContext context = ...;
context.addService(IlpTypeConverter.class, new CustomTypeConverter(context));
```

For more information about how to customize and extend the behavior of the default type converter, refer to [Type converter](#) and to *Complex types*.

The `IlpObject` interface has methods to:

- ◆ Retrieve the object identifier
- ◆ Retrieve the corresponding business class

- ◆ Set/get the attribute values, retrieve the attribute group and notify interested listeners, according to the interface `IlpAttributeHolder`

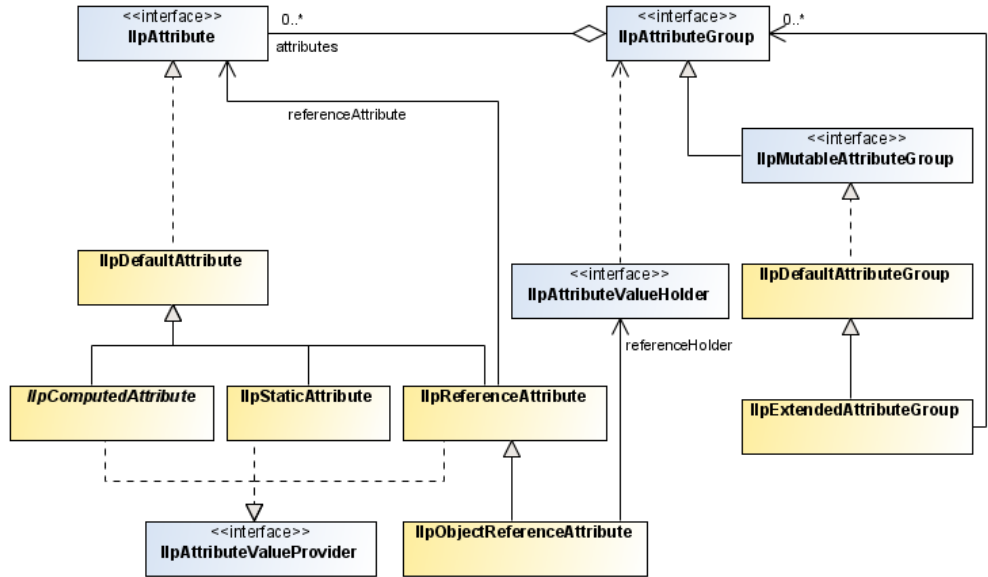
JViews TGO provides the following convenience implementations of this interface that you can use directly when building your application:

- ◆ `IlpDefaultObject`—Default object instance created given a class and an identifier.
- ◆ `IlpBeansObject`—Provides a wrapper for an existing `JavaBean™` instance, which will allow it to be handled as a dynamic object.
- ◆ `IlpObjectSupport`—Provides a default implementation for all `IlpObject` methods. If your Java classes do not follow the `JavaBeans` pattern and you cannot create your business objects by inheriting from `IlpDefaultObject`, you can write your own implementation of the `IlpObject` interface by using the class `IlpObjectSupport`.

Attribute API

Attributes are the properties that qualify a business class. For example, an object of type alarm can be qualified by a severity and a description. An attribute is identified by its name and its value and is typed by a Java™ class.

The following figure illustrates the attribute API:



The attribute API

The interface `IlpAttribute` defines attributes. It provides the following methods:

- ◆ `String getName()` returns the attribute name used as an identifier.
- ◆ `Object getValueClass()` returns the class of the attribute value.
- ◆ `Object getDefaultValue()` returns the default value of the given attribute. The default value is used to initialize the attribute in a newly created business object (see `newInstance(ilog.cpl.model.IlpClass, java.lang.Object, boolean)`).
- ◆ `getAttributeGroup()` `getAttribute(java.lang.String) IlpAttribute.getAttributeGroup()` returns the attribute group that contains the attribute instance.
- ◆ `void setAttributeGroup(IlpAttributeGroup group)` sets the attribute group that contains the instance.
- ◆ `boolean isTransient()` returns whether the attribute should persist or not when the business object is stored.

JViews TGO provides a set of convenience implementations for `IlpAttribute` that you can use directly or subclass in order to obtain application-specific behavior:

- ◆ `IlpDefaultAttribute`—Defines a simple attribute with a name and a value class. The attribute value is not stored in the attribute, but in the object to which this attribute belongs.
- ◆ `IlpStaticAttribute`—Defines an attribute with a static value; in other words, this attribute is the same for all objects that contain it.
- ◆ `IlpReferenceAttribute`—Defines an attribute with a value derived from the value of another attribute in the same object instance.
- ◆ `IlpObjectReferenceAttribute`—Defines an attribute with a value derived from the value of another attribute in another object instance.
- ◆ `IlpComputedAttribute`—Defines an attribute with a value that is calculated from a given formula defined by the user. For more information, see *Computed attributes*.
- ◆ `IlpBeansAttribute`—Provides a wrapper for an existing JavaBean™ class property; this wrapper makes it possible to access the property (through introspection) as if it were a dynamic attribute.

Attribute group

Attributes are logically gathered in *attribute groups*. An attribute group can be static or dynamic. Static attribute groups are defined by the interface `IlpAttributeGroup` and dynamic attribute groups are defined by `IlpMutableAttributeGroup`.

These interfaces provide methods that allow you to perform the following operations:

- ◆ Iterate over the list of attributes
- ◆ Search for a certain attribute given its name
- ◆ Verify whether a given attribute is present in the attribute group
- ◆ Insert and remove attributes
- ◆ Notify interested objects about the modifications described above

JViews TGO provides the following convenience implementations of `IlpAttributeGroup`:

- ◆ `IlpDefaultAttributeGroup`—Defines a simple dynamic attribute group.
- ◆ `IlpExtendedAttributeGroup`—Defines a dynamic attribute group that can be extended with other attribute groups. The attribute groups used are not copied, but simply referred to.

Attribute value holder

In JViews TGO, an attribute value is not carried by the attribute itself but by an *attribute value holder* because this value is specific to the object with which the attribute is associated. An attribute value holder is defined by the `IlpAttributeValueHolder` interface. Business objects and representation objects carry attribute values, and as such they implement this interface. The `IlpAttributeValueHolder` interface includes methods to retrieve and set the value of an attribute, and notify interested objects about changes in attribute values.

Attribute values may be set using the following method:

```
public void setAttributeValue (IlpAttribute attribute, Object value)
```

where value may be null.

In cases where attributes have not been set or initialized, the value is `IlpAttributeValueHolder.VALUE_NOT_SET`.

Attribute values may be retrieved using the following method:

```
public Object getAttributeValue (IlpAttribute attribute)
```

Computed attributes

A computed attribute is an attribute which value is derived from the value of one or more other attributes. To implement a computed attribute, you need to extend the abstract class `IlpComputedAttribute` and implement its abstract part and a constructor. The abstract part of this class is also known as the `IlpAttributeValueProvider` interface.

The `IlpAttributeValueProvider` interface contains the following methods:

```
public Object getValue (IlpAttributeValueHolder h);  
public boolean isDependentOn (IlpAttribute a) ;
```

The `getValue(ilog.cpl.model.IlpAttributeValueHolder)` method returns a value that is computed from its attribute value holder parameter. The `isDependentOn(ilog.cpl.model.IlpAttribute)` method specifies the attributes used to calculate the value of the computed attribute. Whenever the value of one of these attributes changes, the object carrying this attribute notifies its listeners that the computed attribute value has been modified. Note that the computed value is cached. Therefore, calling the `getAttributeValue()` method of `IlpAttributeValueHolder` twice calls the method `IlpComputedAttribute`. `IlpComputedAttribute` only once. This cached value is erased whenever the value of an attribute on which the computed attribute depends is modified.

The example below shows how to define a computed attribute that returns a sorted array of integers calculated from another array of integers.

How to define a computed attribute

```
class SortIntAttribute extends IlpComputedAttribute {  
    IlpAttribute arrayAttribute;  
    public SortIntAttribute(String name,  
        IlpAttributeGroup model,  
        IlpAttribute arrayAttribute) {  
        super(name, model, arrayAttribute.getValueClass());  
        this.arrayAttribute = arrayAttribute;  
    }  
  
    public Object getValue(IlpAttributeValueHolder h) {  
        Object value = h.getAttributeValue(arrayAttribute);  
        if (value != IlpAttributeValueHolder.VALUE_NOT_SET) {  
            int[] array = (int [])h.getAttributeValue(arrayAttribute);
```



```

        array = (int [])array.clone();
        Arrays. sort(array);
        return array;
    }
    return IlpAttributeValueHolder.VALUE_NOT_SET;
}

public boolean isDependentOn (IlpAttribute a) {
    return a == arrayAttribute;
}
}

```

The constructor of `SortIntAttributes` takes three arguments:

- ◆ `name` is the name of the attribute,
- ◆ `model` specifies the attribute group (usually, an `IlpClass`),
- ◆ `arrayAttribute` specifies the attribute that the computed attribute depends on.

The constructor of the abstract class `IlpComputedAttribute` takes three arguments: the name, the model, and the Java class of the value it returns. If the last argument is set to `null`, the computed attribute returns `java.lang.Object`. In this example, this argument is specified and corresponds to the value class of the attribute on which the computed attribute depends.

The `getValue` method computes the attribute value. Computation is protected. As a consequence, if the value of the attribute on which the computed attribute depends is not initialized, or in other words if its value is `IlpAttributeValueHolder`, the method does not perform the computation and returns `IlpAttributeValueHolder.VALUE_NOT_SET`. In this example, when the value is set, the returned array is duplicated, sorted, and returned.

The implementation of the `isDependentOn` method is quite straightforward, since in this example the computed attribute depends only on one other attribute, which is specified in the constructor.

Business class manager API

Once business object classes are created, it is important to make them available to the whole application and allow them to be used by all the components. This is the role of the class manager defined by the interface `IlpClassManager`. This interface provides methods to:

- ◆ Retrieve an `IlpClass` from an identifier
- ◆ Verify whether a given class is part of the business model
- ◆ Retrieve the root classes in the model

For example, the following function writes the entire content of a class manager. It iterates over all the classes stored in a class manager and then over their attributes, displaying the class they belong to along with their default value, if any.

How to write the content of a class manager

```
public void displayModel(IlpClassManager classManager) {
    for (Iterator i = classManager.getClasses().iterator();
         i.hasNext();) {
        IlpClass ilpClass = (IlpClass)i.next();
        System.out.println(ilpClass.getName());
        if (ilpClass.getSuperClass() != null) {
            System.out.println("\t"+ilpClass.getSuperClass().getName());
        }
        for (Iterator j = ilpClass.getAttributes().iterator();
             j.hasNext();) {
            IlpAttribute attr = (IlpAttribute)j.next();
            System.out.println("\t" +attr.getName() +": "
                               +attr.getValueClass().toString());
            if (attr.getDefaultValue() !=
                IlpAttributeValueHolder.VALUE_NOT_SET) {
                System.out.print("\t\tdefault: " +attr.getDefaultValue());
                if (attr.getDefaultValue() != null) {
                    System.out.print(" " +attr.getDefaultValue().getClass());
                }
                System.out.println();
            }
        }
    }
}
```

JViews TGO provides a default implementation of the class manager, which is defined by the class `IlpDefaultClassManager`. This implementation has the following additional functionality:

- ◆ Creates dynamic classes from an XML file. Classes and attributes can be defined by the application using XML and can be loaded in the class manager at runtime.

- ◆ Retrieves a dynamic class from a Java™ class. If the `IlpClass` defining the Java class is not yet present in the class manager, an `IlpBeansClass` is created and the attribute group is automatically built through introspection in the Java class.

Note: You can deactivate this behavior or provide a specific `IlpClass` implementation for your Java classes as described in *Defining the business model with dynamic classes*.

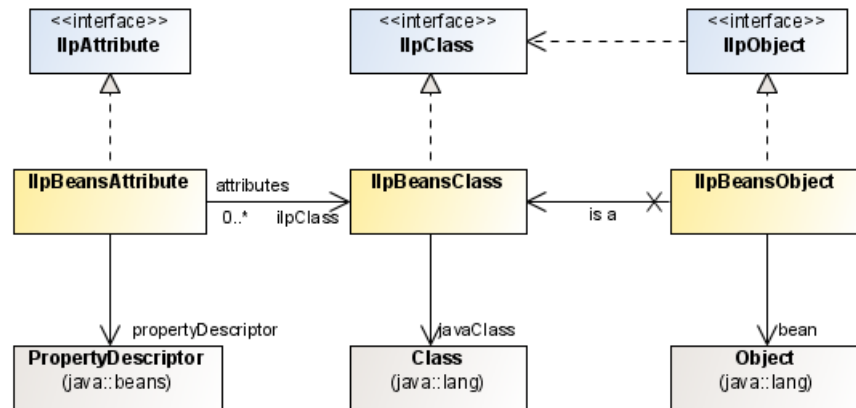
JViews TGO also provides a class manager implementation to automatically handle the predefined business classes. This implementation is defined by class `IltdDefaultClassManager`.

For easier access to the business classes information in your application, the class manager has been defined as one of the application context services. For information on how to customize the class manager information through the application context and the deployment descriptor, refer to *Class manager*.

Defining the business model from JavaBeans classes

If the back-end application is made up of classes that fully comply with the JavaBeans™ pattern, you can integrate these classes easily using the JViews TGO wrapper for existing JavaBeans classes (see *Reminder about JavaBeans design patterns*). This wrapper, defined by `IlpBeansClass`, makes it possible to access the class as if it were a dynamic class. Business object instances also have a corresponding wrapper— `IlpBeansObject`—that allows the user to view them as dynamic objects.

The following figure shows the various JavaBeans wrappers that JViews TGO supplies.



JavaBeans wrappers

Reminder about JavaBeans design patterns

JavaBeans have the following main design patterns:

Properties

Properties can be defined with a pair of `get` and `set` methods. Their names are derived from the method names. The following class contains the "severity" property:

```
public class Alarm {
    public int getSeverity() {...}
    public void setSeverity(int severity) {...}
}
```

If the `getSeverity` method is specified without the `setSeverity` method, the property is readable only.

Bound properties

Each time the value of a bound property changes, other objects are notified accordingly. A `PropertyChange` event is fired specifying the property name, the old value, and the new value. The `IlpBeansObject` class takes advantage of this process, so that each time a bound property changes, its new value is reflected in all the graphic components displaying it.

Mandatory default constructor or serialized instance

A constructor with no parameters should be available on the Bean or a serialized instance should be provided. For more information, see the `instantiate` method of the class `java.beans.Beans`.

The following example shows a Java™ class that conforms to the JavaBeans design pattern.

How to write a Java class that conforms to the JavaBeans design pattern

```
public static class MO {
    String name;
    int state = 0;
    PropertyChangeSupport support = new PropertyChangeSupport(this);
    public MO() {
    }
    public MO(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        String oldName = this.name;
        this.name = name;
        support.firePropertyChange("name", oldName, name);
    }
    public int getState() {
        return state;
    }
    public void setState(int state) {
        int oldState = this.state;
        this.state = state;
        support.firePropertyChange("state", oldState, state);
    }
    public void addPropertyChangeListener(PropertyChangeListener listener) {
        support.addPropertyChangeListener(listener);
    }
    public void removePropertyChangeListener(PropertyChangeListener
listener) {
        support.removePropertyChangeListener(listener);
    }
}
```

To retrieve the corresponding `IlpClass`, execute the following (assuming that you have a class manager):

```
IlpClass moAsIlpClass = classManager.get(MO.getClass().getName());
```

For details, see *Business class manager API*.

For an example of how to use this class in a data source, see *Adding business objects from JavaBeans*.

Defining the business model with dynamic classes

Describes how to define a business model from any Java™ class.

In this section

Defining a dynamic class using the API

Explains how to create new instances of `IlpClass` dynamically.

Extending a predefined business class using the API

Explains how to create a wrapper for a Java object.

Defining a dynamic class using the API

You can create new instances of `IlpClass` dynamically using the Java API.

The easiest way to define a business model from any Java class is to create business classes as instances of `IlpDefaultClass` and register related class attributes with `IlpDefaultAttribute`. For more information, see *Business model API*. The best place to put the `IlpClass` definition is the `IlpObject` implementation.

Here is an example taken from the `FileObject.java` file located in the following sample directory:

```
<installdir>/samples/datasource/explorer2
```

where `<installdir>` is the directory where you have installed JViews TGO.

How to create a business class and register the related class attributes

```
public static IlpDefaultAttribute EXISTS =
    new IlpDefaultAttribute("exists", Boolean.class, true);
public static IlpDefaultAttribute NAME =
    new IlpDefaultAttribute("name", String.class, true);
public static IlpDefaultAttribute PARENT =
    new IlpDefaultAttribute("parent", String.class, true);
public static IlpDefaultAttribute PATH =
    new IlpDefaultAttribute("path", String.class, true);
public static IlpDefaultAttribute DIRECTORY =
    new IlpDefaultAttribute("directory", Boolean.class, true);
public static IlpDefaultAttribute HIDDEN =
    new IlpDefaultAttribute("hidden", Boolean.class, true);
public static IlpDefaultAttribute LASTMODIFIED =
    new IlpDefaultAttribute("lastModified", Long.class, true);
public static IlpDefaultAttribute LASTMODIFIEDDATE =
    new IlpDefaultAttribute("lastModifiedDate", Date.class, true);
public static IlpDefaultAttribute LENGTH =
    new IlpDefaultAttribute("length", Long.class, true);
public static IlpDefaultAttribute ROOT =
    new IlpDefaultAttribute("root", Boolean.class, true);

protected static IlpDefaultClass ILPCLASS;

static {
    ILPCLASS = new IlpDefaultClass("FileObject") {
        /**
         * This method is called when objects are loaded from XML.
         */
        public IlpObject newInstance(IlpClass ilpClass, Object identifier,
            boolean initializeAttributeValues) {
            return new FileObject((File) identifier, ilpClass);
        }
    };
};
```



```

ILPCLASS.addAttribute(EXISTS);
ILPCLASS.addAttribute(NAME);
ILPCLASS.addAttribute(PARENT);
ILPCLASS.addAttribute(PATH);
ILPCLASS.addAttribute(DIRECTORY);
ILPCLASS.addAttribute(HIDDEN);
ILPCLASS.addAttribute(LASTMODIFIED);
ILPCLASS.addAttribute(LASTMODIFIEDDATE);
ILPCLASS.addAttribute(LENGTH);
ILPCLASS.addAttribute(ROOT);
}

```

In this sample code, you have created the `IlpClass` as an `IlpDefaultClass` with a specific implementation of the `newInstance` method. This method is used to create objects read from an XML file. Its implementation depends entirely on the case you have to handle. Note that you do not need to implement the `newInstance` method if you do not intend to use XML with your `IlpClass`.

Here the `FileObject` class uses a `File` as its identifier with all its content. The `IlpClass` parameter allows you to subclass `IlpClass`. The parameter `initializeAttributeValues` has the same meaning as in the XML format. See *Elements in an XML data file*. Here you do not use that parameter because there are no default attribute values.

The class manager is not aware of this new `IlpClass`. You have to define a static method called `GetIlpClass` in your implementation of `IlpObject` to have that class registered with the class manager automatically.

If your Java classes include attributes, you can write an implementation of `IlpObject` that will act as a wrapper. This wrapper will delegate methods to the underlying Java object to access its attributes. This implementation of `IlpObject` is described by an `IlpClass`.

The main methods to implement are the following:

- ◆ `public static IlpClass GetIlpClass`—This method should return the corresponding `IlpClass`. This method tells the class manager how to retrieve the `IlpClass` corresponding to the Java class. When its `getClass` method is called with the Java class as its parameter, the class manager loads the Java class from its name (`Class.forName`) and calls the `GetIlpClass` method that retrieves the `IlpClass`. This way, your `IlpClass` is automatically registered with all the class managers that may use it.

Note: If you do not want a given Java class to be considered as a `JavaBean™` and cannot add the static method `GetIlpClass` to that class, you can register an `IlpClass` that has the same name as the Java class before the class name is queried from the class manager.

- ◆ `public Object getIdentifier`—This method must return an identifier. This identifier can be the underlying Java object, an attribute, or another identifier stored in the implementation of `IlpObject`.
- ◆ `public IlpClass getIlpClass()`—This method returns the `IlpClass`. Usually, you have to store the `IlpClass` as a Java attribute of your `IlpObject` implementation to support

subclasses. If you do not want to support subclasses, you can return the result of the static `GetIlpClass()` method.

- ◆ `public Object getAttributeValue (IlpAttribute attribute)`—This method returns the value of an attribute. Therefore, it results in a call to a method of the delegate Java object.
- ◆ `public void setAttributeValue (IlpAttribute attribute, Object value)`—This method sets the value of an attribute. It is not meant to be supported for all attributes. If supported, it usually results in a call to a method of the delegate Java object.
- ◆ `public Object getAttributeValue (String attributeName)`—This method returns the value of an attribute, given the attribute name. Therefore, it results in a call to a method of the delegate Java object.
- ◆ `public void setAttributeValue (String attributeName, Object value)`—This method sets the value of an attribute. It is not meant to be supported for all attributes. If supported, it usually results in a call to a method of the delegate Java object.

Extending a predefined business class using the API

The wrapper class `FileObject` implements the `IlpObject` interface directly. You could have built it by extending the `IlpDefaultObject` class, or by using the `IlpAttributeValueHolderSupport`. However, both these classes contain the structure to store an arbitrary number of attributes and thus have a footprint. In this example, since the underlying `File` instance stores all the attributes, there is no need for such a structure.

How to create a wrapper for a Java object

```
public class FileObject implements IlpObject {
```

The `FileObject` class contains the definition of the corresponding `IlpClass`. The corresponding code is not repeated here.

This class contains a number of attribute members. It wraps a `java.io.File` and has a `file` attribute, which is also used as the class identifier. This class also contains an `IlpClass` attribute which makes it possible to use it with `IlpClass` instances other than the one it defines (typically subclasses of the one it defines). This class also contains an `AttributeValueChangeSupport` instance that handles notification of attribute changes.

```
protected File file;
protected IlpClass ilpClass;
protected IlpAttributeValueChangeSupport support =
    new IlpAttributeValueChangeSupport(this);
```

The static method `GetIlpClass` allows you to register the `IlpClass` with the class manager automatically. Note that `ILPCLASS` is a static member.

```
public static IlpClass GetIlpClass() {
    return ILPCLASS;
}
```

The class has two constructors. The first one is the easiest to use, as it takes only the `file` to be wrapped as parameter. The second one lets you initialize the `ilpClass` attribute. It corresponds to the method `newInstance(IlpClass ilpClass, Object identifier, boolean initializeAttributeValues)` of the `IlpClass`.

```
public FileObject(File file) {
    this.file = file;
    this.ilpClass = GetIlpClass();
}

public FileObject(File file, IlpClass ilpClass) {
    this.file = file;
    this.ilpClass = ilpClass;
}
```

The following methods add or remove listeners to attribute value changes and fire events. They all delegate to the `IlpAttributeValueChangeSupport` instance.

```
public void addAttributeValueListener (AttributeValueListener l) {
    support.addAttributeValueListener(l);
}

public void removeAttributeValueListener (AttributeValueListener l) {
    support.removeAttributeValueListener(l);
}

public void fireEvent (AttributeValueEvent ev) {
    support.fireEvent(ev);
}
```

The method `getAttributeValue` retrieves attribute values from the `java.io.File` object.

```
public Object getAttributeValue (IlpAttribute attribute) {
    if (attribute == EXISTS) {
        return file.exists()?Boolean.TRUE:Boolean.FALSE;
    }
    if (attribute == NAME) {
        String name = file.getName();
        // for the roots the name may be null, so return the path in this case
        return name != null && name.length() != 0 ?name:file.getPath();
    }
    if (attribute == PARENT) {
        return file.getParent();
    }
    if (attribute == PATH) {
        return file.getPath();
    }
    if (attribute == DIRECTORY) {
        return file.isDirectory()?Boolean.TRUE:Boolean.FALSE;
    }
    if (attribute == HIDDEN) {
        return file.isHidden()?Boolean.TRUE:Boolean.FALSE;
    }
    if (attribute == LASTMODIFIED) {
        return new Long(file.lastModified());
    }
    if (attribute == LASTMODIFIEDDATE) {
        Date date = new Date();
        date.setTime(file.lastModified());
        return date;
    }
    if (attribute == LENGTH) {
        return new Long(file.length());
    }
    if (attribute == ROOT) {
        return file.getParent() == null ?Boolean.TRUE:Boolean.FALSE;
    }
}
```

```
        return null;
    }
}
```

Here the method `setAttributeValue` sets the value for one attribute only and sets off notification about value change.

```
public void setAttributeValue (IlpAttribute attribute, Object value) {
    if (attribute == LASTMODIFIED) {
        if (value instanceof Number) {
            file.setLastModified(((Number)value).longValue());
            fireEvent(new AttributeValueEvent(this, LASTMODIFIED));
        } else
            throw new IllegalArgumentException(value+" is not acceptable as value
of attribute "+attribute);
    }
}
```

The `getIdentifier` method simply returns the wrapped file. The `getIlpClass` method returns the corresponding attribute. The `getIlpAttributeGroup` method also returns the `IlpClass`. The `hasAttributeValue` method is delegated to the attribute group.

The `initializeDefaultValues` method does nothing here. It is invoked to initialize the default values of the attributes and corresponds to the XML attribute `initializeDefaultValue`. See *Elements in an XML data file* for details.

```
public Object getIdentifier() {
    return file;
}

public IlpClass getIlpClass() {
    return ilpClass;
}

public IlpAttributeGroup getAttributeGroup () {
    return getIlpClass();
}

public boolean hasAttributeValue (IlpAttribute a) {
    return getAttributeGroup().hasAttribute(a);
}

public void initializeDefaultValues() {
    // nothing to do here
}
```


Data sources

Explains how the various formats of business objects are handled at the data source level.

In this section

About data sources

Gives a brief description of the purpose of a data source.

Data source API

Describes the different data source interfaces with their usage and provides a class diagram.

Adding business objects from XML

Explains how the default implementation of the data source can create `IlpObject` instances from XML files.

Adding business objects from JavaBeans

Shows how to add JavaBean objects to a data source.

Adding dynamic business objects

Explains how to create dynamic business objects, add them to a data source, and remove them when they are no longer required.

Defining business object relationships

Explains how to define relationships such as links and containment in a data source.

Grouping changes in batches

Explains how to make changes to the contents of a data source or to business objects by grouping them into batches.

Advanced parsing and writing of a data source

Explains how to use the `IlpDataSourceLoader` and `IlpDataSourceOutput` classes.

Implementing a new data source

Shows how to implement a data source for loading on demand a hierarchy of objects and, more precisely, a file system.

About data sources

In IBM® ILOG® JViews TGO, the GUI part of the application connects to the back-end application where it retrieves data to be displayed through a data source.

The role of the data source is to turn business data, whatever the format, into objects that JViews TGO can handle, that is, instances of `IlpObject`. For details, see *Business model API*.

JViews TGO has been designed so that it can integrate smoothly with back-end data in a large variety of formats. JViews TGO plugs to the back-end application through data sources that are specific to the type of data handled.

A data source is a collection of business objects of the class `IlpObject` that notifies its listeners when an object is added or removed or when the object structure is modified. A data source is defined by the `IlpDataSource` interface.

Data source API

JViews TGO provides a default data source implementation, `IltDefaultDataSource`, to connect to XML files. This default data source includes methods for adding or removing business objects implementing the `IlpObject` interface or JavaBeans™, that is, Java™ objects that comply with the JavaBeans design pattern (see *Reminder about JavaBeans design patterns*). It also provides an API that lets you obtain structural information about the existence of a parent, children, an origin or a destination from the back-end and store it in the `IlpObject`.

To learn how to create a custom data source, see *Implementing a new data source*.

The interface `IlpDataSource` defines static data sources. It provides methods to:

- ◆ Retrieve objects based on their identifier.
- ◆ Retrieve all the objects stored in the data source.
- ◆ Notify listeners about objects added or removed from the data source.
- ◆ Retrieve structural information, such as parent, child and containment relationships.

The interface `IlpMutableDataSource` defines dynamic data sources. In addition to the methods listed above, it provides methods to:

- ◆ Add and remove objects.
- ◆ Set structural information, such as parent, child and containment relationships.

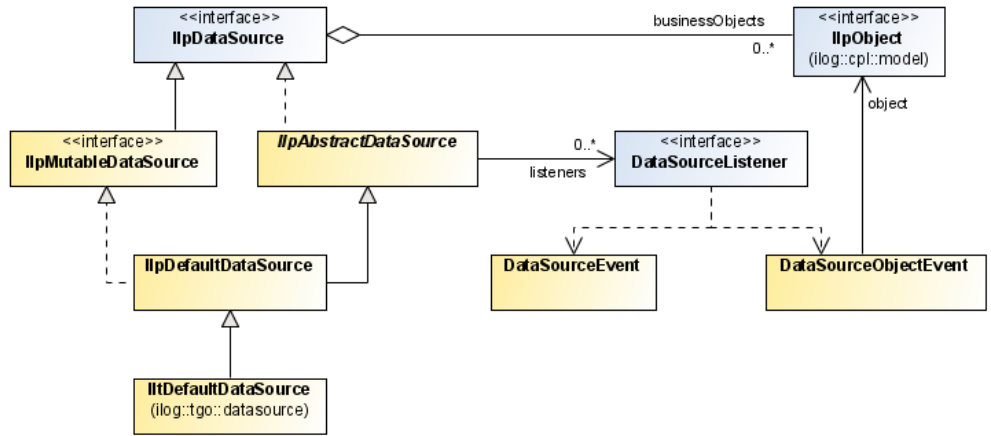
JViews TGO provides a convenience implementation of these interfaces that you can use directly when building an application: `IltDefaultDataSource`. This default implementation for a dynamic data source is able to read business model information from XML and to manage custom business objects as well as predefined business objects.

The following sample code shows how to create a data source instance to be used with the different JViews TGO graphic components.

How to create a data source

```
IlpContext context = IltSystem.GetDefaultContext();
IltDefaultDataSource dataSource = new IltDefaultDataSource(context);
```

A diagram of the data source classes and of their relationships is given in the following figure.



Data source classes

Adding business objects from XML

Explains how the default implementation of the data source can create `IlpObject` instances from XML files.

In this section

Reading an XML file into a data source

Gives examples of correct and incorrect XML business models and explains how to read them into a data source.

Writing the data source content to XML

Explains how to write a data source into an XML file.

Adding predefined business objects

Lists the predefined business objects that you can add into a data source.

Reading an XML file into a data source

The following sample code reads an XML file into a data source.

How to read an XML file into a data source

```
try {
    dataSource.parse("XMLFileExample.xml");
} catch (Exception e) {
    e.printStackTrace();
}
```

Following is an example of an XML file.

```
<cplData xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation = "ilog/cpl/schema/data.xsd">
<cplData>
<classes>
  <class>
    <name>Alarm</name>
    <attribute>
      <name>ID</name>
      <javaClass>java.lang.String</javaClass>
    </attribute>
    <attribute>
      <name>severity</name>
      <javaClass>itest.table.datasourcexmltable.MainFrame$Severity</javaClass>

      <defaultValue>Warning</defaultValue>
    </attribute>
    <attribute>
      <name>acknowledged</name>
      <javaClass>java.lang.Boolean</javaClass>
      <defaultValue>>false</defaultValue>
    </attribute>
  </class>
</classes>
<addObject id="alarm1" initializeDefaultValue="true">
  <class>Alarm</class>
  <attribute name="ID">alarm1</attribute>
</addObject>
<addObject id="alarm2" initializeDefaultValue="false">
  <class>Alarm</class>
  <attribute name="ID">alarm2</attribute>
  <attribute name="severity">Major</attribute>
  <attribute name="acknowledged">>false</attribute>
</addObject >
<addObject id="alarm3" initializeDefaultValue="false">
  <class>Alarm</class>
  <attribute name="ID">alarm3</attribute>
```

```

    <attribute name="severity">Major</attribute>
    <attribute name="acknowledged">true</attribute>
  </addObject >
  <addObject id="alarm4" initializeDefaultValue="false">
    <class>Alarm</class>
    <attribute name="ID">alarm4</attribute>
  </addObject >
  <updateObject id="alarm2">
    <attribute name="acknowledged">true</attribute>
  </updateObject>
</cplData>

```

In this example, the `<classes>` XML element delimits the business model definition. This model can also be defined in a separate file. For details, see *Defining a dynamic class in XML*.

When defining your business model in a data source file, please note that the business class inheritance (defined through the tag `"superClass"`) is resolved at the end of each `<classes>` element. As a consequence, you cannot define within the first `<classes>` element a class that inherits from a class defined within another `<classes>` element later in the file. For example, the following model definition is incorrect:

Incorrect example

```

<cplData xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation = "ilog/cpl/schema/data.xsd">
  <classes>
    <class>
      <name>Domain</name>
      <superClass>NetworkElement</superClass>
    </class>
  </classes>

  <classes>
    <class>
      <name>NetworkElement</name>
      <attribute>
        <name>name</name>
        <javaClass>java.lang.String</javaClass>
      </attribute>
    </class>
  </classes>
</cplData>

```

To get a correct business model definition, you can either define the whole business class hierarchy inside the same `<classes>` element as follows:

Correct example

```

<cplData xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation = "ilog/cpl/schema/data.xsd">

```

```

<classes>
  <class>
    <name>NetworkElement</name>
    <attribute>
      <name>name</name>
      <javaClass>java.lang.String</javaClass>
    </attribute>
  </class>
  <class>
    <name>Domain</name>
    <superClass>NetworkElement</superClass>
  </class>
</classes>
</cplData>

```

or, make sure that the business class needed is already present in the class manager of the current application context. To do so, you load a separate business model XML file directly in the class manager. Please refer to Class manager in the *Application Context and Deployment Descriptor* documentation.

The following table describes the elements that you can use to define actions to be performed on the business model data in XML format. Mandatory attributes appear in boldface. You can also find a description of this format in the XML schema file `data.xsd`, located at **<installDir> /data/ilog/cpl/schema/data.xsd**.

Elements in an XML data file

XML Elements	Attributes	Default	Description
<cplData>	None		Delimits data definition. This element is required. The file containing XML business data should necessarily start and end with this element.
<classes>	None		Delimits the model definition, that is, dynamic class definition. For details, see <i>Defining a dynamic class in XML</i> .
<class>	None		Within an <updateState> element, defines the class of the added object.
<attribute>			Within an <updateState> or <updateObject> element, defines the values of the attributes.
	name		This attribute is mandatory. The name of the attribute as defined in the object class.
	null	false	This attribute is optional. When true, it indicates that the value of the attribute is null.
	javaClass	The attribute Java™	This attribute is optional. It specifies the Java class name of the attribute value. It becomes mandatory if the

XML Elements	Attributes	Default	Description
		class as defined in the <code>IlpClass</code>	attribute value class is an abstract class. See Note 3.
<code><updateState></code>			Within an <code><updateObject></code> element, defines incremental updates to attribute <code>objectState</code> for predefined business objects. See <i>Defining states in XML</i> .
<code><batch></code>			Group data source updates. This element may contain any number of <code><updateState></code> , <code><removeObject></code> , <code><updateObject></code> . Also, <code><batch></code> elements can be nested.
<code><addObject></code>			Adds an object to the data source. This element contains a mandatory <code><class></code> element. It may contain structural elements and <code><attribute></code> elements.
	id		This attribute is mandatory and should be unique.
	<code>idClass</code>	<code>java.lang.String</code>	This attribute is optional. It can be used to specify a Java class name for the <code>id</code> . See Note 2.
	<code>initializeDefaultValue</code>	<code>true</code>	This attribute is optional. When <code>true</code> , it initializes attributes to their default values.
	<code>container</code>		This attribute is optional. It allows you to specify whether the object is a container. An object is considered to be a container if this attribute is set to <code>true</code> , if it contains <code><children></code> , or if another object is declared as its <code><parent></code> .
<code><updateObject></code>			Updates the value of an existing object. It may contain structural elements, <code><attribute></code> and <code><updateState></code> elements.
	id		This attribute is mandatory.
	<code>idClass</code>	<code>java.lang.String</code>	This attribute is optional. It can be used to specify a Java class name for the <code>id</code> . See Note 2.
	<code>container</code>		This attribute is optional. It allows you to specify whether the object is a

XML Elements	Attributes	Default	Description
			container. An object is considered to be a container if this attribute is set to <code>true</code> , if it contains <code><children></code> , or if another object is declared as its <code><parent></code> .
<code><removeObject></code>			Removes an existing object. It should not contain any other data or elements.
	id		This attribute is mandatory.
	<code>idClass</code>	<code>java.lang.String</code>	This attribute is optional. It can be used to specify a Java class name for the <code>id</code> . See Note 2.
	<code>childrenToo</code>	<code>true</code>	This attribute is optional. It specifies whether all the children of the object should be removed from the data source.
<code><children></code>	None		Structural element indicating that an object is a potential container, even if <code><children></code> does not contain any

XML Elements	Attributes	Default	Description
			other subelements. This element can contain <code><child></code> subelements.
<code><link></code>			Structural element that lets you specify the extremities of a link. It contains one <code><to></code> and one <code><from></code> subelement.
	<code>idClass</code>	<code>java.lang.String</code>	This attribute is optional. It can be used to specify a Java class name for the <code>id</code> . See Note 2.
<code><parent></code> See Note 1			Structural element. It contains the <code>id</code> of the parent object.
	<code>idClass</code>	<code>java.lang.String</code>	This attribute is optional. It can be used to specify a Java class name for the <code>id</code> . See Note 2.
<code><child></code> See Note 1			Contains the ID of the child.
	<code>idClass</code>	<code>java.lang.String</code>	This attribute is optional. It can be used to specify a Java class name for the <code>id</code> . See Note 2.
<code><to></code> See Note 1			Contains the <code>id</code> of the <code>to</code> extremity of a link.
	<code>idClass</code>	<code>java.lang.String</code>	This attribute is optional. It can be used to specify a Java class name for the <code>id</code> . See Note 2.
<code><from></code> . See Note 1			Contains the <code>id</code> of the <code>from</code> extremity of a link.
	<code>idClass</code>	<code>java.lang.String</code>	This attribute is optional. It can be used to specify a Java class name for the <code>id</code> . See Note 2.

- Note:**
- Whether the object referred to by the ID in `<parent>`, `<child>`, `<to>`, and `<from>` elements exists or not is not important. When the corresponding object is created, either with the `<addObject>` XML element or through a call to a data source method, the adapter that makes use of the structural information will perform any necessary action to connect the representation objects matching the business objects specified in the structural element.
 - The supported Java classes are those handled by the type converter. See `IlpTypeConverter` and `IlpDefaultTypeConverter` for more information.

3. The `javaClass` attribute of the `<attribute>` element behaves like the `javaClass` attribute of the `<defaultValue>` element. See *Attribute types*.

XML file samples

How to define the business model

The following sample illustrates the use of the XML elements `<cplData>`, `<classes>`, `<class>` and `<attribute>`, which define the business model:

```
<cplData xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation = "ilog/cpl/schema/data.xsd">
<classes>
  <class>
    <name>NetworkElement</name>
    <attribute>
      <name>name</name>
      <javaClass>java.lang.String</javaClass>
    </attribute>
    <attribute>
      <name>site</name>
      <javaClass>java.lang.String</javaClass>
    </attribute>
    <attribute>
      <name>position</name>
      <javaClass>ilog.cpl.graphic.IlpPoint</javaClass>
    </attribute>
  </class>
  <class>
    <name>Domain</name>
    <superClass>NetworkElement</superClass>
  </class>
  <class>
    <name>LinkElement</name>
    <attribute>
      <name>name</name>
      <javaClass>java.lang.String</javaClass>
    </attribute>
  </class>
</classes>
</cplData>
```

How to create a new business object

The following sample illustrates the use of the XML element `<addObject>` to create a new business object of the business class "Domain":

```
<addObject id="Domain1">
```

```

<class>Domain</class>
<attribute name="name">Domain 1</attribute>
<attribute name="site">Gentilly</attribute>
<attribute name="position" javaClass="ilog.cpl.graphic.IlpPoint">
  <x>100</x> <y>100</y>
</attribute>
</addObject>

```

How to create containment relationships between business objects

The following sample shows how to create containment relationships between business objects using the XML element `<parent>`. The example creates a new business object "Server1" which is a child of business object "Domain1".

```

<addObject id="Server1">
  <class>Server</class>
  <parent>Domain1</parent>
  <attribute name="name">S1</attribute>
  <attribute name="site">Montreuil</attribute>
  <attribute name="position" javaClass="ilog.cpl.graphic.IlpPoint">
    <x>100</x> <y>50</y>
  </attribute>
</addObject>

```

How to specify a link relationship

The following sample shows how to specify a link relationship using the XML elements `<link>`, `<to>` and `<from>`. In this example, you create the link "Link1" which connects two objects "Domain1" and "Domain2".

```

<addObject id="Link1">
  <class>LinkElement</class>
  <attribute name="name">InterDomain</attribute>
  <link>
    <from>Domain1</from>
    <to>Domain2</to>
  </link>
</addObject>

```

How to update the values of an object

The following sample shows how to update values of an existing object using the XML element `<updateObject>`. You can update simple attributes, such as "name", and you can also change structural information such as changing the end point of the link:

```

<updateObject id="Link1">
  <attribute name="name">Server1-Domain2</attribute>
  <link>
    <from>Server1</from>
    <to>Domain2</to>
  </link>
</updateObject>

```

```
</link>  
</updateObject>
```

How to remove an object from the data source

The following sample shows how to remove an object from the data source using the XML element `<removeObject>`:

```
<removeObject id="Link1"/>
```

Writing the data source content to XML

You can also write the content of the data source to an XML file like this:

```
datasource.output("network.xml");
```

You can also use a `java.io.Writer`. The following example prints the data source to the default output:

```
Writer writer = new PrintWriter(System.out);  
datasource.output(writer);  
writer.flush();
```

See also *Advanced parsing and writing of a data source* for details on more advanced functionality.

Adding predefined business objects

Almost all predefined business objects can be added to a data source. This is the case for:

- ◆ Alarms (See *Loading an alarm defined in XML.*)
- ◆ Links (See *Loading a link defined in XML.*)
- ◆ Link sets (See *Loading a link set defined in XML.*)
- ◆ Link bundles (See *Loading a link bundle defined in XML.*)
- ◆ Groups (See *Loading a group defined in XML.*)
- ◆ Network elements (See *Loading a network element defined in XML.*)
- ◆ Off-page connectors (See *Loading an off-page connector defined in XML.*)
- ◆ Shelves (See *Loading a shelf defined in XML.*)
- ◆ Cards (See *Loading a card defined in XML.*)
- ◆ Card carriers (See *Loading a card carrier defined in XML.*)
- ◆ LEDs (See *Loading an LED defined in XML.*)
- ◆ Ports (See *Loading a port defined in XML.*)
- ◆ Base transceiver stations (See *Loading a BTS object defined in XML.*)
- ◆ States (See *Defining states in XML.*)

Refer to section *Introducing business objects and data sources* in this documentation for an overview of these objects.

For information on how to add these objects to a data source through XML or through the API, refer to the corresponding sections further in this documentation.

Adding business objects from JavaBeans

The default implementation of the `IlpDataSource` interface provides an API that directly supports JavaBeans™ objects. This API transforms JavaBeans objects into `IlpObject` instances of type `IlpBeansObject`.

If necessary, see *Reminder about JavaBeans design patterns*.

The example below shows how to add JavaBeans objects to a data source. It reuses the `MO` class defined in *Defining the business model from JavaBeans classes*. Objects are created as regular Java™ objects, inserted in a data source, and modified by means of their set methods.

How to add JavaBeans objects to a data source

```
MO mo1 = new MO("mo1");
dataSource.addBean(mo1, "mo1");
mo1.setState(2);
```

Once in the data source, these objects can be displayed in the various JViews TGO graphic components. Bound property updates are dynamically reflected in these components.

The figure below shows an `MO` object displayed in a tree component.



JavaBeans object displayed in a Tree component

Note: The graphical result depends also on the style information associated with the object. Here, the state attribute is mapped to a color. For details, see *Introducing cascading style sheets*.

Adding dynamic business objects

To create a dynamic business object, all you have to do is create an instance of `IlpDefaultObject` for an `IlpClass`, generally an `IlpDefaultClass`.

Here is an example:

How to create a dynamic business object

```
IlpDefaultObject bo = new IlpDefaultObject(alarmClass, "Alarm1");
bo.setAttributeValue(severityAttribute, new Integer(2));
bo.setAttributeValue(ackAttribute, Boolean.FALSE);
```

Note that here the identifier of the object is a string, but it does not have to be. This identifier could be of any class, provided that instances of that class can be read from and converted to string using the type converter.

The interface `IlpObject` also contains convenience methods to retrieve and set attribute values based on the attribute name. These convenience methods can be easily implemented as follows:

```
public Object getAttributeValue (String attribute) {
    IlpAttribute attr = getAttributeGroup().getAttribute(attribute);
    if (attr != null)
        return getAttributeValue(attr);
    return IlpAttributeValueHolder.VALUE_NOT_SET;
}

public void setAttributeValue (String attributeName, Object value) {
    IlpAttribute attr = getAttributeGroup().getAttribute(attribute);
    if (attr != null)
        setAttributeValue(attr, value);
}
```

Adding business objects to the data source

The default data source implementation, `IltDefaultDataSource`, provides methods to add and remove business objects. Once a business object is added into a data source that is connected to a graphic component, it can be automatically displayed by the graphic component.

If you want to add your business objects to the data source without having their graphic representation created, you need to use an `IlpFilter` and apply it to the `IlpAbstractAdapter` in charge. In this way, the objects are filtered out and not included in the graphic representation model. No graphic representation will be created at rendering time. You can, at a later stage, change the filter to make more objects visible as needed.

Note: The filter must be set on the `IlpAbstractAdapter` before the business objects are added to the data source.

Such filters allow the business objects to be managed by the data source and still not be represented graphically.

The following methods are available to add business objects to a data source:

◆ `void`

`addObject(ilog.cpl.model.IlpObject)`. Adds a single object to the data source

◆ `void addObjects(java.util.List)` Adds a collection of objects to the data source.

Whenever possible, large series of `IlpObject` instances should be added as collections instead of one by one. This is typically the case when you initialize the application.

For example, the code sample below:

```
for (...) {
    parent = new IltNetworkElement("ROOT");
    dataSource.addObject(parent);

    for (...) {
        child = new IltNetworkElement("NE");
        dataSource.setParent(child, parent);
        dataSource.addObject(child);
    }
}
```

Could be improved to:

```
List objects = new ArrayList();
for (...) {
    parent = new IltNetworkElement("ROOT");
    objects.add(parent);
    for (...) {
        child = new IltNetworkElement("NE");
        dataSource.setParent(child, parent);
        objects.add(child);
    }
}
dataSource.addObjects(objects);
```

Removing business objects from the data source

When the business objects are no longer needed by the application, they can be removed from the data source using one of the following methods:

- ◆ `IlpObject removeObject(Object idOrIlpObject, boolean childrenToo)`: Removes the required object from the data source. The argument "childrenToo" is used to indicate if the whole object subtree should be removed from the data source at the same time.
- ◆ `List removeObjects(List idsOrIlpObjects, boolean childrenToo)`: Removes a collection of objects from the data source.
- ◆ `void clear()`: Removes all business objects from the data source.

Whenever possible, large series of `IlpObject` instances should be removed as collections instead of removing the objects one by one.

Defining business object relationships

Business object relationships, such as links or containment, are defined in the data source using the following methods:

◆ `void setLink (Object idOrIlpObject, Object fromIdOrIlpObject, Object toIdOrIlpObject)`

This method declares an object as being a link, connecting the `from` business object to the `to` business object. The structural information of the link is defined by the interface `IlpLink`.

◆ `void setParent (Object idOrIlpObject, Object parentIdOrIlpObject)`

This method declares an object as being the parent of another object.

◆ `void setChildren (Object idOrIlpObject, List childrenIdsOrIlpObjects)`

This method declares an object as being the parent of the given list of child objects.

How to define parent-child relationships between business objects

The following example illustrates the use of the methods `setParent` and `setChildren` in a default data source.

```
IlpObject parent = new IltNetworkElement("NE1");
IlpObject child = new IltNetworkElement("NE1_1");
dataSource.setParent(child, parent);
dataSource.addObject(child);
dataSource.addObject(parent);
```

or

```
IlpObject parent = new IltNetworkElement("NE1");
IlpObject child1 = new IltNetworkElement("NE1_1");
IlpObject child2 = new IltNetworkElement("NE1_2");
IlpObject child3 = new IltNetworkElement("NE1_3");
List children = new ArrayList();
children.add(child1);
children.add(child2);
children.add(child3);
dataSource.setChildren(parent, children);
dataSource.addObjects(children);
dataSource.addObject(parent);
```

How to define a link between business objects

The following example illustrates the use of the method `setLink` in a default data source.

```
IlpObject fromEnd = new IltNetworkElement("NE1");
```

```
IlpObject toEnd = new IltNetworkElement("NE2");
IlpObject link = new IltLink("NE1<->NE2");

dataSource.setLink (link, fromEnd, toEnd);
List objects = new ArrayList();
objects.add(fromEnd);
objects.add(toEnd);
dataSource.addObjects(objects);
dataSource.addObject(link);
```

Whenever possible, first add all end-point objects to the data source, then add the corresponding link objects. This avoids internal checks and temporary object storage to properly create and arrange the object hierarchy.

To improve the data source and component performance, it is also recommended to avoid changing relationships after the objects have been added to the data source. For example:

Less efficient:

```
dataSource.addObject(object);
dataSource.setParent(object, parent);
```

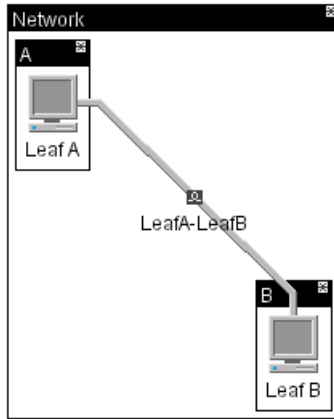
More efficient:

```
dataSource.setParent(object, parent);
dataSource.addObject(object);
```

How to define an intergraph link

JViews TGO is able to display links connecting objects in different hierarchies. These links are known as *intergraph links*. To have an intergraph link properly displayed, you need to take care of how the link object is created in the data source, specially at which hierarchy level the link object is added.

To illustrate an intergraph link use case, let's imagine the following object hierarchy:



Object hierarchy

Suppose you wanted to create a link connecting Leaf A and Leaf B. You would have to add it to the data source in the following way:

```

ArrayList objects = new ArrayList();
IltNetworkElement net = new IltNetworkElement("Network");
net.setName("Network");

IltNetworkElement branchA = new IltNetworkElement("BranchA");
branchA.setName("A");
IltNetworkElement branchB = new IltNetworkElement("BranchB");
branchB.setName("B");

objects.add(net);
objects.add(branchA);
objects.add(branchB);

dataSource.setParent(branchA, net);
dataSource.setParent(branchB, net);

IltNetworkElement leafA = new IltNetworkElement("LeafA");
leafA.setType(IltNetworkElement.Type.NMW);
leafA.setPosition(new IlpPoint(100,100));
leafA.setName("Leaf A");
IltNetworkElement leafB = new IltNetworkElement("LeafB");
leafB.setType(IltNetworkElement.Type.NMW);
leafB.setPosition(new IlpPoint(250,250));
leafB.setName("Leaf B");

dataSource.setParent(leafA, branchA);
dataSource.setParent(leafB, branchB);
objects.add(leafA);
objects.add(leafB);

```

```
// Creating the intergraph link
IltLink link = new IltLink("LeafA-LeafB");
link.setName("LeafA-LeafB");
dataSource.setLink(link, leafA, leafB);
// Setting the link hierarchy level
dataSource.setParent(link, net);
objects.add(link);
dataSource.addObjects(objects);
```

The intergraph link should be placed at the highest hierarchy level common to both end points. This is illustrated in the example above where the link is set as a child of the "network" business object.

How to retrieve business object structural information

The current implementation of `IltDefaultDataSource` handles structural information, such as child-parent relationships described earlier, pertaining to `IlpObject` instances independently of the objects themselves. This implementation makes it possible for you to load objects on demand. See *How to implement load-on-demand in a data source* for more information.

The structural information is defined in JViews TGO through the following interfaces:

- ◆ `IlpChild`—This interface defines the method `Object getParent(IlpObject object)`, which returns the parent object identifier from the given business object.
- ◆ `IlpContainer`—This interface defines the method `Collection getChildren(IlpObject object)`, which returns the list of child identifiers from the given business object.
- ◆ `IlpLink`—This interface defines the structural information needed to indicate that a business object is a link between two other business objects. This interface declares the methods `Object getFrom(IlpObject object)` and `Object getTo(IlpObject object)`.
- ◆ `IlpLinkExtremity`—This interface defines the structural information to indicate that a business object is the end of links. This interface declares method `Collection getLinks(IlpObject object)`.

The default data source implementation provides the following methods to retrieve each one of these interfaces:

- ◆ `IlpChild getChildInterface (Object childIdOrIlpObject)`
- ◆ `IlpContainer getContainerInterface (Object containerIdOrIlpObject)`
- ◆ `IlpLink getLinkInterface (Object linkIdOrIlpObject)`
- ◆ `IlpLinkExtremity getLinkExtremityInterface (Object linkIdOrIlpObject)`

You can use the following convenience methods to retrieve the structural information. These methods return the structural information based on the fact that the queried business objects are all present in the data source. If this is not the case, you should retrieve the structural information using the interfaces instead.

- ◆ `IlpObject getParent (IlpObject object)`: Returns the parent of the given object.

- ◆ `Collection getChildren (IlpObject object)`: Returns the collection of child objects for the given object.
- ◆ `Collection getLinks (IlpObject node)`: Returns the collection of links that have the given node as an end point.
- ◆ `IlpObject getFrom (IlpObject link)`: Returns the from end point of the given link object.
- ◆ `IlpObject getTo (IlpObject link)`: Returns the to end point of the given link object.

Grouping changes in batches

You can . In this way, changes are handled more efficiently and provide improved performance. You can batch additions and removals of objects, as well as changes to the business objects themselves, such as changes in the containment structure, link extremities or attribute values.

How to batch changes

```
IlpAbstractDataSource datasource = ...
...
datasource.startBatch();
// add/remove objects
// update attribute values, change containment, change link extremities
...
datasource.endBatch();
```

Data source changes that are grouped in a batch are delayed until the call to `endBatch`.

Batches can be nested. In this case, changes are delayed until the most enclosing call to `endBatch`).

How to nest batches

```
datasource.startBatch();
  datasource.startBatch();
  // first batch
  ...
  datasource.endBatch();
  datasource.startBatch();
  // second batch
  ...
  datasource.endBatch();
datasource.endBatch();
```

How to batch changes in XML

Data source changes in XML streams can be grouped in batches using the element `<batch>` as illustrated in the following sample:

```
<cplData>
  <batch>
    <addObject>
      <!-- ... -->
    </addObject>
    <updateObject>
      <!-- ... -->
```

```
    </updateObject>  
    <removeObject>  
      <!-- ... -->  
    </removeObject>  
  </batch>  
</cplData>
```

Batches can also be nested in data source XML streams by nesting `<batch>` elements. In this case, changes are delayed until the most enclosing `</batch>`.

Advanced parsing and writing of a data source

In addition to the basic `IlpDefaultDataSource` API for parsing a data source and writing the data source content to an XML file, JViews TGO provides the classes `IlpDataSourceLoader` and `IlpDataSourceOutput` for a more advanced use. Both these classes are located in the package `ilog.cpl.storage`.

Parsing an XML File

The class `IlpDataSourceLoader` has the following functionality:

- ◆ Reads the data source content directly from a SAX `InputSource`.
- ◆ Gives access to the SAX `XMLReader`. You can modify the behavior of the XML reader before the parsing takes place (for example, to disable the validation of the XML schema).

How to modify the behavior of the XML reader

```
IlpDataSourceLoader loader = new IlpDataSourceLoader(inputSource,
                                                    dataSource);
XMLReader reader = loader.getXMLReader();
reader.setFeature("http://xml.org/sax/features/validation", false);
loader.parse();
```

- ◆ Uses an identifier factory. An identifier factory lets you change the identifiers while the XML file is being loaded.
- ◆ Loads new root objects as children of an object present in the data source.
- ◆ Loads objects that match filter criteria.

The example below loads a template file into the data source. It shows how to create an identifier factory to modify parent identifiers of root objects. The complete sample code is located in the following directory:

```
<installdir>/tutorials/browser
```

How to load a file into a data source

```
final Object parentID = expandedObject.getIdentifier();

// Create an identifier factory that prepends the parent ID to
// all identifiers read from the template.
IlpIdentifierFactory idFactory = new IlpIdentifierFactory(){
    public Object getIdentifier (Object previousIdentifier){
        return parentID.toString() + "/" + previousIdentifier.toString();
    }
};
// Load the template into the datasource
```

```
// Load the template objects under the parent node, and transform
// their IDs so they are unique
IlpDataSourceLoader loader = new IlpDataSourceLoader(templateFileName,
                                                    mainDataSource);
loader.setIdentifierFactory(idFactory);
loader.setParentIdOfRootObjects(parentID);
loader.parse();
```

Writing to an XML file

The class `IlpDataSourceOutput` has the following functionality:

- ◆ Writes the contents of the data source directly to a SAX `ContentHandler`. This content handler can be used to feed a DOM tree for example.
- ◆ Enables or disables enhanced printing. By default, pretty printing is enabled but you can deactivate it to improve performance.
- ◆ Uses an identifier factory. An identifier factory lets you change the identifiers while the XML file is being written. Its use is the same as for parsing.
- ◆ Writes a hierarchy of objects by calling the method `outputHierarchy(IlpObject)`. The `IlpObject` parameter and all its children (which are defined by the container interface) are written to the file.
- ◆ Writes a single object by calling `outputObject(IlpObject)`.
- ◆ Writes only the objects that match the specified filter criteria.

Implementing a new data source

This section is based on the sample located in `<installdir>/samples/datasource/explorer`.

Implementing the data source

`IltDefaultDataSource` includes support for any `IlpClass` instances and handles tables mapping `IlpObject` instances with identifiers. Deriving `FileDataSource` from that class allows you to benefit from this functionality.

How to implement a data source

Extend the `IltDefaultDataSource` class as follows:

```
public class FileDataSource extends IltDefaultDataSource {
```

Write the constructor on the following model:

```
public FileDataSource() {
    super();
    // Retrieve the File Java class as an IlpClass
    IlpMutableClassManager classManager = getContext().getClassManager();
    fileClass = (IlpMutableClass)classManager.getClass(File.class.getName());
    // Create a subclass of the java.io.File IlpClass for root objects,
    // so as to apply different styles to them.
    classOfRoots = new IlpDefaultClass("DirectoryRoot",
                                     fileClass,
                                     Collections.EMPTY_LIST);

    fileClass.addSubClass(classOfRoots);
    classManager.addClass(classOfRoots);
    // loads the roots
    File[] rootdirs = File.listRoots();
    addFiles(rootdirs, classOfRoots);
}
```

This data source is meant to read in objects of type `File` which, in Java™, are instances of the class `java.io.File`. In this sample code, `java.io.File` objects are defined by the `IlpClass` automatically generated by the class manager. Here, the class `java.io.File` is considered as a `JavaBeans™` class (which is acceptable, as most of its methods comply with `JavaBeans` conventions). As an alternative, you could create a dedicated `IlpClass` implementation and the corresponding `IlpObject` implementation. See *Defining the business model with dynamic classes* and *Adding dynamic business objects*.

There is a known limitation in Microsoft® Windows® environments: the `getName` method of the class `java.io.File` does not display labels correctly. This is why it is necessary here to subclass the `IlpClass` defining `java.io.File` to represent the roots of the file system.

Once the required `IlpClass` objects are known, the data source is initialized with the root objects returned by the `listRoots` method of the class `java.io.File`.

How to insert objects into a data source by reading from an array of files

The `addFiles` method, detailed below, iterates over an array of `Files`, creates the corresponding `IlpObject` instances if they are not yet present in the data source, and inserts these objects into the data source.

```
protected synchronized void addFiles(File[] files, IlpClass ilpClass) {
    List filesAsIlpObjects = new ArrayList(files.length);
    for (int i = 0; i < files.length; i++) {
        // use the file itself as the object identifier
        Object id = files[i];
        if (getObject(id) == null) {
            IlpBeansObject ilpObject = new IlpBeansObject(files[i],
                ilpClass, files[i]);
            filesAsIlpObjects.add(ilpObject);
        }
    }
    if (filesAsIlpObjects.size() != 0)
        addObjectList(filesAsIlpObjects);
}
```

How to implement load-on-demand in a data source

The current implementation of `IlpDefaultDataSource` handles structural information (that is, child-parent relationships) pertaining to `IlpObject` instances independently of the objects themselves.

This implementation makes it possible for you to load objects on demand in two different ways:

◆ with the `getObject(Object id)` method

or

◆ with the `getContainerInterface` method

This method returns an implementation of the `IlpContainer` interface. This interface contains a `getChildren(log.cpl.model.IlpObject)` method that returns the identifiers of the children of the `IlpObject`.

In our example, we have implemented load-on-demand by redefining the `getContainerInterface` of the data source (see *Implementing the `getContainerInterface` method*). Since we do not use the access to structural information provided by the default data source in order not to duplicate it from the `File` class where it is already present, we also provide a specific implementation of the `getChildInterface` method (see *Implementing the `getChildInterface` method and the `IlpChild` interface*).

Note: You could also create `IlpObject` instances dynamically by reimplementing the method `getObject(Object id)`.

Implementing the `getContainerInterface` method

The method `getContainerInterface` returns an `IlpContainer`. If the object is a container, the method should return a non-null value, whether this container has children or not, as shown in the following example.

How to get a container

```
public synchronized IlpContainer getContainerInterface(Object idOrIlpObject)
{
    IlpObject object = idOrIlpObject instanceof IlpObject ?
    (IlpObject)idOrIlpObject : getObject(idOrIlpObject);
    if (object != null && fileClass.isAssignableFrom(object.getIlpClass()))
    {
        File file = (File)((IlpBeansObject)object).getBean();
        if (file.isDirectory())
            return DIRECTORY_LOADER;
        else
            return null;
    }
    return super.getContainerInterface(idOrIlpObject);
}
```

The `getContainerInterface` method takes an `IlpObject` or an identifier as parameter. It retrieves the `IlpObject` instance from its parameter and checks whether the returned object is an instance of the `IlpClass` corresponding to the `File` class and whether `File` is a directory. If it is not a directory, it returns `null` or the default implementation of the data source, if the object is not of the `File` `IlpClass`. If it is a directory, the method returns a specific implementation of the `IlpContainer` interface (see *Implementing the `IlpContainer` interface*).

Note that in this example, the `IlpObject` is downcast to an instance of `IlpBeansObject` and that the `Bean` it contains is in turn downcast to a `File` instance. We could have used the `directory` attribute of the `IlpClass` directly as follows:

How to use the `directory` attribute of an `IlpClass` directly

```
Boolean isDirectory =
    Boolean)object.getAttributeValue(fileClass.getAttribute("directory"));
if (isDirectory.booleanValue())
    return DIRECTORY_LOADER;
else
    return null;
```

Implementing the `IlpContainer` interface

The default data source provides its own implementations of all the structural interfaces (such as `IlpChild` or `IlpContainer`). These implementations ensure that the information they contain is coherent. For example, if `A` is the child of `B`, `B` is the parent of `A`.

These implementations do not support load-on-demand. To support load on demand, you have to redefine the `IlpContainer` interface.

It is quite easy to retrieve the children of a `File` object and load them. Following is an implementation of `IlpContainer` for the class `java.io.File`. It uses a static instance of the class since the entire context of the `getChildren` method is contained in its parameter, which is the `IlpObject` instance itself.

How to retrieve child objects

```
IlpContainer DIRECTORY_LOADER = new IlpContainer(){
    public Collection getChildren(IlpObject object) {
        // the file is supposed to be a directory
        File directory = (File)((IlpBeansObject)object).getBean();
        File[] files = directory.listFiles();
        List filesIds = new ArrayList(files.length);
        for (int i = 0; i < files.length; i++) {
            // 1st get all the identifiers
            filesIds.add(files[i]);
        }

        // then add the files
        addFiles(files, fileClass);
        return filesIds;
    }
};
```

Note that to avoid recreating the same `IlpObject` instance twice, no record is kept as to whether a directory has already been loaded. The test in the `addFiles` method (described in *Implementing the data source*) is used to check whether an `IlpObject` instance exists before creating it and adding it to the data source.

Implementing the `getChildInterface` method and the `IlpChild` interface

Since you do not use the access to structural information provided by the default data source, you have to provide specific implementations of the `getChildInterface` method and of the `IlpChild` interface. This is quite easy, since the `java.io.File` class has a `getParentFile` method.

How to provide specific implementations of `getChildInterface` and `IlpChild`

```
public synchronized IlpChild getChildInterface(Object idOrIlpObject) {
    IlpObject object = idOrIlpObject instanceof IlpObject ?
    (IlpObject)idOrIlpObject : getObject(idOrIlpObject);
    if (object != null && fileClass.isAssignableFrom(object.getIlpClass()))
    {
        File file = (File)((IlpBeansObject)object).getBean();
        File parentFile = file.getParentFile();
    }
}
```

```
        if (parentFile != null) {
            return FILE_PARENT;
        } else {
            return null;
        }
    }
    return super.getChildInterface(idOrIlpObject);
}
```

The method gets the `IlpObject` and checks whether that object is a file. It then checks whether it has a parent file and returns the specific implementation of the `IlpChild` interface, described below.

Like the other structural interfaces, the `IlpChild` interface returns an identifier. Here, since the identifiers of the objects are the files themselves, it returns the parent file.

How to get the parent file

```
IlpChild FILE_PARENT = new IlpChild() {
    public Object getParent(IlpObject object) {
        File file = (File)((IlpBeansObject)object).getBean();
        File parentFile = file.getParentFile();
        return parentFile;
    }
}
```

Network elements

Explains how to use IBM® ILOG® JViews TGO predefined business objects of type network element in your application.

In this section

Network element class

Describes the attributes of the `IltNetworkElement` class.

Loading a network element defined in XML

Shows how to load a network element from an XML file into a data source.

Creating a network element with the API

Shows how to create a network element using the JViews TGO API and add it to a data source.

Representation of network elements in a network

Describes the different aspects of the graphical representation of a network element in a network.

Representation of network elements in a table and in a tree

Shows how some network elements are represented in a table and in a tree.

Network element class

Network elements include any kind of network managed objects, such as data-communications equipment (a switch, a multiplexer, a cross-connect, for example), outside plant equipment, and peripheral equipment (terminal or printer).

Network elements are predefined business objects of the class `IltNetworkElement` that you can directly insert in a JViews TGO data source and represent graphically in any of the graphic components connected to the data source. For a general introduction to predefined business classes, see *Introducing business objects and data sources*.

The `IltNetworkElement` class defines the following attributes:

- ◆ **Type**—Indicates the category of the network element, which determines the way it will be graphically represented.
 - **Name:** `type`
 - **Value class:** `IltNetworkElement.Type`
 - **Attribute:** `IltNetworkElement.TypeAttribute`
- ◆ **Function**—Indicates the function of the network element according to its category, for example access or switch item of equipment.
 - **Name:** `function`
 - **Value class:** `IltNetworkElement.Function`
 - **Attribute:** `IltNetworkElement.FunctionAttribute`
- ◆ **Family**—Indicates the family of the network element according to its function. Usually, the family represents the equipment capacity.
 - **Name:** `family`
 - **Value class:** `IltNetworkElement.Family`
 - **Attribute:** `IltNetworkElement.FamilyAttribute`
- ◆ **Partial**—Indicates that the network element represents only part of the real-world network element.
 - **Name:** `partial`
 - **Value class:** `java.lang.Boolean`
 - **Attribute:** `IltNetworkElement.PartialAttribute`
- ◆ **Shortcut**—Indicates that the network element is only a reference to an existing network element.
 - **Name:** `shortcut`
 - **Value class:** `ilog.tgo.model.attribute.IltShortcutAttributeType`

- **Attribute:** `IltNetworkElement.ShortcutAttribute`

You can retrieve the class `IltNetworkElement` using its `GetIlpClass()` method. You can handle its instances as simple `IlpObject` instances and set and get its attributes with the generic methods `getAttributeValue` and `setAttributeValue`.

In addition, the class `IltNetworkElement` provides convenience methods, such as `getFunction` and `setFunction`, which you can use directly to access each individual predefined attribute of the class.

Loading a network element defined in XML

For detailed information about data sources, see *Data sources*.

All you have to do to load a network element written in XML into a data source, is create a data source using the data source default implementation defined by `IltDefaultDataSource` and pass the XML file to the `parse` method of the data source, as follows:

```
dataSource = new IltDefaultDataSource();
dataSource.parse("NetworkElementXMLFile.xml");
```

How to define a network element in XML

Below is an example of a network element defined in XML format. For details about the XML elements used in this example, see *Elements in an XML data file*.

In this example, the object state associated with the first network element (NE1) has a default value. The second object (NE2) explicitly defines the primary states of its associated OSI state. For details about states, see *States*.

```
<cplData>
  <addObject id="NE1">
    <class>ilog.tgo.model.IltNetworkElement</class>
    <attribute name="name">NE1</attribute>
    <attribute name="type">NE</attribute>
    <attribute name="function">Transport</attribute>
    <attribute name="family">OC96</attribute>
    <attribute name="position" javaClass="ilog.cpl.graphic.IlpPoint">
      <x>10.0</x>
      <y>50.9</y>
    </attribute>
    <attribute name="objectState"
      javaClass="ilog.tgo.model.IltOSIObjectState"/>
  </addObject>
  <addObject id="NE2">
    <class>ilog.tgo.model.IltNetworkElement</class>
    <attribute name="name">NE2</attribute>
    <attribute name="type">Terminal</attribute>
    <attribute name="position" javaClass="ilog.cpl.graphic.IlpPoint">
      <x>60.0</x>
      <y>50.9</y>
    </attribute>
    <attribute name="objectState"
      javaClass="ilog.tgo.model.IltOSIObjectState">
      <state>
        <administrative>Unlocked</administrative>
        <operational>Enabled</operational>
        <usage>Active</usage>
      </state>
    </attribute>
```

```
</addObject>  
</cplData>
```

The following figure shows the two network elements displayed in a network component:



Network elements displayed in a network component

Creating a network element with the API

All you have to do is create a new network element using the class `IltNetworkElement` and add it to a data source.

How to create a network element through the API

```
IltNetworkElement ne = new IltNetworkElement("NE",
        IltNetworkElement.Type.Digiphone, new IltOSIObjectState());
ne.setAttributeValue(IltObject.PositionAttribute, new IlpPoint(100,100));
IlpDataSource dataSource = new IltDefaultDataSource();
dataSource.addObject(ne);
```


Representation of network elements in a network

Describes the different aspects of the graphical representation of a network element in a network.

In this section

Network element types

Shows the graphical representations used for the different types of network element.

Network element functions

Shows the graphical representations used for the different network element functions.

Network element families

Shows the graphical representations used for the different network element families.

Partial network elements

Explains how to create a partial network element and shows its graphic representation.

Shortcut network elements

Explains how to create a shortcut network element and shows its graphic representation.

Network element sizes

Shows which type of information is represented depending on the size of the network element.

Network element types

The network element type defines how a given network element will be displayed. The network element type is specified by setting the value of the attribute `type` in the business object. This attribute can be set programatically using `IltNetworkElement.TypeAttribute` or through XML.

How to set the network element type using the API

```
IltNetworkElement ne = new IltNetworkElement("NE1");
ne.setType (IltNetworkElement.Type.NMW);
```

or

```
IlpObject ne = ...;
ne.setAttributeValue(IltNetworkElement.TypeAttribute,
IltNetworkElement.Type.NMW);
```

How to set the network element type using XML

```
<addObject id="NE1">
  <class>ilog.tgo.model.IltNetworkElement</class>
  <attribute name="name">NE1</attribute>
  <attribute name="type">NMW</attribute>
</addObject>
```

Depending on the nature of the application, a network element can be represented by a bitmap image, a symbol, or a shape.

Pictorial representation
















In its pictorial representation, the network element base is a bitmap drawing, which is composed of individual graphic objects. This drawing is meant to be realistic, as you can see in the following figure:


















Pictorial representations of a terminal

Several predefined bases are available for shelf-based equipment, terminals, and mobile phone access network elements. JViews TGO includes a number of predefined network element images. The following tables show these drawings and the corresponding network element *type* name.


















IP network elements




Network Element Type	Representation
IP_ComputerFlat	
IP_ComputerTower	
IP_Database	
IP_Desktop	
IP_Equipment	
IP_Firewall	
IP_Firewall2	
IP_InkjetPrinter	
IP_Laptop	
IP_Laptop2	
IP_LaserPrinter	
IP_Mainframe	
IP_Mainframe2	
IP_Mainframe3	
IP_Modem	

Network Element Type	Representation
IP_Modem2	
IP_Network	
IP_Network2	
IP_Network3	
IP_PDA	
IP_Printer	
IP_Printer2	
IP_Router	
IP_Router2	
IP_Router3	
IP_SatelliteAntenna	
IP_Server	
IP_Server2	
IP_Terminal	
IP_Terminal2	















Office network elements



--	--

Network Element Type	Representation
Office_ComputerFlat	
Office_ComputerTower	
Office_Desktop	
Office_Fax	
Office_Fax2	
Office_InkjetPrinter	
Office_Laptop	
Office_Laptop2	
Office_LaserPrinter	
Office_Modem	
Office_Modem2	
Office_PDA	
Office_Phone	
Office_Phone2	
Office_Printer	
Office_Printer2	
Office_Server	
Office_Server2	











Network Element Type	Representation
	
Office_Terminal	
Office_Terminal2	



SAN network elements

Network Element Type	Representation
SAN_CartridgeSystem	
SAN_Database	
SAN_Disk	
SAN_DoubleDatabase	
SAN_FiberChannelSwitch	
SAN_FiberChannelSwitch2	
SAN_Modem	
SAN_Modem2	
SAN_Router	
SAN_Router2	
SAN_Router3	
SAN_ServerExternalDisk	
SAN_ServerInternalDisk	
SAN_SingleDatabase	
SAN_TapeDrive	











Network Element Type	Representation
	
SAN_TripleDatabase	



Telecom network elements

Network Element Type	Representation
Telecom_Database	
Telecom_Lighthouse	
Telecom_Mainframe	
Telecom_Mainframe2	
Telecom_Mainframe3	
Telecom_MD	
Telecom_MD2	
Telecom_NMW	
Telecom_Server	
Telecom_Server2	
Telecom_Terminal	

Network Element Type	Representation
	
Telecom_Terminal2	



Wireless network elements

Network Element Type	Representation
Wireless_Antenna	
Wireless_Antenna2	
Wireless_BSC	
Wireless_BSC2	
Wireless_BTS	
Wireless_BTS2	
Wireless_HLR	
Wireless_MobilePhone	
Wireless_MSC	
Wireless_MSC2	

Network Element Type	Representation
Wireless_Satellite	
Wireless_SatelliteAntenna	

JViews TGO also provides a pictorial representation for clusters. A cluster network element is an abstraction of a network element that is made up of two or more distinct subcomponents such as other network elements that can be (and often are) managed as standalone telecom objects. Clusters can be *co-located* or *distributed*, depending on how their subcomponents are organized (either within a managed area or spread across different areas). As with the new partial network elements concept, it is possible to represent wholly-owned clusters (that is, all the subcomponents are managed by the user), or partially-owned clusters (only some of the subcomponents are managed by the user).

Cluster network elements

Network Element Type	Representation
Cluster_Colocated	
Cluster_Distributed	

You can create custom network element types by providing a bitmap image or a vector drawing. This process is detailed in Customizing network element types in the *Styling* documentation.

Symbolic representation








In the symbolic representation, network elements can be used for components, nodes or clusters. These objects are represented differently depending on which aspect of the OSS application you consider, for example, the data plane of physical managed telecom objects, the control and management planes or the logical managed telecom objects. The default type corresponding to the default symbolic network element representation is simply called NE (for Network Element). The following figure illustrates an NE type network element: here, an add-drop multiplexer with a capacity of OC192.








Symbolic representation of NE type network element

Symbolic information for NE type network elements consists of an icon representing the *function* of the equipment corresponding to the network element and a string representing the *family* of the equipment.

Symbolic representations of nodes, components and clusters

Network Element Type	Representation	Description
Node		
NE		A network element is an abstraction of a physical managed telecom object unit on the data plane, such as a router or a computer.
Control_Element		A control element is an abstraction of a physical managed telecom object unit on the control plane.
Management_Element		A management element is an abstraction of a physical managed telecom object unit on the management plane.
NE_Logical		A logical network element is an abstraction of a non-physical managed telecom object such as a service or a software component.
Component		
NE_Component		An NE component is an abstraction of a physical managed telecom object that is part of a network element, such as a network card attached to a router.
Control_Component		A control component is an abstraction of a physical managed telecom object that is part of a network element, on the control plane.
Management_Component		A management component is an abstraction of a physical managed telecom object that is part of a network element, on the management plane.














Network Element Type	Representation	Description
NEComponent_Logical		A logical NE component is an abstraction of a non-physical managed telecom object that is part of a logical network element, such as a software element or a logical processor.
Cluster		
NE_Cluster		An NE cluster is an abstraction of a physical managed telecom object that is made up of two or more distinct subcomponents such as other network elements that can be (and often are) managed as standalone telecom objects.
Control_Cluster		A control cluster is an abstraction of a physical managed telecom object that is made up of two or more distinct subcomponents, on the control plane.
Management_Cluster		A management cluster is an abstraction of a physical managed telecom object that is made up of two or more distinct subcomponents, on the management plane.
NECluster_Logical		A logical NE cluster is an abstraction of a non-physical managed telecom object made up of two or more subcomponents such as other services or software components that can be (and often are) managed as standalone telecom objects














Shape representation









In this representation, the network element base is depicted by a geometric shape that symbolizes the network element type (or function class). The center of the base can contain an icon that further defines the representation of the network element function. Several predefined shapes are provided as types of the network element.

JViews TGO includes a limited number of network element shapes that can be used to build iconic network elements. The following table gives the available shapes and the corresponding network element *type* name.

Network element shapes

Network Element Type	Shape
Shape_CellShape	
Shape_Circle	
Shape_CircleSmall	
Shape_Diamond	
Shape_Hexagon	
Shape_HexagonFlat	
Shape_HexagonSmall	
Shape_Octagon	
Shape_Oval	
Shape_OvalSmall	
Shape_Pentagon	
Shape_PentagonBottom	
Shape_PentagonLeft	

Network Element Type	Shape
Shape_PentagonRight	
Shape_PentagonTop	
Shape_Rectangle	
Shape_RectangleSmall	
Shape_RoundSquare	
Shape_Square	
Shape_Transceiver	
Shape_TransceiverBottom	
Shape_TransceiverLeft	
Shape_TransceiverRight	
Shape_TransceiverTop	
Shape_Trapezoid	
Shape_TrapezoidBottom	
Shape_TrapezoidLeft	

Network Element Type	Shape
	
Shape_TrapezoidRight	
Shape_TrapezoidTop	
Shape_Triangle	
Shape_TriangleBottom	
Shape_TriangleLeft	
Shape_TriangleRight	
Shape_TriangleTop	

The API that enables developers to include new network element shapes corresponding to new types of network element is detailed in Customizing network element types in the *Styling* documentation.

Network element functions































JViews TGO includes a certain number of icons representing functions. The design of the icons included in the library stem from recommendations of Standards organizations specialized in this domain (ITU/ANSI).

Note: ANSI T1.232-1996: Operations, Administration, Maintenance and Provisioning (OAM&P)
- G Interface Specification for Use with Telecommunications Management Network (TMN).

The following table illustrates the icons that are available in the library.

Equipment function icon table

Equipment Function	Icon	Function Name
Access Equipment		Access
Edge Cell Regenerator		EdgeCellRegenerator
Edge Cross Connect		EdgeCrossConnect
Edge End Office		EdgeEndOffice
Edge Gateway		EdgeGateway
Edge Hub		EdgeHub
Edge Line Terminating Equipment		EdgeLineTerminatingEquipment
Edge Mediation Device		EdgeMediationDevice
Edge Mux		EdgeMux
Edge Non-technology Specific Mediation Device		EdgeNonTechnologySpecificMediationDevice
Edge Ratio Cell Site Equipment		EdgeRatioCellSiteEquipment
Edge Wireless Edge		EdgeWirelessEdge
IP Equipment		IP
IP Access		IPAccess
IP Alarm Collector		IPAlarmCollector
IP Bridge		IPBridge
IP Hub		IPHub
IP LAN Regenerator		IPLANRegenerator
IP Line Terminating Equipment		IPLineTerminatingEquipment
IP Mediation Device		IPMediationDevice
IP MUX		IPMUX
IP Router		IPRouter
IP Signaling Gateway		IPSignalingGateway
IP STP		IPSTP
IP Switch		IPSwitch
IP Switch Router		IPSwitchRouter
IP Traffic Gateway		IPTrafficGateway
Multi-layer Equipment		MultiLayer

Equipment Function	Icon	Function Name
Multi-layer Access		MultiLayerAccess
Switch Equipment		Switch
ATM/Frame Relay Mediation Device		SwitchingMediationDevice2
Circuit Switching Mediation Device		SwitchingMediationDevice1
Switch Cross Connect		SwitchCrossConnect
Switching ATM		SwitchingATM
Switching Database		SwitchingDatabase
Switching DMS		SwitchingDMS
Switching End Office1		SwitchingEndOffice1
Switching End Office2		SwitchingEndOffice2
Switching Hub1		SwitchingHub1
Switching Hub2		SwitchingHub2
Switching Line Terminating Equipment1		SwitchingLineTerminatingEquipment1
Switching Line Terminating Equipment2		SwitchingLineTerminatingEquipment2
Switching MUX1		SwitchingMUX1
Switching MUX2		SwitchingMUX2
Switching STP1		SwitchingSTP1
Switching STP2		SwitchingSTP2
Switching Toll Gateway1		SwitchingTollGateway1
Switching Toll Gateway2		SwitchingTollGateway2
Switching Toll Tandem1		SwitchingTollTandem1
Switching Toll Tandem2		SwitchingTollTandem2
Transport Equipment		Transport
Transport Access1		TransportAccess1
Transport Access2		TransportAccess2
Transport Add-drop Mux1		TransportAddDropMux1
Transport Add-drop Mux2		TransportAddDropMux2
Transport Amplifier		TransportAmplifier
Transport Circulator		TransportCirculator
Transport Combiner		TransportCombiner

Equipment Function	Icon	Function Name
Transport Cross Connect	◆	TransportCrossConnect
Transport Cross Connect1	◆	TransportCrossConnect1
Transport Cross Connect2	▲	TransportCrossConnect2
Transport Digital Video	◆	TransportDigitalVideo
Transport Dispersion Component Module	▲	TransportDispersionComponentModule
Transport DWDM Optical	▲	TransportDWDMOptical
Transport Fixed Attenuator	▲	TransportFixedAttenuator
Transport Hub1	◆	TransportHub1
Transport Hub2	▲	TransportHub2
Transport Interleave Filter	▲	TransportInterleaveFilter
Transport Line Terminating Equipment1	▲	TransportLineTerminatingEquipment1
Transport Line Terminating Equipment2	◆	TransportLineTerminatingEquipment2
Transport Mediation Device 1	◆	TransportMediationDevice1
Transport Mediation Device 2	▲	TransportMediationDevice2
Transport Optical Switching Module	◆	TransportOpticalSwitchingModule
Transport Regenerator2	◆	TransportRegenerator2
Transport SONET SDH	◆	TransportSONET_SDH
Transport Traffic Gateway1	◆	TransportTrafficGateway1
Transport Traffic Gateway2	▲	TransportTrafficGateway2
Transport Variable Attenuator	▲	TransportVariableAttenuator
Transport Wave Length Translator	◆	TransportWaveLengthTranslator
Other Equipment	□	Other
Unknown Product	?	Unknown

New functions can be added, as detailed in Customizing network element types in the *Styling* documentation.

Network element families

For a given function, several families of network elements can exist. Usually the family represents the capacity of the network element. Values of the network element families are listed in the following table along with their corresponding numbers as shown on the symbolic network element.

Network element families

Network Element Family	Label on symbolic NE
OC1	1
OC3	3
OC9	9
OC12	12
OC18	18
OC24	24
OC36	36
OC48	48
OC96	96
OC192	192
STM1	S1
STM3	S3
STM4	S4
STM6	S6
STM8	S8
STM12	S12
STM16	S16
STM32	S32
STM64	S64

Partial network elements

A *partial network element* is an abstraction which denotes a network element that is only part of the real-world network element. Partial network elements can be used in several situations, for example:

- ◆ To represent distributed clusters where parts of a cluster need to be divided across different subnetworks.
- ◆ To allow one network element to be used by different service providers. In this case, the network element needs to be divided in several parts. Each part is represented as a partial network element and its state reflects only the elements that are interesting for the service provider that is using it.

A network element can be defined as partial by setting the value of the attribute `partial` in the business object. This attribute can be set programmatically using `PartialAttribute`.

How to create a partial network element

The following example illustrates how a network element is created using the attribute `partial` via XML

```
<addObject id="NE1">
  <class>ilog.tgo.model.IltNetworkElement</class>
  <attribute name="name">NE1</attribute>
  <attribute name="partial">true</attribute>
  <attribute name="type">NE</attribute>
</addObject>
```

Partial network elements are graphically represented by an icon located at the bottom left of the network element base.



Partial network elements expanded and collapsed

Shortcut network elements

A *shortcut* network element is an abstraction denoting an object that is only a reference to an existing network element. `IltNetworkElement` provides a new attribute, `ShortcutAttribute`, whose value can be:

- ◆ `IltShortcutAttributeType.STANDARD`: The network element is a regular shortcut.
- ◆ `IltShortcutAttributeType.DANGLING`: The network element is a shortcut to an object that is no longer available.
- ◆ `null`: The network element is not a shortcut

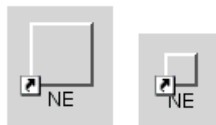
A network element can be defined as shortcut by setting the value of attribute `shortcut` in the business object. This attribute can be set programmatically using `IltNetworkElement.ShortcutAttribute`.

How to create a shortcut

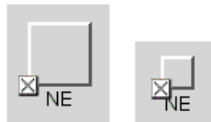
The following example illustrates how a network element is created using the attribute `shortcut` via XML.

```
<addObject id="NE1">
  <class>ilog.tgo.model.IltNetworkElement</class>
  <attribute name="name">NE1</attribute>
  <attribute name="shortcut">STANDARD</attribute>
  <attribute name="type">NE</attribute>
</addObject>
```

Shortcut network elements are graphically represented by an icon located at the bottom left of the network element base.



Standard shortcuts



Dangling shortcuts

Network element sizes

Network elements can be represented at various scales. The amount of information attached to the network element is proportional to the size of the network element. Two sizes are provided by default:

- ◆ The *standard size*, which displays the network element function, family, and label (in the case of a symbolic representation) and which also supports icons reflecting changes in states and alarms.



Standard size network element

- ◆ The *small size*, which displays only the network element type and label.



Site



Site



Site

Small size network elements

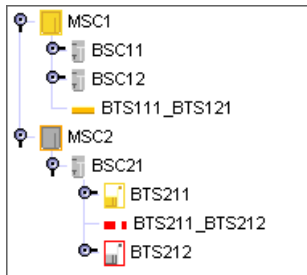
Using the network element sizes in custom programming is shown in Customizing network element types in the *Styling* documentation.

Representation of network elements in a table and in a tree

Objects of the `IltnetworkElement` class are represented in a table as follows:

Name	▼ O...	Type	Function	New Alarms	Alarms	Primary State	Secondary States
BTS212		BTS			1 C	Operational:Enabled; Us...	
NE4		Logical		1 C	1 C	Operational:Enabled; Us...	
NE3		Component			1 M	Operational:Enabled; Us...	
NE1		Network Element		1 w	2 m+	Operational:Enabled; Us...	Procedural:Reporting; Repair:Und...
MSC1		MSC		1 w	1 m+	Operational:Enabled; Us...	Procedural:Reporting; Repair:Und...
BTS211		BTS		4 w	4 w	Operational:Enabled; Us...	
MSC2		MSC		1 u	1 u	Operational:Enabled; Us...	
BSC11		BSC				Operational:Enabled; Us...	

Objects of the `IltnetworkElement` class are represented with other predefined business objects in a tree as follows:



Customizing the representation of network elements

For information on how to customize the graphic representation of network elements, refer to Customizing network elements.

Links

Explains how to use IBM® ILOG® JViews TGO predefined business objects of type link in your applications.

In this section

Classes overview

Lists the three types of link classes.

Links

Describes the attributes of the `IltLink` class and explains how to define a link in XML or how to create a link with the API.

Link sets

Describes the `IltLinkSet` class and explains how to define a link set in XML or how to create a link set with the API.

Link bundles

Describes the `IltLinkBundle` class and explains how to define a link bundle in XML or how to create a link bundle with the API.

Representation of links in a network

Presents the various graphical representations that a link can have.

Representation of links in a table and in a tree

Shows how links are represented in a table and in a tree.

Link connection ports

Explains how you can customize the way links connect to nodes by using link connection ports.

Link programming examples

Contains examples of how to create self-links, nested link sets, and links associated with the BiSONET object state.

Classes overview

A link is a physical connection between two network elements.

The complete hierarchy of predefined link business objects is as follows:

- ◆ Links : instances of the class `IltLink`
- ◆ Link sets: instances of the class `IltLinkSet`
- ◆ Link bundles: instances of the class `IltLinkBundle`

For a general introduction to predefined business objects, refer to section *Introducing business objects and data sources*.

You can retrieve any of the above classes using the corresponding `GetIlpClass` method. You can handle any instance of these classes as a simple `IlpObject` and get and set their attributes with the generic methods `getAttributeValue(ilog.cpl.model.IlpAttribute)` and `setAttributeValue(ilog.cpl.model.IlpAttribute, java.lang.Object)`.

Links

Link class

Links are used to display the transmission elements making up the network lines. They feature the same dynamic display as network elements.

Links are predefined business objects of the class `IltLink` used to represent connections between network resources.

The `IltLink` class defines the following attributes:

- ◆ **Media**—Indicates the physical medium connecting two network elements (fiber, for example).
 - **Name:** `media`
 - **Value class:** `IltLink.Media`
 - **Attribute:** `IltLink.MediaAttribute`
- ◆ **Technology**—Indicates the networking technology represented by the link (circuit switching, for example).
 - **Name:** `technology`
 - **Value class:** `IltLink.LinkTechnology`
 - **Attribute:** `IltLink.TechnologyAttribute`

The link technology is very similar to the link media. While the media represents the physical connection (fiber, electrical, for example), the technology represents the various networking technologies that a link can carry. For example, you may have two fiber links, one of them being used for voice and the other one for data.

You can retrieve the class `IltLink` using its `GetIlpClass()` method. You can handle its instances as simple `IlpObject` instances and set and get its attributes with the generic methods `getAttributeValue(java.lang.String)` and `setAttributeValue(java.lang.String, java.lang.Object)`.

Loading a link defined in XML

This section shows how to load a link from an XML file in a data source. For detailed information about data sources, see *Data sources*.

All you have to do is create a data source using the data source default implementation defined by `IltDefaultDataSource` and pass the XML file to the `parse` method of the data source, as shown below:

```
dataSource = new IltDefaultDataSource();
dataSource.parse("LinkXMLFile.xml");
```

How to define a link in XML

The following is an example of a link defined in XML format. For details about the XML elements used in this example, see *Elements in an XML data file* .

```
<cpldata>
  <addObject id="NE1">
    <class>ilog.tgo.model.IltNetworkElement</class>
    <attribute name="name">NE1</attribute>
    <attribute name="family">OC12</attribute>
    <attribute name="type">MD</attribute>
    <attribute name="position" javaClass="ilog.cpl.graphic.IlpPoint">
      <x>200</x> <y>200</y>
    </attribute>
  </addObject>
  <addObject id="NE2">
    <class>ilog.tgo.model.IltNetworkElement</class>
    <attribute name="name">NE2</attribute>
    <attribute name="type">MD</attribute>
    <attribute name="position" javaClass="ilog.cpl.graphic.IlpPoint">
      <x>400</x> <y>200</y>
    </attribute>
  </addObject>
  <addObject id="NE1-NE2">
    <class>ilog.tgo.model.IltLink</class>
    <link> <from>NE1</from> <to>NE2</to> </link>
    <attribute name="name">Link1</attribute>
    <attribute name="media">Fiber</attribute>
    <attribute name="objectState"
      javaClass="ilog.tgo.model.IltSONETObjectState">
      <state>ActiveProtecting</state>
      <protection>Exercisor</protection>
    </attribute>
  </addObject>
</cplData>
```

The following figure shows the link displayed in a network component:



Link displayed in a network component

Creating a link with the API

This section shows how to create a link through the API and how to add it to a data source.

How to create a link with the API

```
IltNetworkElement ne1 = new IltNetworkElement("NE1",
                                                IltNetworkElement.Type.MD,
                                                new IltOSIObjectState());
ne1.setAttributeValue(IltObject.PositionAttribute,
                     new IlpPoint(200, 200));
IltNetworkElement ne2 = new IltNetworkElement("NE2",
                                                IltNetworkElement.Type.MD,
                                                new IltOSIObjectState());
ne2.setAttributeValue(IltObject.PositionAttribute,
                     new IlpPoint(400, 200));
IltSONETObjectState linkState = new
    IltSONETObjectState(IltSONET.State.ActiveProtecting);
linkState.addProtection(IltSONET.End.From, IltSONET.Protection.Exercisor);
linkState.addProtection(IltSONET.End.To, IltSONET.Protection.Exercisor);
IltLink link = new IltLink (linkState, "Link1", IltLink.Media.Fiber);

IltDefaultDataSource dataSource = new IltDefaultDataSource();
dataSource.setLink(link.getIdentifier(), ne1.getIdentifier(),
                  ne2.getIdentifier());
List objs = new ArrayList();
objs.add(ne1);
objs.add(ne2);
objs.add(link);
dataSource.addObjects(objs);
```

The result looks like this:



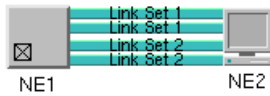
Link displayed in a network component

Link sets

Link set class

Link sets are predefined business objects of the class `IltLinkSet` used to represent a collection of links between network resources, laid out in a specific order and with a specific distance.

Link sets let you group together links between two nodes, so that the graph layout cannot insert a link that is not in the link set between them or have them follow different paths. You can fix the order of the links in the set and specify the distance that separates two links.



Link sets

Note: Link sets can be nested. In other words, they can include other link sets, which in turn can group a set of links.

The `IltLinkSet` class does not declare any specific attribute.

You can retrieve the class `IltLinkSet` using its `GetIlpClass()` method. You can handle its instances as simple `IlpObject` instances and set and get its attributes with the generic methods `getAttributeValue(java.lang.String)` and `setAttributeValue(java.lang.String, java.lang.Object)`.

Loading a link set defined in XML

This section shows how to load a link set from an XML file in a data source. For detailed information about data sources, see *Data sources*.

All you have to do is create a data source using the data source default implementation defined by `IltDefaultDataSource` and pass the XML file to the `parse` method of the data source, as shown below:

```
dataSource = new IltDefaultDataSource();
dataSource.parse("LinkSetXMLFile.xml");
```

How to define a link set in XML

The following is an example of a link set defined in XML format. For details about the XML elements used in this example, see *Elements in an XML data file*.

```
<addObject id="Link1">
  <class>ilog.tgo.model.IltLink</class>
```

```

<link> <from>Paris</from> <to>Berlin</to> </link>
<attribute name="name">1</attribute>
<attribute name="objectState" javaClass="ilog.tgo.model.IltSONETObjectState">

  <state>
    Active
  </state>
</attribute>
</addObject>

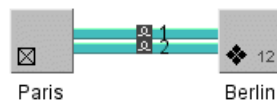
<addObject id="Link2">
  <class>ilog.tgo.model.IltLink</class>
  <link> <from>Paris</from> <to>Berlin</to> </link>
  <attribute name="name">2</attribute>
  <attribute name="objectState" javaClass="ilog.tgo.model.IltSONETObjectState">

    <state>
      Active
    </state>
  </attribute>
</addObject>

<addObject id="linkSet12">
  <class>ilog.tgo.model.IltLinkSet</class>
  <link>
    <from>Paris</from>
    <to>Berlin</to>
  </link>
  <children>
    <child>Link1</child>
    <child>Link2</child>
  </children>
</addObject>

```

The figure below shows the link set displayed in a network component:



Link set displayed in a network component

Creating a link set with the API

This section shows how to create a link set through the API and how to add it to a data source.

How to create a link set with the API

```

IltNetworkElement paris = new IltNetworkElement("Paris",

```



```

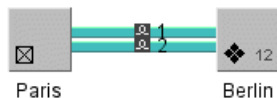
        IltNetworkElement.Type.NE, null);
    paris.setFunction(IltNetworkElement.Function.SwitchCrossConnect);
    paris.setPosition(new IlpPoint(120, 350));

    IltNetworkElement berlin = new IltNetworkElement("Berlin",
        IltNetworkElement.Type.NE, null);
    berlin.setFunction(IltNetworkElement.Function.TransportCrossConnect);
    berlin.setFamily(IltNetworkElement.Family.OC12);
    berlin.setPosition(new IlpPoint(250, 350));

    IltLink link1 = new IltLink(new IltSONETObjectState(IltSONET.State.Active),
        "1", null);
    IltLink link2 = new IltLink(new IltSONETObjectState(IltSONET.State.Active),
        "2", null);
    IltLinkSet linkSet = new IltLinkSet();
    List objects = new ArrayList();
    objects.add(paris);
    objects.add(berlin);
    objects.add(link1);
    objects.add(link2);
    objects.add(linkSet);
    dataSource.setLink(link1.getIdentifier(), paris.getIdentifier(),
        berlin.getIdentifier());
    dataSource.setLink(link2.getIdentifier(), paris.getIdentifier(),
        berlin.getIdentifier());
    dataSource.setLink(linkSet.getIdentifier(), paris.getIdentifier(),
        berlin.getIdentifier());
    dataSource.setParent(link1.getIdentifier(), linkSet.getIdentifier());
    dataSource.setParent(link2.getIdentifier(), linkSet.getIdentifier());
    dataSource.addObjects(objects);

```

The result looks like this:



Link set displayed in a network component

Link bundles

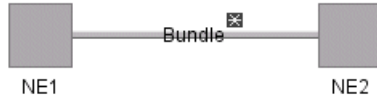
Link bundle class

Link bundles are predefined business objects of the class `IltLinkBundle` used to represent a container with an overview object and a set of detail objects. They can hold any number of links that all start at the same `IltObject` instance and all end at the same other `IltObject` instance.

You can collapse a link bundle to show only a single link. The single link has an icon that, when you click it, causes the link bundle to expand and show the child links

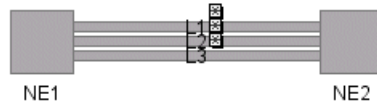
The default overview object has the normal link representation, with the same start object and the same end object.

Here is how a link bundle looks in its *collapsed* state:



Collapsed link bundle

Here is how a link bundle looks in its *expanded* state:



Expanded link bundle

The `IltLinkBundle` class does not define any specific attribute. However, any attribute defined in business class `IltLink` can be set in link bundle instances and will be graphically represented in the link bundle overview.

You can retrieve the class `IltLinkBundle` using its `GetIlpClass()` method. You can handle its instances as simple `IlpObject` instances and set and get its attributes with the generic methods `getAttributeValue(iolog.cpl.model.IlpAttribute)` and `setAttributeValue(iolog.cpl.model.IlpAttribute, java.lang.Object)`.

Loading a link bundle defined in XML

This section shows how to load a link bundle from an XML file in a data source. For detailed information about data sources, see *Data sources*.

All you have to do is create a data source using the data source default implementation defined by `IltDefaultDataSource` and pass the XML file to the `parse` method of the data source, as shown below:

```
dataSource = new IltDefaultDataSource();
dataSource.parse("LinkBundleXMLFile.xml");
```

How to define a link bundle in XML

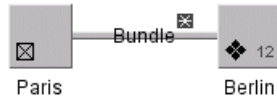
The following is an example of a link bundle defined in XML format. For details about the XML elements used in this example, see *Elements in an XML data file* .

```
<addObject id="Link1">
  <class>ilog.tgo.model.IltLink</class>
  <link> <from>Paris</from> <to>Berlin</to> </link>
  <attribute name="name">1</attribute>
  <attribute name="objectState" javaClass="ilog.tgo.model.IltSONETObjectState">
    <state>
      Active
    </state>
  </attribute>
</addObject>

<addObject id="Link2">
  <class>ilog.tgo.model.IltLink</class>
  <link> <from>Paris</from> <to>Berlin</to> </link>
  <attribute name="name">2</attribute>
  <attribute name="objectState" javaClass="ilog.tgo.model.IltSONETObjectState">
    <state>
      Active
    </state>
  </attribute>
</addObject>

<addObject id="linkBundle">
  <class>ilog.tgo.model.IltLinkBundle</class>
  <attribute name="name">Bundle</attribute>
  <link>
    <from>Paris</from>
    <to>Berlin</to>
  </link>
  <children>
    <child>Link1</child>
    <child>Link2</child>
  </children>
</addObject>
```

The figure below shows the link bundle displayed in a network component:



Link bundle displayed in a network component

The following example shows you how to create a link bundle and set states that are graphically represented when the link bundle is collapsed:

```
<addObject id="linkBundle">
  <class>ilog.tgo.model.IltLinkBundle</class>
  <attribute name="objectState" javaClass="ilog.tgo.model.IltSONETObjectState">
    <state>
      Active
    </state>
  </attribute>
  <link>
    <from>Oslo</from>
    <to>Berlin</to>
  </link>
  <children>
    <child>Link1</child>
    <child>Link2</child>
  </children>
</addObject>
```

Creating a link bundle with the API

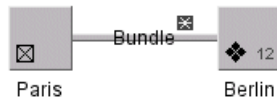
This section shows how to create a link bundle through the API and how to add it to a data source.

How to create a link bundle with the API

```
IltLink link1 = new IltLink(new IltSONETObjectState(IltSONET.State.Active),
    "1", null);
IltLink link2 = new IltLink(new IltSONETObjectState(IltSONET.State.Active),
    "2", null);
IltLinkBundle bundle = new IltLinkBundle();
bundle.setName("Bundle");
List objects = new ArrayList();
objects.add(Paris);
objects.add(Berlin);
objects.add(link1);
objects.add(link2);
objects.add(bundle);
dataSource.setLink(link1.getIdentifier(), Paris.getIdentifier(),
    Berlin.getIdentifier());
dataSource.setLink(link2.getIdentifier(), Paris.getIdentifier(),
```

```
berlin.getIdentifier());
dataSource.setLink(bundle.getIdentifier(), paris.getIdentifier(),
    berlin.getIdentifier());
dataSource.setParent(link1.getIdentifier(), bundle.getIdentifier());
dataSource.setParent(link2.getIdentifier(), bundle.getIdentifier());
dataSource.addObjects(objects);
```

The result looks like this:



Link bundle displayed in a network component

Representation of links in a network

As shown in the following figures, links can be represented in various colors and line types, depending on the state they are in. Links can show an icon representing their secondary state or have a label. They can also be displayed with an information cluster showing associated alarms. For a reference list of link states, see *Graphical representation of SONET primary states* and *Graphical representation of SONET secondary states*.

These display modes apply to all kinds of link. Link representations can also display the link physical medium (see *Link media*), its networking technology (see *Link technology*), its orientation (see *Oriented links*), or whether this link is linked to itself (see *Self-links*).



Links disabled, inactive, and active states



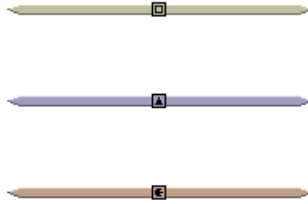
Link with status icons



Link with label



Links showing media attribute



Links showing technology attribute

Generally, the label appears at the center of the link. When the link displays additional information, such as the media icon or alarms, the label is moved either below that information or to the right of it.



Link with alarm cluster

Link media

The link media is represented with an icon that appears at its center. The following table lists the predefined media icons.

Link media representation









Link Media	Representation	Media Name
Communication network		CNET
Fiber		Fiber
Electrical		Electrical

You can extend this small set of predefined link media using a dedicated API, which is detailed in Customizing link media in the *Styling* documentation.

Link technology

The link technology is represented by an icon in the center of the link, and a corresponding base color. The following table lists the predefined technology icons and colors.

Link technology representation

Link Technology	Representation	Technology	Technology Name
Circuit switching		Switching	CircuitSwitching
ATM/Frame Relay		Switching	ATM_FrameRelay
Wireless Edge		Edge	WirelessEdge
IP		IP	IP
SONET/SDH		Transport	SONET_SDH
DWDM Optical		Transport	DWDM_Optical
Multi Layer		Multiple	MultiLayer
Other		Unknown	Other

As this table implies, the default representation uses icons to identify link technologies graphically and colors to group similar technologies together. You can extend this small set of predefined link technologies through the dedicated API or CSS, see Customizing links. Note also that the link technology color is overridden by the primary state color defined by the link object state.

Oriented links

Oriented links provide a representation for links with an arrow at one end or at both.



Link with an arrow



Link with two arrows

The presence or absence of arrows, as well as their graphical characteristics are driven by dedicated CSS properties. (See table CSS properties applying to arrows on link base elements in the *Styling* documentation.) By default, no arrow is displayed except in the case of links that have an object state of type `IlTbiSONETObjectState`. These links have arrows at both

ends with predefined graphical characteristics. For a reference list of the common double SONET states, see *Common pairs of SONET primary states* .

Self-links

A self-link has both ends connected to the same network element.







Self-link

In a self-link, the origin and the destination are the same. See *Link programming examples* for an example on how to create a self-link.

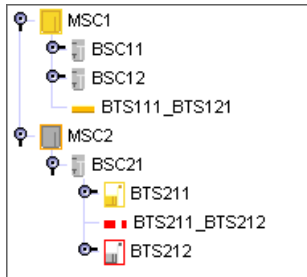
For information on how to customize the graphic representation of links, refer to *Customizing links*.

Representation of links in a table and in a tree

Objects of the `IlLink` class are represented in a table as follows:

Object	Name	Media	New Alarms	Alarms	Primary State	Secondary States
	BTS211_BTS212				SONET State:Troubled U...	
	BTS111_BTS121			1 m	SONET State:Active Prot...	

Objects of the `IlLink` class are represented (with other predefined business objects) in a tree as follows:



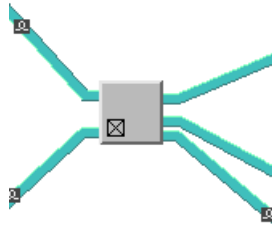
Link connection ports

A connection port is the logical position where a link is attached to a node. Normally, this position is automatically determined by the link layout algorithm. Link connection ports provide a way to modify this behavior by forcing the links into a specific position that will be taken into account by the graph layout algorithm.

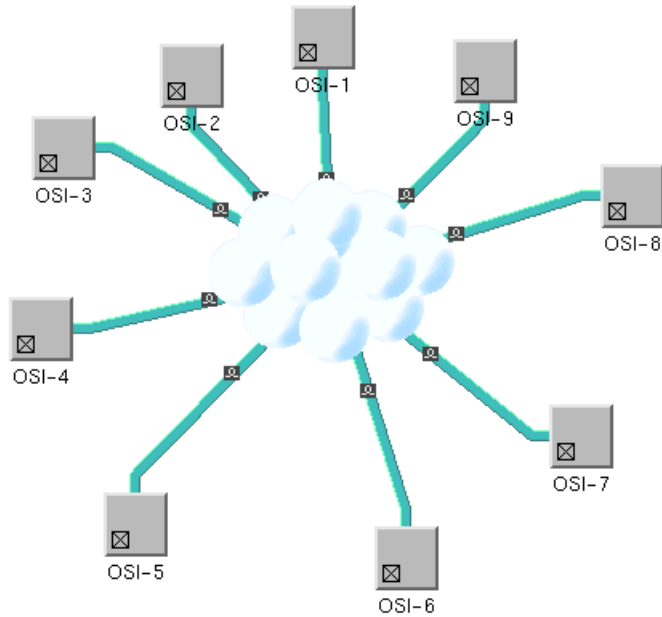
By default, connection ports are not visible on a node. However, you can display them by attaching to the view an interactor of type `IltMakeLinkInteractor` operating in pin mode.

Connection ports can be either directed or undirected. Links attached to a directed connection port originate from a specific direction given by the port, and are drawn either vertically or horizontally, depending on the direction. If the main segment of the link is neither horizontal nor vertical, a short segment is added to the extremity of the link to compensate for this. This is the normal behavior implemented by the direct link style of the link layout algorithm. On the other hand, links coming from an undirected connection port can go in any direction, defined randomly by the layout algorithm. In this case, no additional segment is appended to their extremities.

These two types of connection port are illustrated in the following figures:



A node with four directed connection ports



A node with an undirected link connection port (at the center of the cloud)

A link connection port is defined by the following:

- ◆ The point on the node where the link should end. This point is specified with two floating point values, f_x and f_y , ranging between 0 and 1. These are proportional values, relative to the link connection rectangle.

Note: The connection rectangle is generally represented by the bounding box of the node, except for groups for which this connection rectangle can be the group plinth.

- $f_x = 0$ corresponds to the left side
- $f_x = 0.5$ corresponds to the vertical center line
- $f_x = 1$ corresponds to the right side
- $f_y = 0$ corresponds to the top side
- $f_y = 0.5$ corresponds to the horizontal center line
- $f_y = 1$ corresponds to the bottom

Note: When there is more than one link, and the connection port is a directed one, the link bundling feature of the graph layout makes the link end in the vicinity of this point, not exactly on it.

- ◆ The direction of the outgoing link. For directed link connection ports, this direction is generally defined by `IlvDirection.Top`, `IlvDirection.Bottom`, `IlvDirection.Left`, and `IlvDirection.Right`, depending on which side of the node the link starts from. For undirected links, this value is `IlvDirection.Center`.
- ◆ Optionally, the distance between two links ending at the same link connection port. Negative values are ignored. This option is generally used to override the distance between two links set for the `IltShortLinkLayout` instance of a specific port.

When a link connection port is applied to a node, the target point is computed from the values of `fx`, `fy`, and the link connection rectangle.

The class `IltLinkPort` is an enumeration type that defines the following values: `Top`, `Bottom`, `Left`, `Right`, and `Center`. The first four values indicate that the link should connect to the center point of the corresponding node side. For example, `Left` specifies that the link will be attached to the center of the left side of the node. These values apply to directed link ports. The last value, `Center`, applies to undirected link ports. These values can be customized through CSS as illustrated in [Customizing link port configuration in the *Styling* documentation](#). You can create your own instances of `IltLinkPort` by providing specific values to the `fx`, `fy`, and `direction` parameters.

Only the `IltShortLinkLayout` and `IltLinkLayout` (when set in short link mode) layout algorithms implement link connection ports. When used in an `JViews TGO` network component, these layouts use the following optional information through the `IltDefaultNodeSideFilter` that is installed automatically:

- ◆ Each node can specify an array of pins as the `linksPorts` property. If no array is defined, the following set of values is used by default: `{IltLinkPort.Top, IltLinkPort.Bottom, IltLinkPort.Left, IltLinkPort.Right}`.

For information on how to configure link ports, refer to [Customizing link port configuration in the *Styling* documentation](#).

- ◆ Each link can specify the link port to which it will connect at both ends through the `fromPort` and `toPort` properties. If no link port is defined, the layout algorithm will select one among those that are allowed.

How to create link connection ports

1. Create an `IltShortLinkLayout` algorithm and attach it to the network component .

```
IltShortLinkLayout layout = new IltShortLinkLayout();
...
network.setLinkLayout(layout);
```

This layout algorithm allows you to handle link connection ports. You could also use `IltLinkLayout`.

2. Create new link connection ports using `IltLinkPort`.

```
...  
IltLinkPort port1=  
    new IltLinkPort("Right, above middle",1.0f,0.2f,1,0,-1);
```

Customizing the representation of links

For information on how to configure the link connection ports, refer to Customizing link port configuration in the *Styling* documentation.

Link programming examples

The same example using XML can be found in `<installdir>/samples/network/links`.

How to create a network

The network is created using the network component and the data source API, as illustrated in the following code:

```
/**
 * Execute the main part of the sample.
 */
void doSample (Container container) {
    try {
        // Initialize JTGO
        // Read a deployment descriptor file to initialize JTGO services
        if (isApplet())
            IltSystem.Init(this, "deploy.xml");
        else
            IltSystem.Init("deploy.xml");

        // Create a datasource
        IltDefaultDataSource dataSource = new IltDefaultDataSource();
        fillNetwork(dataSource);

        // Create a network component
        IlpNetwork networkComponent = new IlpNetwork();

        // Connect network component to datasource
        networkComponent.setDataSource(dataSource);

        // Add view to the frame
        container.add(networkComponent);
    }
    catch(Exception e){
        e.printStackTrace();
    }
}
```

How to add network elements and links to the network

The function `fillNetwork` creates network elements and links and adds these instances to the given data source. Then the data source is linked to the network component.

```
/**
 * Adds nodes and all kinds of links to the network.
 */
public void fillNetwork (IltDefaultDataSource dataSource) {
    // Create the network elements.
```

```

IltNetworkElement paris;
IltNetworkElement berlin;
IltNetworkElement oslo;
IltNetworkElement nwest;

paris =
    new IltNetworkElement("Paris",
        IltNetworkElement.Type.NE,
        IltNetworkElement.Function.SwitchCrossConnect,
        null,
        new IltBellcoreObjectState());
berlin =
    new IltNetworkElement("Berlin",
        IltNetworkElement.Type.NE,
        IltNetworkElement.Function.TransportCrossConnect,
        IltNetworkElement.Family.OC12,
        new IltBellcoreObjectState());
oslo =
    new IltNetworkElement("Oslo",
        IltNetworkElement.Type.NE,
        IltNetworkElement.Function.Transport,
        null,
        new IltBellcoreObjectState());
nwest =
    new IltNetworkElement("North West",
        IltNetworkElement.Type.NMW,
        new IltBellcoreObjectState());

// Place the network elements at the right location while inserting
// them in the network.
paris.setState(IltBellcore.State.EnabledActive);
paris.setAttributeValue(IltObject.PositionAttribute, new IlpPoint(120,
                                                                    365));
berlin.setState(IltBellcore.State.EnabledActive);
berlin.setAttributeValue(IltObject.PositionAttribute, new IlpPoint(300,
                                                                    250)
);
oslo.setState(IltBellcore.State.EnabledActive);
oslo.setAttributeValue(IltObject.PositionAttribute, new IlpPoint(190,
                                                                    100));
nwest.setState(IltBellcore.State.EnabledActive);
nwest.setAttributeValue(IltObject.PositionAttribute, new IlpPoint(37, 40)
);

dataSource.addObject(paris);
dataSource.addObject(berlin);
dataSource.addObject(oslo);
dataSource.addObject(nwest);

// Add all kinds of links
addSelfLink(dataSource, paris);

```



```
    addBiSONETLinks(dataSource, oslo, nwest);
}
```

How to create a self-link

The `addSelfLink` method creates a self link to Paris with an associated state and an arrow. Note that in self-links the "From" and "To" nodes are the same.

```
/**
 * Adds a link with the given node as both its from and to end.
 */
public void addSelfLink (IltDefaultDataSource dataSource, IltObject node) {
    IltLink link = new IltLink(new IltSONETObjectState(), null);
    link.setState(IltSONET.State.Active);
    dataSource.addObject(link);
    dataSource.setLink(link.getIdentifier(),
                       node.getIdentifier(),
                       node.getIdentifier());
}
```

How to create links with BiSONET object states

The `addBiSONETLinks` method creates two links with an object state of type `IltBiSONETObjectState`. The `setReverseState` method is used to set the state of the link in the opposite direction.

```
/**
 * Adds some bi-SONET links between the given nodes.
 */
public void addBiSONETLinks (IltDefaultDataSource dataSource,
                             IltObject node1,
                             IltObject node2) {

    // Create a bi-SONET link.
    IltBiSONETObjectState biState = new IltBiSONETObjectState();
    IltLink link = new IltLink(biState);
    // Set its two states.
    biState.setState(IltSONET.State.TroubledProtected);
    biState.setReverseState(IltSONET.State.Active);
    dataSource.addObject(link);
    dataSource.setLink(link.getIdentifier(),
                       node1.getIdentifier(),
                       node2.getIdentifier());

    IltBiSONETObjectState biState2 = new IltBiSONETObjectState();
    IltLink link2 = new IltLink(biState2);
    biState2.setState(IltSONET.State.ActiveProtecting);
    biState2.setReverseState(IltSONET.State.TroubledUnprotected);
    dataSource.addObject(link2);
    dataSource.setLink(link2.getIdentifier(),
                       node2.getIdentifier(),
```

```
        model.getIdentifier();  
    }
```

Groups

Explains how to use IBM® ILOG® JViews TGO predefined business objects of type group in your applications.

In this section

Group class

Describes the attributes of the IltGroup class.

Group shapes

Describes the three different shapes of groups and the corresponding classes.

Loading a group defined in XML

Shows how to load a group from an XML file into a data source.

Creating a group with the API

Shows how to create groups using the JViews TGO API and add them to a data source.

Representation of groups in a table and in a tree

Shows how groups are represented in a table and in a tree.

Group class

Groups are predefined business objects of the class `IltGroup` that are used to represent a set of network resources grouped logically or geographically. For a general introduction to predefined business classes, see *Introducing business objects and data sources*.

The `IltGroup` class defines the following attributes:

- ◆ `Icon`—Specifies an image representing the group category:
 - **Name:** `icon`
 - **Value class:** `IlSerializableImage`
 - **Attribute:** `IltGroup.IconAttribute`
- ◆ `Shortcut`—Indicates that the network element is only a reference to an existing network element.
 - **Name:** `shortcut`
 - **Value class:** `ilog.tgo.model.attribute.IltShortcutAttributeType`
 - **Attribute:** `IltNetworkElement.ShortcutAttribute`

There are three types of groups characterized by a different shape and defined by the following `IltGroup` subclasses:

- ◆ `IltPolyGroup`,
- ◆ `IltRectGroup`, and
- ◆ `IltLinearGroup`

For more information, see *Group shapes*.

You can retrieve the class `IltGroup` using its `GetIlpClass()` method. You can handle its instances as simple `IlpObject` instances and set and get its attributes with the generic methods `getAttributeValue(java.lang.String)` and `setAttributeValue(java.lang.String, java.lang.Object)`.

Group shapes

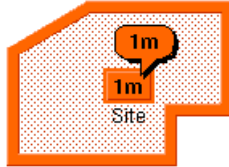
A group object can have one of the following three shapes:

Polygonal shape

Polygonal groups are defined by the class `IltPolyGroup`. Polygonal groups are very useful for dividing a network into regions and associating those regions with topographic zones visible on a map.

The shape of a polygonal group is defined by the class `IlpPolygon`. This class describes a closed polyline made up of an array of points. This polyline can have any number of sides.

Polygonal groups have a semitransparent background (through which a background map can be seen) and a thick outline. When alarms or statuses are displayed, they are grouped in an information cluster that is positioned at the center of the polygon, as shown in the following figure. For more information, see *Customizing the group information cluster*.



Polygonal group with information cluster

Rectangular shape

Rectangular groups are defined by the class `IltRectGroup`. Rectangular groups are generally used to hold network elements located in the same place such as a site, a building, or a city.

The shape of a rectangular group is defined by the class `IlpRect`, which describes a rectangle. Rectangular groups can be resized to create any kind of rectangular container.

Rectangular groups are represented by opaque relief rectangles as shown in the following figure. When alarms are displayed, they are grouped in an information cluster that is positioned at the center of the rectangle. For more information, see *Customizing the group information cluster*.



Rectangular group with information cluster

Linear shape

Linear groups are defined by the class `IltLinearGroup`.

Linear groups represent a linear collection of objects and can be used to display, for example, all the repeaters between two line termination network elements.

The shape of a linear group is defined by the class `IlpPolyline`. This class describes an open polyline made up of an array of points.

When alarms or secondary states are displayed on a linear group, an information cluster appears at the center of its median segment. The median segment is the segment containing the midpoint of the shape.



Linear group with information cluster

Loading a group defined in XML

All you have to do is create a data source using the data source default implementation defined by `IltDefaultDataSource` and pass the XML file to the `parse` method of the data source, as shown below.

```
dataSource = new IltDefaultDataSource();
dataSource.parse("GroupXMLFile.xml");
```

For detailed information about data sources, see *Data sources*.

How to define a group in XML

The following is an example of a group defined in XML format. For details about the XML elements used in this example, see *Elements in an XML data file*.

Depending on the group class, the shape of a group is defined either by `IlpPolygon`, `IlpRect`, or `IlpPolyline`. See *Group shapes* for details.

```
<cplData>
<addObject id="RectGroup">
  <class>ilog.tgo.model.IltRectGroup</class>
  <attribute name="name">RectGroup</attribute>
  <attribute name="position" javaClass="ilog.cpl.graphic.IlpRect">
    <x>100</x> <y>200</y> <width>100</width> <height>50</height>
  </attribute>
  <attribute name="objectState" javaClass="ilog.tgo.model.IltOSIObjectState">
    <state>
      <administrative>Locked</administrative>
      <operational>Enabled</operational>
      <usage>Idle</usage>
    </state>
    <availability>PowerOff</availability>
    <control>ReservedForTest</control>
    <alarms>
      <new severity="Critical">5</new>
      <ack severity="Warning">12</ack>
    </alarms>
  </attribute>
</addObject>
<addObject id="PolyGroup">
  <class>ilog.tgo.model.IltPolyGroup</class>
  <attribute name="name">PolyGroup</attribute>
  <attribute name="position" javaClass="ilog.cpl.graphic.views.IlpPolygon">
    <point <x>50.0</x> <y>20.0</y> </point>
    <point <x>140.0</x> <y>20.0</y> </point>
    <point <x>140.0</x> <y>100.0</y> </point>
    <point <x>90.0</x> <y>100.0</y> </point>
    <point <x>90.0</x> <y>140.0</y> </point>
    <point <x>20.0</x> <y>140.0</y> </point>
```

```

    <point> <x>20.0</x> <y>90.0</y> </point>
  </attribute>
  <attribute name="objectState"
    javaClass="ilog.tgo.model.IltBellcoreObjectState">
    <state>EnabledActive</state>
    <secState>TestFailure</secState>
    <procedural>Initializing</procedural>
    <misc>HighTemperatureWarning</misc>
    <performance state="Input">50</performance>
    <alarms>
      <new severity="Minor">3</new>
      <ack severity="Warning">4</ack>
    </alarms>
  </attribute>
</addObject>
<addObject id="LinearGroup">
  <class>ilog.tgo.model.IltLinearGroup</class>
  <attribute name="name">LinearGroup</attribute>
  <attribute name="position" javaClass="ilog.cpl.graphic.views.IlpPolyline">
    <point> <x>350.0</x> <y>100.0</y> </point>
    <point> <x>400.0</x> <y>100.0</y> </point>
    <point> <x>420.0</x> <y>50.0</y> </point>
    <point> <x>450.0</x> <y>50.0</y> </point>
  </attribute>
  <attribute name="objectState"
    javaClass="ilog.tgo.model.IltBellcoreObjectState">
    <state>EnabledIdle</state>
    <secState>Busy</secState>
    <procedural>Initializing</procedural>
    <misc>DoorAjar</misc>
    <alarms>
      <ack severity="Warning">2</ack>
    </alarms>
  </attribute>
</addObject>
</cplData>

```

Creating a group with the API

Depending on the group class, the shape of a group is defined either by `IlpPolygon`, `IlpRect`, or `IlpPolyline`. See *Group shapes* for details.

How to create a group with the API

```
IltRectGroup rectgroup = new IltRectGroup(new IltOSIObjectState(),
                                           "RectGroup");
rectgroup.setAttributeValue(IltObject.PositionAttribute, new IlpRect(100, 200,
                                                                    100, 50)
);

IltPolyGroup polygroup = new IltPolyGroup (new IltBellcoreObjectState(),
                                           "PolyGroup");
polygroup.setAttributeValue(IltObject.PositionAttribute,
                            new IlpPolygon(new IlvPoint[] {
new IlvPoint(50, 20),
new IlvPoint(140, 20),
new IlvPoint(140, 100),
new IlvPoint(90, 100),
new IlvPoint(90, 140),
new IlvPoint(20, 140),
new IlvPoint(20, 90)
})));

IltLinearGroup lineargroup = new IltLinearGroup (new IltBellcoreObjectState()
,
                                           "LinearGroup");
lineargroup.setAttributeValue(IltObject.PositionAttribute,
                              new IlpPolyline(new IlvPoint[] {
new IlvPoint(350, 100),
new IlvPoint(400, 100),
new IlvPoint(420, 50),
new IlvPoint(450, 50)
})));

IltDefaultDataSource dataSource = new IltDefaultDataSource();
dataSource.addObject(rectgroup);
dataSource.addObject(polygroup);
dataSource.addObject(lineargroup);
```

The group constructor has two arguments:

- ◆ A newly created `IltObjectState` from the class corresponding to the chosen standard.
- ◆ A name that is displayed at the center of the group representation.





In addition to the group name, an icon can be set to each group instance. This icon will be displayed above the group label and plinth, if any. The following code extract shows how a logo (logo.png) is set on one of the groups of the sample.

```
IlpContext context = IltSystem.GetDefaultContext();  
IlpImageRepository imageRep = context.getImageRepository();  
Image groupImage = imageRep.getImage("logo.png");  
group.setIcon(groupImage);
```

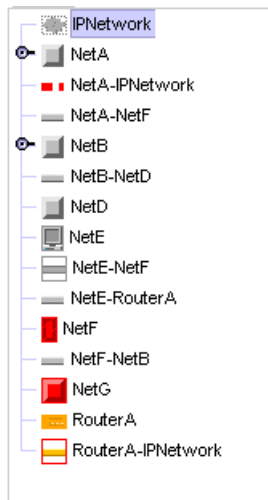
Here the image is obtained from a disk file.

Representation of groups in a table and in a tree

Objects of the `IltGroup` class (`IltPolyGroup`, `IltRectGroup`, and `IltLinearGroup`) are represented in a table as follows:

Object	Name	Icon	New Alarms	Alarms	Primary State	Secondary States
	NetA				Operational:Enabled; Usa...	Availability:Log Full
	NetB				Operational:Enabled; Usa...	Availability:Failed
	NetD				Operational:Enabled; Usa...	Availability:Power Off; Control:Rese...
	NetG		1 C+	1 C+	Operational:Enabled; Usa...	

Objects of the `IltGroup` class (`IltPolyGroup`, `IltRectGroup`, and `IltLinearGroup`) are represented in a tree as follows:



Customizing the representation of groups

For information on how to customize the graphic representation of groups, refer to Customizing groups.

Subnetworks

Explains how to use subnetworks in your applications.

In this section

About subnetworks

Describes subnetworks and how they are displayed.

Loading a subnetwork defined in XML

Shows how to load a subnetwork from an XML file in a data source.

Creating a subnetwork with the API

Shows how to create a subnetwork through the API and how to add it to a data source.

Representing alarms in expanded subnetworks

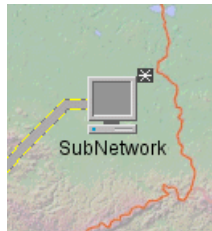
Describes the meaning of the visual elements used to represent alarms in a subnetwork.

About subnetworks

Subnetworks allow you to create applications that display a network inside another network. They are created automatically by the IBM® ILOG® JViews TGO network component when you define a containment relationship between objects in the data source.

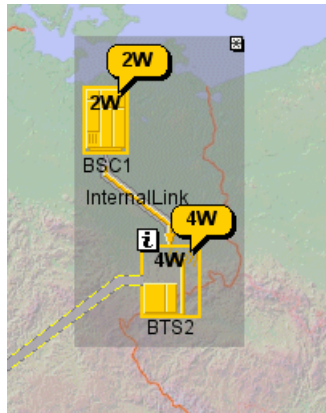
A subnetwork can be defined as a node object with child objects in a network component. It can be displayed either collapsed or expanded.

- ◆ In the *collapsed* state, the subnetwork is represented as a single object.



Collapsed subnetwork

- ◆ In the *expanded* state, the subnetwork is displayed with all the objects contained in it.



Expanded subnetwork

For information on how to customize the graphic representation of subnetworks, refer to [Customizing subnetworks](#).

Loading a subnetwork defined in XML

All you have to do is create a data source using the data source default implementation defined by `IltDefaultDataSource` and pass the XML file to the `parse` method of the data source, as shown below:

```
dataSource = new IltDefaultDataSource();
dataSource.parse("SubnetworkXMLFile.xml");
```

For detailed information about data sources, see *Data sources*.

How to define a subnetwork in an XML file

The following is an example of a subnetwork defined in XML format. For details about the XML elements used in this example, see *Elements in an XML data file*.

The example creates a network element with identifier `SubNetwork1`. This network element is automatically interpreted by the network component as a subnetwork when you add the containment relationship using the XML tag `<parent>`.

The example creates a network element as a subnetwork that contains two children objects connected by a link.

◆ Add the subnetwork

```
<addObject id="SubNetwork1">
  <class>ilog.tgo.model.IltNetworkElement</class>
  <attribute name="name">SubNetwork</attribute>
  <attribute name="type">NMW</attribute>
</addObject>Add the subnetworks.
```

◆ Add the child objects

```
<addObject id="SubNode1">
  <class>ilog.tgo.model.IltNetworkElement</class>
  <parent>SubNetwork1</parent>
  <attribute name="name">BSC1</attribute>
  <attribute name="type">BSC</attribute>
  <attribute name="position" javaClass="ilog.cpl.graphic.IlpPoint">
    <x>580</x> <y>80</y>
  </attribute>
  <attribute name="objectState"
javaClass="ilog.tgo.model.IltAlarmObjectState">
    <alarms>
      <new severity="Warning">2</new>
    </alarms>
  </attribute>
</addObject>

<addObject id="SubNode2">
```

```

<class>ilog.tgo.model.IltNetworkElement</class>
<parent>SubNetwork1</parent>
<attribute name="name">BTS2</attribute>
<attribute name="type">BTS_Image</attribute>
<attribute name="position" javaClass="ilog.cpl.graphic.IlpPoint">
  <x>620</x> <y>180</y>
</attribute>
<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>
</addObject>

```

◆ Create a link connecting the two child objects

```

<addObject id="SubNode1-SubNode2">
  <class>ilog.tgo.model.IltLink</class>
  <parent>SubNetwork1</parent>
  <link> <from>SubNode1</from> <to>SubNode2</to> </link>
  <attribute name="name">InternalLink</attribute>
  <attribute name="media">null</attribute>
  <attribute name="objectState"
javaClass="ilog.tgo.model.IltBiSONETObjectState">
    <state>ActiveProtecting</state>
  </attribute>
</addObject>

```

Creating a subnetwork with the API

The following procedure creates the same subnetwork as in *Loading a subnetwork defined in XML* but through coding.

1. Create the subnetwork

```
IltNetworkElement subNetwork1 = new IltNetworkElement("SubNetwork1");
subNetwork1.setName("SubNetwork");
subNetwork1.setType(IltNetworkElement.Type.NMW);
```

2. Create the first child objects

```
List children = new ArrayList();
IltNetworkElement subNode1 = new IltNetworkElement("SubNode1");
subNode1.setName("BSC1");
subNode1.setType(IltNetworkElement.Type.BSC);
subNode1.setPosition(new IlpPoint(580, 80));
IltAlarmObjectState alarmState = new IltAlarmObjectState();
IltAlarm.State alarms = (IltAlarm.State)alarmState.getAlarmState();
alarms.setNewAlarmCount(IltAlarm.Severity.Warning, 2);
subNode1.setObjectState(alarmState);
children.add(subNode1);
```

3. Add the first child object to the subnetwork

```
datasource.setParent(subNode1, subNetwork1);
```

4. Create the second child object

```
IltNetworkElement subNode2 = new IltNetworkElement("SubNode2");
subNode2.setName("BTS2");
subNode2.setType(IltNetworkElement.Type.BTS_Image);
subNode2.setPosition(new IlpPoint(620, 180));
IltOSIObjectState osiState =
    new IltOSIObjectState(new IltOSI.State(IltOSI.State.Operational.Enabled,
                                           IltOSI.State.Usage.Active,
                                           IltOSI.State.Administrative.ShuttingDown));
IltAlarm.State alarms = (IltAlarm.State)osiState.getAlarmState();
alarms.setNewAlarmCount(IltAlarm.Severity.Warning, 4);
osiState.set(IltOSI.Procedural.Reporting);
osiState.set(IltOSI.Repair.UnderRepair);
osiState.set(IltPerformance.SecState.Output, new Float(150));
subNode2.setObjectState(osiState);
children.add(subNode2);
```

5. Add the second child to the subnetwork

```
datasource.setParent(subNode2, subNetwork1);
```

6. Create an internal link

```
IltLink link = new IltLink("SubNode1-SubNode2");  
link.setName("InternalLink");  
link.setMedia(null);  
link.setObjectState(new  
    IltBiSONETObjectState(IltSONET.State.ActiveProtecting,  
        null));  
children.add(link);
```

7. Add the link to the subnetwork

```
datasource.setParent(link, subNetwork1);
```

8. Add the link relationship

```
datasource.setLink(link, subNode1, subNode2);
```

9. Add all the objects to the data source

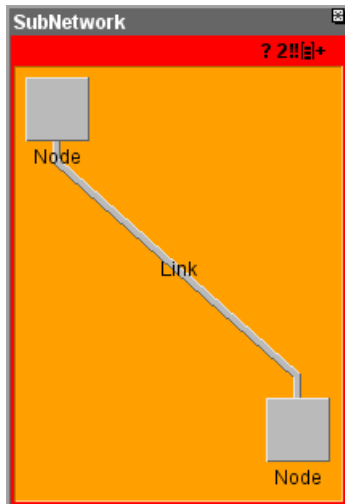
```
datasource.addObject(subNetwork1);  
datasource.addObjects(children);
```

Note: For a subnetwork to be properly displayed, the parent object must be customized as a container node. Customization is achieved through CSS using the **expansion** property. For details, refer to Customizing subnetworks.

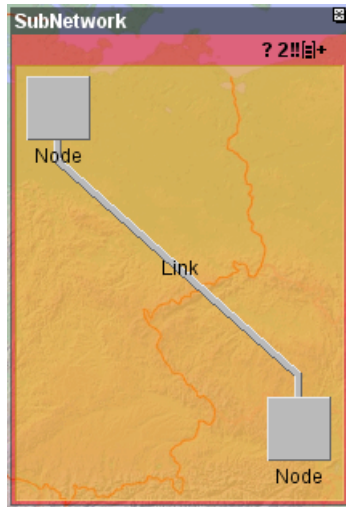
Representing alarms in expanded subnetworks

In a network view, the collapsed representation of a subnetwork corresponds to a standard node. This means that the alarm representation conforms to the type of object, for example, a network element or a polygon group. On the other hand, the expanded representation of a subnetwork is always defined by a frame containing all the child objects. The expanded representation can also display Raw and Impact alarm information. This is done in a way that preserves the containment perception while not interfering with the individual alarm representation of child objects.

Unlike other predefined business objects, the expanded subnetwork can occupy a large view space, sometimes even larger than the visible area. This imposes some restrictions on the standard graphical mapping of alarm information. Additionally, the subnetwork alarm mapping cannot interfere with the alarm representation of the child objects while also being consistent with the frame configuration defined by the user.



Subnetwork with impact alarm and solid color



Subnetwork with impact alarm and transparency

The alarm is represented as an internal frame surrounding all child objects within the expanded subnetwork object. The following alarm information is mapped:

◆ Alarm Count Summary

Located in the top-right portion of the alarm frame, the alarm count summarizes information about the highest severity new or outstanding alarm. It also displays the `Not Reporting` and `Loss of Connectivity` abbreviations when applicable.

◆ Most Severe New Alarm

The internal area of the alarm frame maps the color of the most severe new alarm. When there are no new alarms, this area automatically switches to the expanded subnetwork background color.

◆ Most Severe Outstanding Alarm

The border around the alarm frame maps the color of the most severe outstanding alarm. When there are no outstanding alarms, this border automatically switches to the expanded subnetwork background color.

The subnetwork customization can influence the displaying of the alarm frame in two ways:

1. By defining the transparency level
2. By disabling completely the alarm mapping

The alarm frame transparency is supported in the same way as the background transparency: if the subnetwork background (set through CSS property `subnetworkBackground`) has a color with transparency, the alarm frame will automatically use the same alpha level, leveraging the user's customization. Please note that rendering transparency consumes more CPU cycles due to color blending with the background and this might be noticeable with larger subnetworks. Additionally, very low alpha levels (very transparent colors) might affect the correct alarm perception. In the following example, the background color is set to 50% transparent white, which will enable 50% transparency (7F) for the alarm frame:

```
#MySubnetworkId {
  subnetworkBackground: #7FFFFFFF;
}
```

The subnetwork frame type (set through CSS property `subnetworkFrame`) determines whether the alarm frame is displayed or not. For `TITLEBAR_FRAME` and `FILLED_RECTANGLE_FRAME` types, the alarm frame automatically appears when the subnetwork has an alarm. For type `NO_FRAME`, the alarm frame never appears because there is no frame nor background for this subnetwork frame type. Please note that the alarm frame visibility is also affected by alarm CSS properties, more precisely by the properties `alarmColorVisible`, `alarmCountVisible`, and `alarmBorderVisible`.

As expanded subnetworks are special containment objects, some alarm mappings do not apply to them. For example, secondary alarm decorations and alarm balloons are not supported, as it would be difficult for an operator to notice such decorations on large subnetworks where they may not be visible at all. The complete list of special cases is:

- ◆ Alarm Balloon not mapped
- ◆ Secondary Alarm decorations not mapped
- ◆ Tool tip text not enabled by default

The tool tip can be enabled through CSS as follows:

```
#MySubnetworkId:expanded {
  tooltipText: '@|alarmSummary("Default", "Description")';
}
```

Note that this code uses the `alarmSummary` CSS function to retrieve the default description for alarms. It also uses the `expanded` pseudoclass to apply this CSS property only to the expanded subnetwork. This pseudoclass enables different CSS styling for the collapsed and expanded subnetworks. The following example illustrates how to customize different alarm borders and alarm count fonts for expanded and collapsed representations:

```
#MySubnetworkId {
  alarmBorderWidth: 2;
  alarmCountFont: 'arial-bold-12';
}
#MySubnetworkId:expanded {
  alarmBorderWidth: 4;
  alarmCountFont: 'arial-bold-14';
}
```

Please refer to the *Alarm states* section for more details on alarms and how to customize them.

Shelves and cards

Explains how to create physical views of telecommunication equipment in the form of shelves holding cards (shelf items), and cards holding ports and LEDs (card items).

In this section

Overview of classes

Lists the classes of predefined objects for shelves and cards.

Shelves

Describes the facilities available for shelves.

Shelf items

Describes the following shelf items: cards, empty slots, and card carriers.

Card items

Describes the following card items: LEDs and ports.

Representation of shelves and cards in a table and in a tree

Shows how shelves and cards are represented in a table and in a tree.

Overview of classes

The complete hierarchy of predefined business objects is as follows:

- ◆ Shelves: instances of the class `IltShelf`
- ◆ Cards: instances of the class `IltCard`
- ◆ Empty slots: instances of the class `IltEmptySlot`
- ◆ Card carriers: instances of the class `IltCardCarrier`
- ◆ Ports: instances of the class `IltPort`
- ◆ LEDs: instances of the class `IltLed`

For a general introduction to predefined business objects, refer to *Introducing business objects and data sources*.

You can retrieve any of the above classes using the corresponding `GetIlpClass` method. You can handle any instance of these classes as a simple `IlpObject` and get and set their attributes with the generic methods `getAttributeValue(java.lang.String)` and `setAttributeValue(java.lang.String, java.lang.Object)`.

Shelves

Describes the facilities available for shelves.

In this section

Overview of shelves

Provides details about the components of a shelf.

Shelf class

Describes the attributes of the `IltShelf` class.

Loading a shelf defined in XML

Shows how to load a shelf from an XML file in a data source.

Creating a shelf with the API

Shows how to create a shelf using the API and how to add it to the data source.

Overview of shelves

A standard shelf is a rectangular frame made up of smaller rectangles (slots) placed side by side in a line. Each slot is assigned a number (slot number) that graphically identifies the slot in the shelf.

A complex shelf is a rectangular frame made up of an array of slots, each column being assigned a number (slot numbering) that graphically identifies it within the shelf.

Each slot can hold one card object. There are three different types of card object: cards, empty slots, and card carriers. Each of these types is described in this topic.

Although states and alarms can be associated with a shelf, it is not possible to display them in the shelf graphic representation. This is not necessary since shelves are only holders of shelf items. On the other hand, the states and alarms associated with shelf items can be represented graphically.

It is possible to set a label for any shelf, but it will only be visible in the logical and tiny representations. The symbolic representation does not display it.

For information on other styling capabilities available for shelves, refer to CSS properties for the representation of shelves, card carriers, cards and ports .

Slots are numbered sequentially (slot numbering), starting from the initial index which is passed as an argument to the constructor (`logicalFirstIndex` argument in most `IlTShelf` constructors). By default, the slot labels display the sequential numbers, but you can customize them using cascading style sheets.

Slots in a shelf can be defined as fixed-width, where all slots have the same width, or variable-width, where each slot has its width set individually. A slot is not displayed when its width is set to zero, which breaks the sequence of the slot numbering. The width of the slots can be defined either when constructing a shelf or by invoking the shelf API (method `setSlotSizes`).

For array shelves, slot columns are numbered sequentially and both the horizontal and the vertical dimensions can be defined as fixed-width or variable-width using an API similar to standard linear shelves.

Since the shelf determines the size of the slots and consequently the size of the shelf items, it is important to design it with the card objects that will be placed in the slots in mind. For example, cards which contain card items such as ports and LEDs must be placed in slots big enough to host them. In addition, even when using cards without ports and LEDs, small shelves with tiny slots tend to overlap alarm representations and labels, which is confusing for the end user.

The positioning point of a shelf is given by the `PositionAttribute` attribute (from class `IlTObject`); it is based on the `IlPPoint` object and defines its top left corner.

The slots of a shelf are accessed through a pair of XY indices defining a column (X index) and a row (Y index). The top left slot is assigned the indices (0,0). The X index increases horizontally from left to right and the Y index increases vertically from top to bottom. For linear shelves, the Y index is handled automatically. Objects are placed on a shelf based on X and Y slot indices plus an X and Y span, which defines how much the object will expand over its neighboring slots. An X span of 2.0, for instance, determines that the object will fully occupy the slot on its right.

Shelf class

Shelves are predefined business objects of the class `IltShelf` that are top-level containers in a hierarchy of predefined business objects used to model telecommunication equipment.

The `IltShelf` class defines the following attributes:

- ◆ **Type**—Specifies the category of a shelf. There is one predefined type of shelf, but you can define new types.

Name: `type`

Value class: `IltShelf.Type`

Attribute: `IltShelf.TypeAttribute`

- ◆ **Direction**—Specifies the direction of a shelf. The possible values are `IlpDirection.Right`, `IlpDirection.Left`, `IlpDirection.Top` or `IlpDirection.Bottom`. The default value is `Right`, and corresponds to a shelf with slot numbers at the bottom. The shelf direction affects the slot numbering (ascending or descending), but not the slot number position, which is given by the `setSlotNumsOnTop(boolean)` method of the `IltShelf` class.

Name: `direction`

Value class: `IlpDirection`

Attribute: `IltShelf.DirectionAttribute`

- ◆ **Slot sizes**—Specifies the width and height of each column and row of slots.

Name: `slotSizes`

Value class: `IlpSlotSizes`

Attribute: `IltShelf.SlotSizesAttribute`

- ◆ **XSlotIndex**—Specifies the initial number displayed to count the slots.

Name: `xslotindex`

Value class: `java.lang.Integer`

Attribute: `IltShelf.XSlotIndexAttribute`

Loading a shelf defined in XML

All you have to do is create a data source using the data source default implementation defined by `IltDefaultDataSource` and pass the XML file to the `parse` method of the data source, as shown below.

```
IlpDataSource datasource = new IltDefaultDataSource();
datasource.parse("ShelfXMLFile.xml");
```

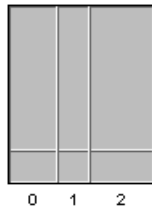
For detailed information about data sources, see *Data sources*.

How to define a shelf in XML

The following is an example of a shelf defined in XML format. For details about the XML elements used in this example, see *Elements in an XML data file*.

```
<cplData>
<addObject id="Shelf">
  <class>ilog.tgo.model.IltShelf</class>
  <attribute name="name">Shelf</attribute>
  <attribute name="slotSizes" javaClass="ilog.cpl.equipment.IlpSlotSizes">
    <width>
      <value>30</value>
      <value>20</value>
      <value>40</value>
    </width>
    <height>
      <value>90</value>
      <value>20</value>
    </height>
  </attribute>
  <attribute name="position" javaClass="ilog.cpl.graphic.IlpPoint">
    <x>100</x> <y>50</y>
  </attribute>
</addObject>
</cplData>
```

The result looks like this:



An array shelf defined in an XML file

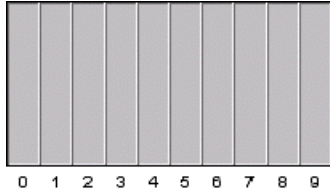
Creating a shelf with the API

All you have to do is create a shelf using the class `IltShelf` and add it to a data source, as shown in the following example.

How to create a linear shelf with the API

```
IltShelf s1 = new IltShelf(10, //Number of slots in the shelf
    20, //Width of all the slots in the shelf
    100, //Height of all the slots in the shelf
    0); //Value of the first slot number
IlpDataSource dataSource = new IltDefaultDataSource();
dataSource.addObject(s1);
```

The result looks like this:



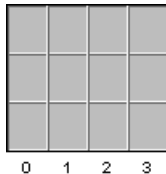
A linear shelf without cards

How to create an array shelf with the API

The following code creates an empty 4x3 array shelf:

```
IltShelf s1 = new IltShelf(4, //Number of slots along the x axis
    25, //Width of a slot on the x axis
    3, //Number of slots along the y axis
    30, //Width of a slot on the y axis
    0); //Value of the first slot number
s1.setAttributeValue(IltShelf.PositionAttribute, new IlpPoint(50, 50));
IlpDataSource dataSource = new IltDefaultDataSource();
dataSource.addObject(s1);
```

The result looks like this:



An empty 4x3 array shelf with fixed-width slots

Shelf items

Describes the following shelf items: cards, empty slots, and card carriers.

In this section

Cards

Describes the facilities available for cards.

Empty slots

Describes the facilities available for empty slots.

Card carriers

Describes the facilities available for card carriers.

Cards

Describes the facilities available for cards.

In this section

Overview of cards

Provides details about cards, like positioning, size, decorations.

Card class

Describes the attributes of the `IltCard` class.

Loading a card defined in XML

Shows how to load a card from an XML file in a data source.

Creating a card with the API

Shows how to create a card using the API and add it to the data source.

Overview of cards

Like any `IlTShelfItem` implementation, a card occupies a slot within a shelf. It is related to the shelf in the same way as any of the other shelf items described in this section. It is highly recommended to plan ahead the size of the shelves and the slot widths, based on the size of the cards and card carriers that they will host.

When a card is rotated, either manually by a developer or automatically by the system (following the rotation of a parent shelf, for instance), all its child card items are rotated accordingly.

Cards can represent alarms and states graphically according to their associated state object, in the same way as network elements can, through alarm balloons and colors corresponding to the alarm severity. It is possible to define a customized background image for them.

A card is represented in the form of a rectangle occupying the whole slot area; it bears a label, as well as status icons.

You can span a card over the neighboring slot by defining a span greater than 1 when positioning it into the shelf. The positioning of objects on a shelf is defined by an object of class `IlpShelfItemPosition` and defines the X and Y slots plus the X and Y spans.

When an object spans over other objects on the shelf, these objects are removed from the shelf.

It is important to note that the slot index is not related to the slot numbering. The slot index is used internally by the class `IlTShelf` to manage slots and cannot be changed; the slot numbering is defined by the user when creating a shelf and can be customized.

You should be careful when changing the orientation of a card, as it affects the positioning of decorations such as label, status icons, and alarm balloon. The position of the decorations around a card can be customized through properties.

Card class

Cards are predefined business objects of the class `IltCard` and are the most widely used shelf items. They can contain card item objects such as LEDs (`IltLed` class) and ports (`IltPort` class).

The `IltCard` class defines the following attributes:

- ◆ `Type`—Specifies the category of a card. It is possible to define new types of card.

Name: `type`

Value class: `IltCard.Type`

Attribute: `IltCard.TypeAttribute`

- ◆ `Direction`—Specifies the direction of the card within the shelf. By default, the orientation of a card is set to `Top`, which defines a card with a label placed vertically, status icons placed at the bottom of the card, and the alarm balloon placed at the top of the card.

Name: `direction`

Value class: `IlpDirection`

Attribute: `IltCard.DirectionAttribute`

Loading a card defined in XML

For detailed information about data sources, see *Data sources*.

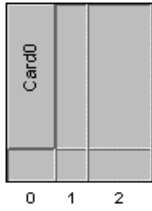
How to add a card to a shelf defined in XML

The following example extends the example presented in *Loading a shelf defined in XML* and adds a card to the shelf. Note the tags `<parent>` and `</parent>`, which define the parent object of the card. Note also that the positioning of an object in a shelf is given by an `IlpShelfItemPosition`, which defines its slot indices and span.

For details about the XML elements used in this example, see *Elements in an XML data file*.

```
<cplData>
<addObject id="Shelf">
  <class>ilog.tgo.model.IltShelf</class>
  <attribute name="name">Shelf</attribute>
  <attribute name="slotSizes" javaClass="ilog.cpl.equipment.IlpslotSizes">
    <width>
      <value>30</value>
      <value>20</value>
      <value>40</value>
    </width>
    <height>
      <value>90</value>
      <value>20</value>
    </height>
  </attribute>
  <attribute name="position" javaClass="ilog.cpl.graphic.Ilppoint">
    <x>100</x> <y>50</y>
  </attribute>
</addObject>
<addObject id="Card0">
  <class>ilog.tgo.model.IltCard</class>
  <parent>Shelf</parent>
  <attribute name="name">Card0</attribute>
  <attribute name="position"
    javaClass="ilog.cpl.graphic.views.IlpsHelfItemPosition">
    <x>0</x> <y>0</y> <width>1</width> <height>1</height>
  </attribute>
</addObject>
</cplData>
```

The result looks like this:



An array shelf with a card from an XML file

Creating a card with the API

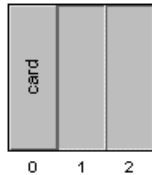
How to create a shelf with a card through the API

```
IlpDataSource dataSource = new IltDefaultDataSource();
// create shelf, set its position and add to datasources
IltShelf s1 = new IltShelf(3, 30, 90, 0);
s1.setAttributeValue(IltShelf.PositionAttribute, new IlpPoint(20, 50));
dataSource.addObject(s1);

// create card, set its position (relative to the shelf slots) and
// add to datasources
IltCard c1 = new IltCard(null, "card");
c1.setAttributeValue(IltCard.PositionAttribute,
                    new IlpShelfItemPosition(0, 0, 1, 1));
dataSource.addObject(c1);

// set parent-child relationship
dataSource.setParent(c1.getIdentifier(), s1.getIdentifier());
```

The result looks like this:



A card in a shelf

How to associate cards with an array shelf by spanning slots

The following code illustrates cards associated with an array shelf through the use of slot spanning.

```
// create datasource
IltDefaultDataSource dataSource = new IltDefaultDataSource();

// Create shelf
IltShelf s1 = new IltShelf(5, 20, 4, 25, 0);
s1.setAttributeValue(IltShelf.PositionAttribute,
                    new IlpPoint(50, 50));
dataSource.addObject(s1);

// Create cards
IltCard c1 = new IltCard(null, "c1");
c1.setAttributeValue(IltCard.PositionAttribute,
```

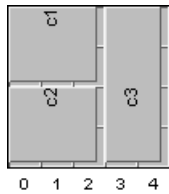
```

    new IlpShelfItemPosition(0, 0, 2.8f, 1.9f));
dataSource.addObject(c1);
IltCard c2 = new IltCard(null, "c2");
c2.setAttributeValue(IltCard.PositionAttribute,
    new IlpShelfItemPosition(0, 2, 2.8f, 1.9f));
dataSource.addObject(c2);
IltCard c3 = new IltCard(null, "c3");
c3.setAttributeValue(IltCard.PositionAttribute,
    new IlpShelfItemPosition(3, 0, 1.8f, 3.9f));
dataSource.addObject(c3);

// create relationship
dataSource.setParent(c1.getIdentifier(), s1.getIdentifier());
dataSource.setParent(c2.getIdentifier(), s1.getIdentifier());
dataSource.setParent(c3.getIdentifier(), s1.getIdentifier());

```

The result looks like this:



An array shelf with spanned cards

Empty slots

Describes the facilities available for empty slots.

In this section

Overview of empty slots

Provides details about the use of empty slots.

Empty slot class

Describes the attributes of the `IltEmptySlot` class.

Loading an empty slot defined in XML

Shows how to load an empty slot from an XML file in a data source.

Creating an empty slot with the API

Shows how to create an empty slot using the API and add it to the data source.

Overview of empty slots

An empty slot object can be placed within a free slot in the same way as a card. Graphically, `ItEmptySlot` objects only differ from free slots by a dark grey label.

Empty slots can represent alarms and states graphically; however, their base representation cannot be changed. In other words, a state that would induce a change in the graphic representation of the object base is ignored.

You should associate an empty slot with a free slot if you want to manage this free slot in some way. Unlike an empty slot, a regular free slot is not considered a business object and cannot represent alarms or states by itself.

Empty slot class

Empty slots are predefined business objects of the class `IltEmptySlot`, which extends the class `IltCard`. `IltEmptySlot`, does not define any attribute, but inherits the `TypeAttribute` attribute from its parent class and defines itself as a specific type of card.

Loading an empty slot defined in XML

Loading an empty slot from an XML file is similar to loading a card, except for the business object class defined between the `<class>` and `</class>` tags:

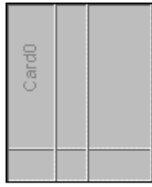
```
<class>ilog.tgo.model.IltEmptySlot</class>
```

How to load an empty slot defined in XML

The following sample uses the same XML file as in *Loading a card defined in XML*, to create an empty slot. For details about the XML elements used in this example, see *Elements in an XML data file*.

```
<cplData>
<addObject id="Shelf">
  <class>ilog.tgo.model.IltShelf</class>
  <attribute name="name">Shelf</attribute>
  <attribute name="slotSizes" javaClass="ilog.cpl.equipment.IlpSlotSizes">
    <width>
      <value>30</value>
      <value>20</value>
      <value>40</value>
    </width>
    <height>
      <value>90</value>
      <value>20</value>
    </height>
  </attribute>
  <attribute name="position" javaClass="ilog.cpl.graphic.IlpPoint">
    <x>100</x> <y>50</y>
  </attribute>
</addObject>
<addObject id="Card0">
  <class>ilog.tgo.model.IltEmptySlot</class>
  <parent>Shelf</parent>
  <attribute name="name">Card0</attribute>
  <attribute name="position"
    javaClass="ilog.cpl.graphic.views.IlpShelfItemPosition">
    <x>0</x> <y>0</y> <width>1</width> <height>1</height>
  </attribute>
</addObject>
</cplData>
```

The result looks like this:



0 1 2

An empty slot in a shelf

Creating an empty slot with the API

As the class `IltEmptySlot` extends the class `IltCard`, the API to create an empty slot instance in a shelf is similar to the API for creating a card.

How to create an empty slot with the API

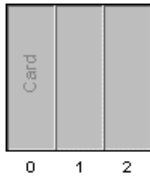
The following sample updates the sample in *Creating a card with the API*.

```
IlpDataSource dataSource = new IltDefaultDataSource();
// create shelf, set its position and add to datasources
IltShelf s1 = new IltShelf(3, 30, 90, 0);
s1.setAttributeValue(IltShelf.PositionAttribute, new IlpPoint(20, 50));
dataSource.addObject(s1);

// create empty slot, set its position (relative to the shelf slots) and
// add to datasources
IltEmptySlot c1 = new IltEmptySlot(null, "card");
c1.setAttributeValue(IltEmptySlot.PositionAttribute,
    new IlpShelfItemPosition(0, 0, 1, 1));
dataSource.addObject(c1);

// set parent-child relationship
dataSource.setParent(c1.getIdentifier(), s1.getIdentifier());
```

The result looks like this:



An Empty Slot in a Shelf

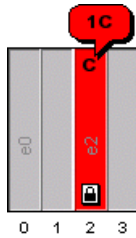
How to create a shelf with two empty slots

The following code creates a shelf with two empty slots, one showing an alarm.

```
IltDefaultDataSource dataSource = new IltDefaultDataSource();
IltShelf s1 = new IltShelf(4, 20, 100, 0);
s1.setAttributeValue(IltShelf.PositionAttribute, new IlpPoint(50, 50));
dataSource.addObject(s1);
IltEmptySlot e0 = new IltEmptySlot(null, "e0");
e0.setAttributeValue(IltEmptySlot.PositionAttribute,
    new IlpShelfItemPosition(0, 0, 1, 1));
dataSource.addObject(e0);
IltEmptySlot e2 = new IltEmptySlot(new IltOSIObjectState(), "e2");
```

```
e2.setState(IltOSI.State.Administrative.Locked);
e2.getAlarmState().setNewAlarmCount(IltAlarm.Severity.Critical, 1);
e2.setAttributeValue(IltEmptySlot.PositionAttribute,
    new IlpShelfItemPosition(2, 0, 1, 1));
dataSource.addObject(e2);
dataSource.setParent(e0.getIdentifier(), s1.getIdentifier());
dataSource.setParent(e2.getIdentifier(), s1.getIdentifier());
```

The result looks like this:



A shelf with two empty slots, one of them showing an alarm

Card carriers

Describes the facilities available for card carriers.

In this section

Overview of card carriers

Provides details about card carriers.

Card carrier class

Describes the attributes of the `IltCardCarrier` class.

Loading a card carrier defined in XML

Shows how to load a card carrier from an XML file in a data source.

Creating a card carrier with the API

Shows how to create a card carrier using the API and add it to the data source.

Overview of card carriers

Like a card, a card carrier can represent alarms and states graphically, with alarm balloons and colors defined by alarm severity. A card carrier is represented graphically as a rectangle with two distinct areas: a fixed-size utility area to represent alarm states and decorations and another area to represent equal-sized slots. (A card carrier does not support variable-width slots or slots in an array, but it is possible to set child objects with span.) The `setBottomSpacing` method allows you to customize the size of the utility area; the default size value is 30.

You can set a label for a card carrier, but it will only be visible in the logical and tiny representations, not in the symbolic representation.

The utility area can be distinguished from the cards of the card carrier through a more marked 3D effect.

Shelf items are positioned within card carriers according to a single index. The index 0 denotes the slot at the opposite end of the utility area.

Like any other shelf item, you can set the direction of a card carrier. The default setting is `Top`. When the direction is set to `Right`, for example, the utility area is displayed on the left and the slots on the right.

Card carriers can appear very cramped when they concentrate lots of information in a small space. Therefore, it is important to design the right size of object to be able to accommodate all the graphic representations, such as secondary state icons, alarm balloons, and so forth.

Card carrier class

Card carrier objects are predefined business objects of the class `IltCardCarrier` which is also a container for shelf item objects. They allow you to associate more than one card with a single slot.

The `IltCardCarrier` class defines the following attributes:

- ◆ **Type**—Specifies the category of a card carrier. There is one predefined type of card carrier, but it is possible to define other types.
Name: `type`
Value class: `IltCardCarrier.Type`
Attribute: `IltCardCarrier.TypeAttribute`
- ◆ **Slot Count**—Specifies the number of slots for the card carrier
Name: `slotCount`
Value class: `Integer`
Attribute: `IltCardCarrier.SlotCountAttribute`
- ◆ **Direction**—Specifies the direction of a card carrier. The possible values are `IlpDirection.Right`, `IlpDirection.Left`, `IlpDirection.Top` or `IlpDirection.Bottom`. The default value is `Top`. The card carrier direction affects the decorations that are attached to the object, such as label, alarm count and secondary states.
Name: `direction`
Value class: `IlpDirection`
Attribute: `IltCardCarrier.DirectionAttribute`
- ◆ **Bottom Spacing**—Specifies the size of the utility area in the card carrier, where the state and alarm information is displayed. The default value is 30.
Name: `bottomSpacing`
Value class: `Integer`
Attribute: `IltCardCarrier.BottomSpacingAttribute`

Loading a card carrier defined in XML

For detailed information about data sources, see *Data sources*.

How to load a card carrier defined in XML

The following example creates a card carrier with a card and adds it to a shelf. Note the tags `<parent>` and `</parent>`, which define the parent object of the card items, and the attribute `slotCount`, which defines the number of slots for the card carrier.

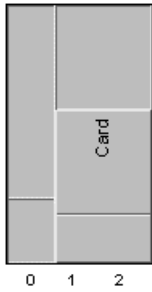
Note: The positioning of an object in a card carrier is given by an `IlpPoint`, not by an `IlpShelfItemPosition` as in the shelf.

For details about the XML elements used in this example, see *Elements in an XML data file*.

```
<cplData>
<addObject id="Shelf">
  <class>ilog.tgo.model.IltShelf</class>
  <attribute name="name">Shelf</attribute>
  <attribute name="slotSizes" javaClass="ilog.cpl.equipment.IlpSlotSizes">
    <width>
      <value>30</value>
      <value>20</value>
      <value>40</value>
    </width>
    <height>
      <value>90</value>
      <value>20</value>
    </height>
  </attribute>
  <attribute name="position" javaClass="ilog.cpl.graphic.IlpPoint">
    <x>100</x> <y>50</y>
  </attribute>
</addObject>
<addObject id="Carrier">
  <class>ilog.tgo.model.IltCardCarrier</class>
  <parent>Shelf</parent>
  <attribute name="name">Carrier</attribute>
  <attribute name="slotCount" javaClass="java.lang.Integer">2</attribute>
  <attribute name="position"
    javaClass="ilog.cpl.graphic.views.IlpShelfItemPosition">
    <x>1</x> <y>0</y> <width>2</width> <height>2</height>
  </attribute>
</addObject>
<addObject id="Card">
  <class>ilog.tgo.model.IltCard</class>
  <parent>Carrier</parent>
  <attribute name="name">Card</attribute>
</addObject>
</cplData>
```

```
<attribute name="position" javaClass="ilog.cpl.graphic.IlpPoint">  
  <x>0</x> <y>1</y>  
</attribute>  
</addObject>  
</cplData>
```

The result looks like this:



An array shelf with a card carrier and a card

Creating a card carrier with the API

When creating a card carrier from the API, you must specify the number of slots it uses. This value is set in the constructor.

How to create a card carrier with the API

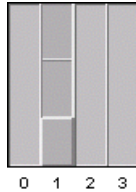
The following example shows how to create an empty card carrier with two free slots.

```
IltDefaultDataSource dataSource = new IltDefaultDataSource();
IltShelf s1 = new IltShelf(4, 20, 100, 0);
s1.setAttributeValue(IltShelf.PositionAttribute, new IlpPoint(50, 50));
dataSource.addObject(s1);

IltCardCarrier ccl = new IltCardCarrier(null, 2); // number of slots
ccl.setAttributeValue(IltCardCarrier.PositionAttribute,
    new IlpShelfItemPosition(1, 0, 1, 1));
dataSource.addObject(ccl);

dataSource.setParent(ccl.getIdentifier(), s1.getIdentifier());
```

The result looks like this:



Shelf with a card carrier containing two free slots

How to create a shelf with two card carriers

The following code creates a shelf with two card carriers.

```
// Create a datasource
IltDefaultDataSource dataSource = new IltDefaultDataSource();

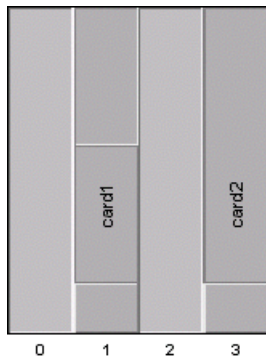
// create shelf, set its position and add to datasource
IltShelf s1 = new IltShelf(4, 20, 200, 0);
s1.setAttributeValue(IltShelf.PositionAttribute, new IlpPoint(50, 50));
dataSource.addObject(s1);

// create card carrier 1, set its position (relative to s1)
// and add to datasource
IltCardCarrier ccl = new IltCardCarrier(null, 2);
ccl.setAttributeValue(IltCardCarrier.PositionAttribute,
```

```
    IlpShelfItemPosition(1, 0, 1, 1));  
dataSource.addObject(cc1);
```

```
// create card carrier 2, set its position (relative to s1)  
// and add to datasource  
IltCardCarrier cc2 = new IltCardCarrier(null, 1);  
cc2.setAttributeValue(IltCardCarrier.PositionAttribute,  
    new IlpShelfItemPosition(3, 0, 1, 1));  
dataSource.addObject(cc2);  
  
// create card 1, set its position (relative to cc1) and add  
// to datasource  
// note that the position is instance of IlpPoint, not  
// IlpShelfItemPosition (card carriers require just one slot index and span)  
IltCard c1 = new IltCard(null, "card1");  
c1.setAttributeValue(IltCard.PositionAttribute, new IlpPoint(1, 1));  
dataSource.addObject(c1);  
  
// create card 2, set its position (relative to cc2) and add  
// to datasource  
IltCard c2 = new IltCard(null, "card2");  
c2.setAttributeValue(IltCard.PositionAttribute, new IlpPoint(0, 1));  
dataSource.addObject(c2);  
  
// set relationship  
dataSource.setParent(c1.getIdentifier(), cc1.getIdentifier());  
dataSource.setParent(c2.getIdentifier(), cc2.getIdentifier());  
dataSource.setParent(cc1.getIdentifier(), s1.getIdentifier());  
dataSource.setParent(cc2.getIdentifier(), s1.getIdentifier());
```

The result looks like this:



Shelf with two card carriers

In this figure, the carrier of *card1* has a free slot also. The second card carrier is fully occupied by *card2* only.

Card items

Describes the following card items: LEDs and ports.

In this section

Overview of card items

Provides details about the positioning of card items.

Card item class

Describes the attributes of the `IltCardItem` class.

LEDs

Describes the facilities available for LEDs.

Ports

Describes the facilities available for ports.

Overview of card items

Card items are placed inside a card item container (`Il+Card`). Their positioning is relative to the top left corner of the container. Regardless of the orientation of the container, the top left corner constitutes the origin for positioning the card items; the X coordinate of card items increases from left to right and the Y coordinate increases from top to bottom, with regard to this origin.

The positioning point of a card item is its center point and is relative to the origin of the card item container (top left corner).

When planning the size of shelves and slots, remember to take into account the child objects, particularly the card and its card items.

IBM® ILOG® JViews TGO defines two types of card items: LEDs and ports.

Card item class

The abstract class `IltCardItem` defines all the common characteristics of a business object that can be associated with a card or any implementation of `IltCard`.

The `IltCardItem` class defines the following attributes:

- ◆ `Type`— Specifies the category of a card item. There are different predefined types of card items in JViews TGO; all of them are described in this section.

Name: `type`

Value class: `IltCardItem.Type`

Attribute: `IltCardItem.TypeAttribute`

- ◆ `Direction`— Specifies the direction of the card item.

Name: `direction`

Value class: `IlpDirection`

Attribute: `IltCardItem.DirectionAttribute`

LEDs

Describes the facilities available for LEDs.

In this section

Overview of LEDs

Provides details about the use of LEDs.

LED class

Describes the attributes of the `IltLed` class.

Loading an LED defined in XML

Shows how to load an LED from an XML file in a data source.

Creating an LED with the API

Shows how to create an LED using the API and how to add it to the data source.

Predefined LED types

Lists the different types of LED and their graphic representation.

Overview of LEDs

An LED (Light Emitting Diode) is an object used to represent a state through a color. Most types of equipment use LEDs as interfaces that give the user information on hardware and software conditions. IBM® ILOG® JViews TGO provides an LED business object to help you create real world items of equipment.

Although a state object can be associated with LEDs, it does not affect their graphic representation. LEDs are not managed objects themselves; they help represent the state of managed objects such as ports or cards.

Depending on the nature of the application, LEDs can be represented by a bitmap image or by a vector graphic. Like a network element, the LED representation is set through the `TypeAttribute` attribute assigned to the object represented by the LED.

To find all necessary information on the styling properties of LEDs, refer to Customizing shelves and cards.

LED class

LEDs are predefined business objects of the class `IltLed`, which defines a card item by extending the `IltCardItem` abstract class.

The parent class `IltCardItem` defines the `TypeAttribute`, which is used to create several different graphic representations for different instances of the same `IltLed` class.

The `IltLed` class defines the following attributes:

- ◆ `Width`—Specifies the width of the LED in its vector graphic representation

Name: `width`

Value class: `java.lang.Float`

Attribute: `IltLed.WidthAttribute`

- ◆ `Height`—Specifies the height of the LED in its vector graphic representation

Name: `height`

Value class: `java.lang.Float`

Attribute: `IltLed.HeightAttribute`

Loading an LED defined in XML

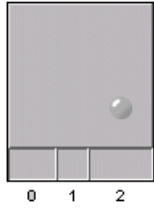
For detailed information about data sources, see *Data sources*.

How to add an LED to a card defined in XML

The following example extends the example in *Loading a shelf defined in XML* and adds an LED to a card. For details about the XML elements used in this example, see *Elements in an XML data file*.

```
<cplData>
<addObject id="Shelf">
  <class>ilog.tgo.model.IltShelf</class>
  <attribute name="name">Shelf</attribute>
  <attribute name="slotSizes" javaClass="ilog.cpl.equipment.IlpSlotSizes">
    <width>
      <value>30</value>
      <value>20</value>
      <value>40</value>
    </width>
    <height>
      <value>90</value>
      <value>20</value>
    </height>
  </attribute>
  <attribute name="position" javaClass="ilog.cpl.graphic.IlpPoint">
    <x>100</x> <y>50</y>
  </attribute>
</addObject>
<addObject id="Card">
  <class>ilog.tgo.model.IltCard</class>
  <parent>Shelf</parent>
  <attribute name="name"></attribute>
  <attribute name="position"
    javaClass="ilog.cpl.graphic.views.IlpShelfItemPosition">
    <x>0</x> <y>0</y> <width>3</width> <height>1</height>
  </attribute>
</addObject>
<addObject id="Led">
  <class>ilog.tgo.model.IltLed</class>
  <parent>Card</parent>
  <attribute name="name">Led</attribute>
  <attribute name="type">Circular</attribute>
  <attribute name="position" javaClass="ilog.cpl.graphic.IlpRelativePoint">
    <x>70</x> <y>65</y>
  </attribute>
</addObject>
</cplData>
```

The result looks like this:



An array shelf with a card and an LED

Creating an LED with the API

When you create an LED with the API, you must provide its type, which can be set through the constructor or through the `setAttributeValue` method.

How to create an LED with the API

The following sample shows how to create an LED of type `Circular`.

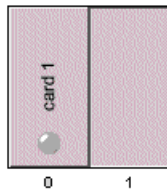
```
List objects = new ArrayList();
// create shelf identified as myShelf
IltShelf s1 = new IltShelf("myShelf");
s1.setSlotSizes(2,50,1,100);
objects.add(s1);

// create card
IltCard c1 = new IltCard(new IltOSIObjState(), "card 1");
c1.setPosition(new IlpShelfItemPosition(0, 0, 1, 1));
objects.add(c1);

// create card item
IltLed l1 = new IltLed("myLed");
l1.setType(IltLed.Type.Circular);
l1.setPosition(new IlpRelativePoint(25,85));
objects.add(l1);

// add all objects to data source
datasource.addObjects(objects);
```

The result looks like this:

























A shelf containing a card with an LED

Predefined LED types

There are several predefined LED types, using both bitmap and vector representations. You can extend the predefined types and define new representations using either images or drawers capable of creating vector images. For information on how to create new LED types, see Customizing LED types.

LED types and their graphic representation

LED Type	Graphic Representation	
Circular		
CircularShape		
CircularFlat		
Rectangular		
RectangularShape		
HardDisk A		
HardDisk B		
HardDiskPwr A		
HardDiskPwr B		
Power A		
Power B		

Among the LEDs listed in this table, `IltLed.Type.Circular`, `IltLed.Type.CircularShape`, `IltLed.Type.CircularFlat`, `IltLed.Type.Rectangular`, and `IltLed.Type.RectangularShape` are vector graphic representations; all the others are bitmap images.

Ports

Describes the facilities available for ports.

In this section

Overview of ports

Provides details about the use of ports.

Port class

Describes the attributes of the `IltPort` class.

Loading a port defined in XML

Shows how to load a port from an XML file in a data source.

Creating a port with the API

Shows how to create a port using the API and how to add it to the data source.

Predefined port types

Lists the different types of port and their graphic representation.

Overview of ports

Most real world cards have a physical interface to connect them to other sets of equipment or to a network. The connections are usually achieved through connectors or ports. IBM® ILOG® JViews TGO provides port objects to represent connections between a card (`ItCard`) and the external world.

Ports allow you to represent alarms and states graphically in the same way as network elements do, with alarm balloons and colors depending on the alarm severity.

Depending on the nature of the application, ports can be represented by a bitmap image or by a vector graphic. Like a network element, the port representation is set through the `TypeAttribute` attribute assigned to the object represented by the port.

To find all necessary information on the styling properties of ports, refer to Customizing shelves and cards.

Port class

Ports are predefined business objects of the class `IltPort`, which defines a card item by extending the `IltCardItem` abstract class to fit inside card objects.

The parent class `IltCardItem` defines the `TypeAttribute`, which is used to create different graphic representations for different instances of the same `IltLed` class.

The `IltPort` class does not define any specific attribute:

Loading a port defined in XML

For detailed information about data sources, see *Data sources*.

How to add a port to a card defined in XML

The following example extends the example in *Loading a shelf defined in XML* and adds a port to a card. For details about the XML elements used in this example, see *Elements in an XML data file*.

```
<cplData>
<addObject id="Shelf">
  <class>ilog.tgo.model.IltShelf</class>
  <attribute name="name">Shelf</attribute>
  <attribute name="slotSizes" javaClass="ilog.cpl.equipment.IlpSlotSizes">
    <width>
      <value>30</value>
      <value>20</value>
      <value>40</value>
    </width>
    <height>
      <value>90</value>
      <value>20</value>
    </height>
  </attribute>
  <attribute name="position" javaClass="ilog.cpl.graphic.IlpPoint">
    <x>100</x> <y>50</y>
  </attribute>
</addObject>
<addObject id="Card">
  <class>ilog.tgo.model.IltCard</class>
  <parent>Shelf</parent>
  <attribute name="name"></attribute>
  <attribute name="position"
    javaClass="ilog.cpl.graphic.views.IlpShelfItemPosition">
    <x>0</x> <y>0</y> <width>3</width> <height>1</height>
  </attribute>
</addObject>
<addObject id="Port">
  <class>ilog.tgo.model.IltPort</class>
  <parent>Card</parent>
  <attribute name="name">Port</attribute>
  <attribute name="type">DB15_f</attribute>
  <attribute name="position" javaClass="ilog.cpl.graphic.IlpRelativePoint">
    <x>30</x> <y>65</y>
  </attribute>
</addObject>
</cplData>
```

The result looks like this:



An array shelf with a card and a port

Creating a port with the API

When you create a port with the API, you must provide its type, which can be set through the constructor or through the `setAttributeValue` method.

How to create a port with the API

The following sample shows how to create a port of type `Centronics_36f`.

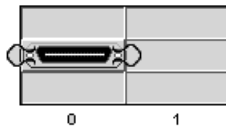
```
// Create a shelf, set its position and add to datasource
IltShelf s1 = new IltShelf(2, 65, 3, 20, 0);
s1.setAttributeValue(IltShelf.PositionAttribute, new IlpPoint(20, 50));
dataSource.addObject(s1);

// create a card, set its position (relative to s1) and
// add to datasource
IltCard c1 = new IltCard();
c1.setAttributeValue(IltCard.PositionAttribute,
    new IlpShelfItemPosition(0, 1, 1, 1));
dataSource.addObject(c1);

// create shelf item, set its position (relative to c1) and
// add it to datasource
IltPort port = new IltPort("port", IltPort.Type.Centronics_36f, null);
port.setAttributeValue(IltPort.PositionAttribute,
    new IlpRelativePoint(33, 10));
dataSource.addObject(port);

// set relationship
dataSource.setParent(port.getIdentifier(), c1.getIdentifier());
dataSource.setParent(c1.getIdentifier(), s1.getIdentifier());
```

The result looks like this:



A shelf containing a card with a port


















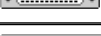









Predefined port types


























There are several predefined port types using both vector and bitmap representations. You can also define new types using either images or drawers capable of creating vector images.























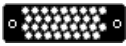

For information on how to create new port types, see [Customizing port types](#).

All port types in the following table are image ports.

Port types and their graphic representation

Port Type	Graphic Representation
BNC_f	
BNC_m	
CardEdge_34f	
Centronics_36f	
Centronics_36m	
Centronics_50f	
Centronics_50m	
Centronics_HP_36f	
Centronics_HP_36m	
Centronics_HP_50f	
Centronics_HP_50m	
Centronics_VHD_68f	
Centronics_VHD_68m	
Composed_13W3_f	
Composed_13W3_m	
DB_15f	
DB_15m	
DB_25f	
DB_25m	
DB_37f	
DB_37m	
DB_50f	
DB_50m	
DB_9f	
DB_9m	
DB_HD_15f	
DB_HD_15m	

Port Type	Graphic Representation
DB_HP_50f	
DB_HP_50m	
DB_HP_68f	
DB_HP_68m	
DIN_4f	
DIN_4m	
DIN_5f	
DIN_5m	
DIN_6f	
DIN_6m	
DIN_8f	
DIN_8m	
ExternalPower_f	
ExternalPower_m	
FDD_Power_f	
FDD_Power_m	
HDD_Power_f	
HDD_Power_m	
HoodedPower_f	
HoodedPower_m	
IDC_34f	
IDC_34m	
IDC_40f	
IDC_40m	
IDC_50f	

Port Type	Graphic Representation
IDC_50m	
IDC_68	
IEEE_1394_4f	
IEEE_1394_4m	
IEEE_1394_6f	
IEEE_1394_6m	
LFH_60f	
LFH_60m	
RJ45_f	
RJ45_m	
SC_Fiber_f	
SC_Fiber_m	
SCA_80f	
SCA_80m	
ST_Fiber_f	
ST_Fiber_m	
TwoProngPower_f	
TwoProngPower_m	
USB_A_f	
USB_A_m	
USB_B_f	
USB_B_m	
V35_f	
V35_m	











Representation of shelves and cards in a table and in a tree

Representing objects in a table

Objects of the `IltShelf` class are represented in a table as follows.

Object	Name	New Alarms	Alarms	Primary State	Secondary States
	BTS Shelf				

Objects of classes `IltCard` and `IltEmptySlot` are represented in a table as follows.

Object	Name	New Alarms	Alarms	Primary State	Secondary States
	CompositeCard12				
	CompositeCard6				
	CompositeAdapter1				
	Empty				
	CompositeCard7			Operational:Enabled; Usa...	Availability:Not Installed
	CompositeCard5				
	CompositeCard3				
	Empty				
	CompositeCard0	2 m	2 m	Bellcore State:Enabled Idle	
	CompositeCard9				

Objects of the class `IltPort` are represented in a table as follows.

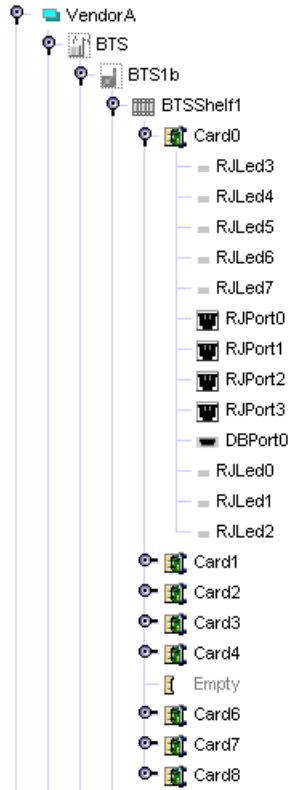
Object	Name	Type	New Alarms	Alarms	Primary State	Secondary States
	RJ45_f	RJ 45f				
	RJ45_f	RJ 45f			Operational:Enabled; U...	Availability:Not Installed
	RJ45_f	RJ 45f				
		Centronics...				
		Centronics...			Operational:Enabled; U...	Availability:Not Installed
		Centronics...				
	RJ45_f	RJ 45f			Operational:Enabled; U...	Availability:Not Installed

Objects of the class `IltLed` are represented in a table as follows.

Object	Name	Type	New Alarms	Alarms	Primary State	Secondary States
	Circular	Circular				
	Rectangular	Rectangular				
	Rectangular	Rectangular				
	Rectangular	Rectangular				
	Rectangular	Rectangular				
	Power_B	Power				

Representing objects in a tree

Objects of the classes `IltShelf`, `IltCard`, `IltPort`, and `IltLed` are represented in a tree as follows.



Customizing the representation of shelves and cards

For information on how to customize the graphic representation of shelves and cards, refer to Customizing port types.

BTS (Base Transceiver Station)

Explains how to use IBM® ILOG® JViews TGO predefined business objects of type BTS in your applications.

In this section

BTS Class

Describes the `IltBTS` class and its components.

Loading a BTS object defined in XML

Shows how to load a BTS object from an XML file into a data source.

Creating a BTS object with the API

Explains how to create an `IltBTS` object using the JViews TGO API and how to add it to a data source.

Representation of BTS objects in a table and in a tree

Shows how BTS objects are represented in a table and in a tree.

BTS Class

A BTS is a predefined business object of the class `IltBTS` that you can directly insert in a JViews TGO data source to represent graphically base transceiver stations in any of the graphic components connected to the data source. For a general introduction to predefined business classes, see *Introducing business objects and data sources*.

The `IltBTS` class does not define any specific attribute.

You can retrieve the class `IltBTS` using its `GetIlpClass()` method. You can handle its instances as simple `IlpObject` instances and set and get its attributes with the generic methods `getAttributeValue(java.lang.String)` and `setAttributeValue(ilog.cpl.model.IlpAttribute, java.lang.Object)`.

An `IltBTS` is made up of two types of object:

- ◆ `IltBTSAntenna` objects, which represent cellular transmitting and receiving antennas.
- ◆ An `IltNetworkElement` object, which represents the BTS item of electronic equipment carrying the antennas. For details, see *Network elements*.

The antennas and the item of electronic equipment constitute the detail objects of the `IltBTS` container object.

Antennas

An `IltBTS` object enables you to represent the antenna coverage of each cell in a cellular system. A cell corresponds to a limited geographic zone.

An antenna is an instance of the class `IltBTSAntenna`.

The `IltBTSAntenna` class defines the following attributes:

- ◆ **Power**—Indicates the power of the antenna in watts
Name: `power`
Value class: `java.lang.Integer`
Attribute: `IltBTSAntenna.PowerAttribute`
- ◆ **Beam direction**—Indicates the beam direction of the antenna in degrees
Name: `beamDirection`
Value class: `java.lang.Integer`
Attribute: `IltBTSAntenna.BeamDirectionAttribute`
- ◆ **Beam width**—Indicates the beam width of the antenna, in degrees
Name: `beamWidth`
Value class: `java.lang.Integer`
Attribute: `IltBTSAntenna.BeamWidthAttribute`

The graphic representation of an antenna includes:

- ◆ An arrow line. The direction of the line corresponds to the beam direction and its length is proportional to the power of the antenna.
- ◆ An arc. The span of the arc corresponds to the beam width and the radius of the arc is proportional to the power of the antenna.

Note: For a given power value, the length of the arrow line and the radius of the arc may differ. This is to allow for more flexibility in the graphic representation.

You can choose to display both the arrow line and the arc, or only one of them.

BTS equipment

A base transceiver station can be made up of an item of electronic equipment carrying the cellular antennas. An item of BTS equipment is an instance of an `IltnetworkElement` of type `IltnetworkElement.Type.BTSEquipment`. For details on network elements, see *Network elements*.

Loading a BTS object defined in XML

All you have to do is create a data source using the default data source implementation defined by `IltDefaultDataSource` and pass the XML file to the `parse` method of the data source as shown below:

```
dataSource = new IltDefaultDataSource();
dataSource.parse("BTSXMLFile.xml");
```

For detailed information about data sources, see *Data sources*.

How to define a BTS object in XML

The following is an example of a BTS object defined in XML format. For details about the XML elements used in this example, see *Elements in an XML data file*.

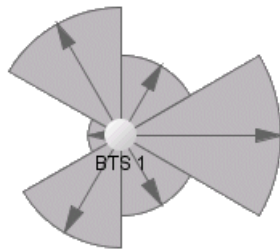
```
<cplData>
<addObject id="bts1" container="true">
  <class>ilog.tgo.model.IltBTS</class>
  <attribute name="name">BTS 1</attribute>
  <attribute name="position" javaClass="ilog.cpl.graphic.IlpPoint">
    <x>200</x>
    <y>200</y>
  </attribute>
</addObject>
<addObject id="BTSEquipment">
  <class>ilog.tgo.model.IltNetworkElement</class>
  <parent>bts1</parent>
  <attribute name="name">BTSEquipment</attribute>
  <attribute name="type">BTSEquipment</attribute>
</addObject>
<addObject id="antenna1">
  <class>ilog.tgo.model.IltBTSAntenna</class>
  <parent>bts1</parent>
  <attribute name="name">BTS Antenna 1</attribute>
  <attribute name="beamDirection">0</attribute>
  <attribute name="power">100</attribute>
  <attribute name="beamWidth">60</attribute>
</addObject>
<addObject id="antenna2">
  <class>ilog.tgo.model.IltBTSAntenna</class>
  <parent>bts1</parent>
  <attribute name="name">BTS Antenna 2</attribute>
  <attribute name="beamDirection">60</attribute>
  <attribute name="power">50</attribute>
  <attribute name="beamWidth">60</attribute>
</addObject>
<addObject id="antenna3">
  <class>ilog.tgo.model.IltBTSAntenna</class>
  <parent>bts1</parent>
```

```

<attribute name="name">BTS Antenna 3</attribute>
<attribute name="beamDirection">120</attribute>
<attribute name="power">80</attribute>
<attribute name="beamWidth">60</attribute>
</addObject>
<addObject id="antenna4">
  <class>ilog.tgo.model.IltBTSAntenna</class>
  <parent>bts1</parent>
  <attribute name="name">BTS Antenna 4</attribute>
  <attribute name="beamDirection">180</attribute>
  <attribute name="power">20</attribute>
  <attribute name="beamWidth">60</attribute>
</addObject>
<addObject id="antenna5">
  <class>ilog.tgo.model.IltBTSAntenna</class>
  <parent>bts1</parent>
  <attribute name="name">BTS Antenna 5</attribute>
  <attribute name="beamDirection">240</attribute>
  <attribute name="power">70</attribute>
  <attribute name="beamWidth">60</attribute>
</addObject>
<addObject id="antenna6">
  <class>ilog.tgo.model.IltBTSAntenna</class>
  <parent>bts1</parent>
  <attribute name="name">BTS Antenna 6</attribute>
  <attribute name="beamDirection">300</attribute>
  <attribute name="power">50</attribute>
  <attribute name="beamWidth">60</attribute>
</addObject>
</cplData>

```

The result looks like this:



A BTS as defined in XML

Creating a BTS object with the API

How to create an `IltBTS` object

To create an `IltBTS` object and add it to a data source:

- ◆ Initialize JViews TGO.

```
IltSystem.Init() ;
```

- ◆ Instantiate one or more antennas.

```
IltBTSAntenna antenna1 = new IltBTSAntenna("A1", //label
                                           new IltBellcoreObjectState(), //state
                                           0, //beam direction
                                           100, //power
                                           60); //beam width
IltBTSAntenna antenna2 = new IltBTSAntenna("A2", new
                                           IltBellcoreObjectState(), 60, 50, 60);
IltBTSAntenna antenna3 = new IltBTSAntenna("A3", new
                                           IltBellcoreObjectState(), 120, 80, 60);
IltBTSAntenna antenna4 = new IltBTSAntenna("A4", new
                                           IltBellcoreObjectState(), 180, 20, 60);
IltBTSAntenna antenna5 = new IltBTSAntenna("A5", new
                                           IltBellcoreObjectState(), 240, 70, 60);
IltBTSAntenna antenna6 = new IltBTSAntenna("A6", new
                                           IltBellcoreObjectState(), 300, 50, 60);
```

- ◆ Instantiate an item of BTS equipment (optional).

```
IltNetworkElement btsEquipment = new IltNetworkElement("bts
equipment", IltNetworkElement.Type.BTSEquipment, new
IltBellcoreObjectState());
```

- ◆ Instantiate the BTS container.

```
IltBTS bts = new IltBTS("BTS", null, null);
bts.setPosition(new IlpPoint(500, 200));
```

- ◆ Create a data source for `IltObject` instances and define the parent-child relationships.

```
IltDataSource dataSource = new IltDefaultDataSource();
dataSource.setParent(btsEquipment, bts);
dataSource.setParent(antenna1, bts);
dataSource.setParent(antenna2, bts);
dataSource.setParent(antenna3, bts);
dataSource.setParent(antenna4, bts);
```

```
dataSource.setParent (antenna5,bts);  
dataSource.setParent (antenna6,bts);
```

◆ Add the BTS to the data source.

```
dataSource.addObject (bts);  
dataSource.addObject (btsEquipment);  
dataSource.addObject (antenna1);  
dataSource.addObject (antenna2);  
dataSource.addObject (antenna3);  
dataSource.addObject (antenna4);  
dataSource.addObject (antenna5);  
dataSource.addObject (antenna6);
```

Representation of BTS objects in a table and in a tree

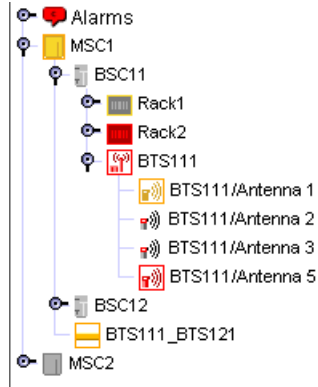
Objects of the `Il+BTS` class are represented in a table as follows:

Object	Name	New Alarms	Alarms	Primary State	Secondary States
	BTS121	1 C+	1 C*	Operational:Enabled; Us...	
	BTS111	1 M	1 M*	Operational:Enabled; Us...	Availability:Power Off, Control:Res...

Objects of the `Il+BTSAntenna` class are represented in a table as follows:

Object	Name	Power	Width	Direction	New Alarms	Alarms	Primary State	Secondary States
	BTS111/Antenna 3	80	60	120			Operational:Enabled; Us...	
	BTS121/Antenna 2	50	60	60			Operational:Enabled; Us...	
	BTS111/Antenna 5	70	60	240	1 C+	1 C*	Operational:Enabled; Us...	
	BTS121/Antenna 1	100	60	0	1 w	1 m*	Operational:Enabled; Us...	Procedural:Reporting; Repair:Und...
	BTS111/Antenna 1	100	60	0	1 w	1 m*	Operational:Enabled; Us...	Procedural:Reporting; Repair:Und...
	BTS121/Antenna 3	80	60	120			Operational:Enabled; Us...	
	BTS111/Antenna 2	50	60	60			Operational:Enabled; Us...	

Objects of classes `Il+BTS` and `Il+BTSAntenna` are represented (with other predefined business objects) in a tree as follows:



Customizing the representation of BTS

For information on how to customize the graphic representation of BTS, refer to Customizing BTS.

Off-page connectors

Explains how to use predefined business objects of type off-page connector in your applications.

In this section

Off-page connector class

Describes the `IlOffPageConnector` class and its attributes.

Loading an off-page connector defined in XML

Shows how to load an off-page connector from an XML file into a data source.

Creating an off-page connector with the API

Shows how to create an off-page connector through the API and how to add it to a data source.

Representation of off-page connectors in a network

Shows the graphical representation of the different types of off-page connectors.

Representation of off-page connectors in a table and in a tree

Shows how off-page connectors are represented in a table and in a tree.

Off-page connector class

Off-page connectors can be inserted in a network to replace nodes. They are used to indicate that the link continues in a network part that is outside of the current view.

An off-page connector can be associated with information used to:

- ◆ display the corresponding view,
- ◆ indicate visually on which neighbor view the object represented by the off-page connector is located.

Off-page connectors are predefined business objects of the class `IltOffPageConnector`. For a general introduction to predefined business classes, see *Introducing business objects and data sources*.

You can retrieve the class `IltOffPageConnector` using its `GetIltClass()` method.

The `IltOffPageConnector` class defines the following attribute:

- ◆ `Type`—Specifies the category of off-page connector. It is possible to define new types of off-page connectors.

Name: `type`

Value class: `IltOffPageConnector.Type`

Attribute: `IltOffPageConnector.TypeAttribute`

Loading an off-page connector defined in XML

All you have to do is create a data source using the default data source implementation defined by `IltDefaultDataSource` and pass the XML file to the `parse` method of the data source, as follows:

```
dataSource = new IltDefaultDataSource();
dataSource.parse("OffPageConnectorXMLFile.xml");
```

For detailed information about data sources, see *Data sources*.

How to define an off-page connector in XML

The following is an example of an off-page connector defined in XML format. For details about the XML elements used in this example, see *Elements in an XML data file*.

```
<addObject id="Region B">
  <class>ilog.tgo.model.IltOffPageConnector</class>
  <attribute name="name">Region B</attribute>
  <attribute name="position" javaClass="ilog.cpl.graphic.IlpPoint">
    <x>480</x> <y>200</y>
  </attribute>
  <attribute name="objectState"
    javaClass="ilog.tgo.model.IltOSIObjectState">
  </attribute>
  <alarms>
    <new severity="Warning">4</new>
  </alarms>
</addObject>
```

An off-page connector only displays alarm states to the exclusion of any other states such as primary states, secondary states, or statuses.

The following figure shows an off-page connector displayed in a network component:



Off-page connector displayed in a network component

Creating an off-page connector with the API

The following example shows how to create an off-page connector and add it to a network component.








How to create an off-page connector with the API

```
IltDefaultDataSource dataSource = new IltDefaultDataSource();
IltOffPageConnector opc = new IltOffPageConnection("Region B");
IlpPoint center = new IlpPoint(190, 190);
opc.setPosition(center);
dataSource.addObject(opc);}
```

Representation of off-page connectors in a network

Off-page connectors (OPC) can have different graphic representations according to the value of their attribute `type`. By default, IBM® ILOG® JViews TGO provides the following OPC types and representations:

OPC types and their representation

Type	Representation	Description
Standard		Standard off-page connector
Managed		Generic managed entity
SingleManaged		Single entity currently managed by the system
MultipleManaged		Multiple entities currently managed by the system
Unmanaged		Generic entities not managed by the system
SingleUnmanaged		Single entity currently not managed by the system
MultipleUnmanaged		Multiple entities currently not managed by the system

The off-page connector type can be set to the object through XML or through the API. Refer to the class `ilog.tgo.model. IltOffPageConnector` for more information.

You can customize the graphic representation of an off-page connector type by registering a new base renderer (see `ilog.tgo.graphic.renderer. IltOPCBaseRenderer`). This base renderer is registered in the JViews TGO default settings (see `ilog.tgo.resource. IltSettings.SetValue(java.lang.Object, java.lang.Object)`). Besides the predefined base renderers, you can also register base renderers based on images or on any `IlvGraphic` class, including SVG support.

For details about how to create new off-page connector types, refer to Customizing new off-page connector types.

Customizing the representation of off-page connectors

For information on how to customize the graphic representation of off-page connectors, refer to Customizing off-page connectors.

Representation of off-page connectors in a table and in a tree

Like network elements, ports and LEDs, off-page connectors are represented in the tree and table components with a reduced version of the type representation.

Objects of the `IltOffPageConnector` class are represented in a table as follows:

Object	Name	New Al...	Alarms	Primary State	Secondary States
◆	Managed				
⊞	MultipleUnmanaged				
⊞	Unmanaged				
◆	SingleManaged				
◆	Standard				
⊞	SingleUnmanaged				

Objects of the `IltOffPageConnector` class are represented in a tree as follows:



Customizing the representation of off-page connectors

For information on how to customize the graphic representation of off-page connectors, refer to [Customizing off-page connectors](#).

Alarms

Explains how to use predefined business objects of type alarm in your applications.

In this section

Alarm object class

Describes the IltAlarm class and its attributes.

Loading an alarm defined in XML

Shows how to load an alarm from an XML file into a data source.

Creating an alarm with the API

Shows how to create an alarm using the JViews TGO API and add it to a data source.

Representation of alarms in a network

Explains that alarms cannot be represented in a network.

Representation of alarms in a table and in a tree

Shows how alarms are represented in a table and in a tree.

Alarm object class

Alarms are predefined business objects of the class `ilog.tgo.model.IltAlarm`, that you can directly insert in an IBM® ILOG® JViews TGO data source and represent graphically in a table or tree component connected to the data source.

Unlike the other predefined business objects, alarms are not managed objects but they are closely related to them, since they reflect alarm conditions affecting managed objects.

The `IltAlarm` class defines the following attributes:

Attributes of IltAlarm

Attribute Name	Value Class	Description
<code>ackSystemId</code>	<code>String</code>	Identifier of the system used by the author of the last modification of the acknowledged state.
<code>ackTime</code>	<code>java.util.Date</code>	Date of the last modification of the acknowledged state.
<code>ackUserId</code>	<code>String</code>	Identifier of the author of the last modification of the acknowledged state.
<code>additionalText</code>	<code>String</code>	Additional information about the alarm.
<code>alarmAckState</code>	<code>Boolean</code>	The acknowledged state of the alarm.
<code>alarmChangedTime</code>	<code>java.util.Date</code>	Date of the last modification of alarm attribute values.
<code>alarmClearedTime</code>	<code>java.util.Date</code>	Date when the perceived severity of the alarm was changed to Cleared.
<code>alarmRaisedTime</code>	<code>java.util.Date</code>	Date when the alarm was raised.
<code>alarmType</code>	<code>IltAlarm.AlarmType</code>	The type of alarm.
<code>attributesChanges</code>	<code>Object</code>	The list of changed attributes.
<code>backedUpStatus</code>	<code>Boolean</code>	The backup status. Indicates whether the managed object has a backup object.
<code>backUpObject</code>	<code>String</code>	Distinguished name of the backup object.
<code>clearSystemId</code>	<code>String</code>	Identifier of the system used by the author of the last request to clear the alarm.
<code>clearUserId</code>	<code>String</code>	Identifier of the author of the last request to clear the alarm.
<code>comments</code>	<code>Object</code>	List of comments.
<code>correlatedNotifications</code>	<code>Object</code>	The correlated notifications. This attribute identifies a set of notifications to which this notification is considered to be correlated.
<code>graphicRepresentation</code>		Fake attribute used to make it possible to display the alarm object in one column of a

Attribute Name	Value Class	Description
		table. This attribute cannot be set a value directly.
managedObjectClass	String	Managed object class of the managed object instance in which the alarm occurred.
managedObjectInstance	String	The managed object instance in which the alarm occurred. See <i>Setting the alarm counters</i> .
monitoredAttributes	Object	The attributes that are monitored.
notificationId	String	Identifier of the notification that carries the alarm information.
perceivedSeverity	IltAlarmSeverity	The perceived severity of the alarm. It indicates the relative level of urgency for operator attention. Values are defined in <code>IltAlarm.Severity</code> for raw alarms, and in <code>IltAlarm.ImpactSeverity</code> for impact alarms. The perceived severity is used to determine the color of the alarm icon in table and tree components.
probableCause	IltAlarm. ProbableCause	The probable cause of the alarm.
proposedRepairActions	String	The proposed repair actions.
specificProblem	String	The specific problem. Provides more information on the alarm than <code>probableCause</code> .
systemDN	String	The distinguished name of the system that detected the network event and generated the notification
thresholdInfo	Object	The information about the threshold.
trendIndication	IltAlarm. TrendIndication	The trend indication. This attribute indicates whether the observed condition is getting better, worse, or is unchanged.

Note: The attributes with the Java™ class Object may be populated with objects of any type.

You can retrieve the class `IltAlarm` using its `GetIltClass()` method. You can handle its instances as simple `GetIltClass()` instances and set and get its attributes with the generic methods `getAttributeValue(java.lang.String)` and `setAttributeValue(ilog.cpl.model.IltAttribute, java.lang.Object)`.

The class `IltAlarm` also provides convenience methods, such as `getPerceivedSeverity()` and `setPerceivedSeverity(ilog.tgo.model.IltAlarmSeverity)`, that you can use directly to access each individual predefined attribute of the class.

The perceived severity of raw alarms is of type `IltAlarmSeverity`, whereas the perceived severity of impact alarms is of type `IltAlarm`, `IltAlarm.ImpactSeverity`. `JViews TGO` provides the following predefined severities that are statically allocated and stored in static data members of `IltAlarm`:

For raw alarms, the available severity values are the following:

- ◆ `IltAlarm.Severity.Critical`
- ◆ `Major`
- ◆ `Minor`
- ◆ `Warning`
- ◆ `Unknown`
- ◆ `Cleared`

For impact alarms, the available severity values are the following:

- ◆ `IltAlarm.ImpactSeverity.CriticalHigh`
- ◆ `IltAlarm.ImpactSeverity.CriticalLow`
- ◆ `IltAlarm.ImpactSeverity.MajorHigh`
- ◆ `IltAlarm.ImpactSeverity.MajorLow`
- ◆ `IltAlarm.ImpactSeverity.MinorHigh`
- ◆ `IltAlarm.ImpactSeverity.MinorLow`
- ◆ `IltAlarm.ImpactSeverity.WarningHigh`
- ◆ `IltAlarm.ImpactSeverity.WarningLow`
- ◆ `IltAlarm.ImpactSeverity.Unknown`
- ◆ `IltAlarm.ImpactSeverity.Cleared`

You can define other severities to extend the default alarm model. For details, refer to [Customizing alarm severities in the *Styling* documentation](#).

Loading an alarm defined in XML

All you have to do is create a data source using the data source default implementation defined by `IltDefaultDataSource` and pass the XML file to be read to its `parse` method, as follows:

```
dataSource = new IltDefaultDataSource();
dataSource.parse("AlarmXMLFile.xml");
```

For detailed information about data sources, see section *Data sources*.







How to define an alarm in XML

The following is an example of an alarm defined in XML format. For details about the XML elements used in this example, see table *Elements in an XML data file*.

In this example, the first alarm object (alarm 1) is an acknowledged raw alarm with a perceived severity level of `Warning`, and affecting the managed object `Router1`. The second alarm object (alarm 2) is a nonacknowledged impact alarm with a perceived severity level of `MajorHigh`, and affecting the managed object `Gateway1`.

```
<cplData>
  <addObject id="alarm 1">
    <class>ilog.tgo.model.IltAlarm</class>
    <attribute name="notificationId">alarm 1</attribute>
    <attribute name="alarmAckState">true</attribute>
    <attribute name="ackSystemId">leipzig</attribute>
    <attribute name="ackUserId">leibniz</attribute>
    <attribute name="ackTime">Mon, 05 Jan 2004 13:33:25 GMT+0430</attribute>
    <attribute name="alarmRaisedTime">Mon, 05 Jan 2004 13:30:12 GMT+0430</
attribute>
    <attribute name="managedObjectInstance"
javaClass="java.lang.String">Router1</attribute>
    <attribute name="alarmType"></attribute>
    <attribute name="perceivedSeverity">Raw.Warning</attribute>
    <attribute name="probableCause">0</attribute>
  </addObject>
  <addObject id="alarm 2">
    <class>ilog.tgo.model.IltAlarm</class>
    <attribute name="notificationId">alarm 2</attribute>
    <attribute name="alarmAckState">>false</attribute>
    <attribute name="alarmRaisedTime">Mon, 05 Jan 2004 13:54:52 GMT+0430</
attribute>
    <attribute name="managedObjectInstance"
javaClass="java.lang.String">Gateway1</attribute>
    <attribute name="perceivedSeverity">Impact.MajorHigh</attribute>
    <attribute name="probableCause">303</attribute>
  </addObject>
</cplData>
```

The following figure shows the two alarms displayed in a table component:

Alarm	Notification	Severity	Ack	Date Raised	Managed Object Instance	Probable Cause
	alarm 1	Warning	✓	06:00:12 GMT-03:00		Indeterminate
	alarm 2	Minor	✓	06:24:52 GMT-03:00		Bandwidth reduction
	alarm 3	Cleared		06:32:28 GMT-03:00		Excessive bit error rate
	alarm 4	Minor!!	✓	08:48:02 GMT-03:00		Indeterminate
	alarm 5	Cleared	✓	09:07:05 GMT-03:00		Unavailable
	alarm 6	Minor	✓	09:09:22 GMT-03:00		Excessive bit error rate

Alarms Displayed in a Table Component

The attribute `managedObjectInstance` may be of any Java™ class. To benefit from the automatic consolidation of alarm states from individual alarms, use the same value as the object identifier of the corresponding managed object. See *Setting the alarm counters*.

To set a value in XML, specify the Java class of the value. For well-known classes, use:

```
<attribute name="comments" javaClass="java.lang.String">comment</attribute>.
```

For other specific classes, the XML format will be the same but the classes must conform to the JViews TGO type converter constraints.

For details about the well-known classes and the type converter, refer to Type converter in the *Context and Deployment Descriptor* documentation.

Creating an alarm with the API

All you have to do is create a new alarm using the class `IltAlarm` and add it to a data source, as follows:

How to create an alarm through the API

```
IltAlarm alarm = new IltAlarm("alarm 1");
alarm.setAttributeValue(IltAlarm.PerceivedSeverityAttribute, IltAlarm.Severity.
W
arning);
alarm.setAttributeValue(IltAlarm.AlarmAckStateAttribute, Boolean.FALSE);
alarm.setAttributeValue(IltAlarm.ProbableCauseAttribute,
    IltAlarm.ProbableCause.ExcessiveBitErrorRate);
alarm.setAttributeValue(IltAlarm.ManagedObjectInstanceAttribute, new
String("Router1"));
alarm.setAttributeValue(IltAlarm.AlarmRaisedTimeAttribute, new Date());

IltDataSource dataSource = new IltDefaultDataSource();
dataSource.addObject(alarm);
```

Representation of alarms in a network












Objects of the `IlAlarm` class have no representation in a network. Alarms are instead represented aggregated in alarm states. See *Alarm states* for more details.

Note: By default, business objects of the class `IlAlarm` are filtered out by the network and equipment adapters. This behavior is controlled by the property `excludedClasses` of classes `ilog.cpl.network.IlpNetworkAdapter` and `ilog.cpl.equipment.IlpEquipmentAdapter`.

Representation of alarms in a table and in a tree

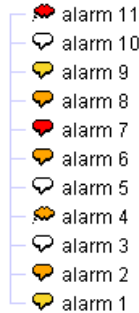
In the table component, each alarm object is represented as a row, with columns corresponding to attributes of the alarm object. The `graphicalRepresentation` attribute displays raw alarms with a tiny round rectangle alarm balloon, and impact alarms with a tiny alarm cloud. See the following figure.

Objects of the class `IltAlarm` are represented in a table as follows:

Alarm	Notification	Severity	Ackn...	Date Raised	Managed Object Instance	Probable Cause
 alarm 1		Warning	✓	10:00:12 CET	Router1	Indeterminate
 alarm 2		Minor	✓	10:24:52 CET	Gateway1	Bandwidth reduction
 alarm 3		Cleared		10:32:28 CET	Link1	Excessive bit error rate
 alarm 4		Minor High	✓	12:48:02 CET	Router2	Indeterminate
 alarm 5		Cleared	✓	13:07:05 CET	Link1	Unavailable
 alarm 6		Minor	✓	13:09:22 CET	Gateway2	Excessive bit error rate
 alarm 7		Major		15:10:21 CET	Gateway1	Congestion
 alarm 8		Minor		15:11:09 CET	Gateway1	Excessive bit error rate
 alarm 9		Warning		15:44:59 CET	Link2	Line card problem
 alarm 10		Cleared		15:56:32 CET	Router2	Commercial power failure
 alarm 11		Major Low		15:57:07 CET	Router1	Indeterminate

In the tree component, each alarm object is represented as a node. The graphical representation is the same as in the table component. See the following figure.

Objects of the class `IltAlarm` are represented in a tree as follows:



Customizing the representation of alarms

For information on how to customize the graphic representation of alarms, see [Customizing alarms](#).

Lookup tables for state visuals

Provides tables showing the default graphical representations of the dictionaries of states available in IBM® ILOG® JViews TGO.

In this section

The OSI state dictionary visuals

Describes the eight primary states of a telecom object and the five groups of secondary states in the OSI state dictionary.

The Bellcore state dictionary visuals

Describes the three primary states and various secondary states of the Bellcore state dictionary.

The SNMP state dictionary visuals

Describes the five primary states and 40 secondary states of the SNMP state dictionary.

The Misc state dictionary visuals

Describes the secondary states of the Misc state dictionary, which are used to complement those of the other state dictionaries.

The Performance state dictionary visuals

Describes the secondary states of the Performance state dictionary.

The SAN state dictionary visuals

Describes the secondary states of the SAN state dictionary.

The SONET state dictionary visuals

Describes the six primary states and six secondary states of the SONET state dictionary.

The OSI state dictionary visuals

Describes the eight primary states of a telecom object and the five groups of secondary states in the OSI state dictionary.

In this section

Graphical representation of the OSI primary states

Illustrates the graphical representation of the eight legal OSI primary states.

Graphical representation of OSI secondary states

Provides the graphical representations of OSI secondary states.






Graphical representation of the OSI primary states





These states are represented graphically in three different ways:

- ◆ The appearance of the object base changes (hatched around the perimeter, flat base, or base in relief).
- ◆ An icon appears in the top left corner of the object base.
- ◆ Both changes occur.

The table illustrates these changes on a node element. The same graphical representations are used for links, groups, and cards. Note that the primary state of empty slots is not represented graphically.

Graphical representation of the eight valid OSI primary states

OSI State Value	Primary State	Visual	Icon Properties (IltSettings)	Comment
Operational: Disabled Usage: Idle Administrative: Unlocked	OSS			The resource is not available or depends upon another source that is not available.
Operational: Disabled Usage: Idle Administrative: Locked	OSS		OSI.State.Administrative.Locked.Icon	The resource is not available and is administratively prohibited from performing user services.
Operational: Enabled Usage: Idle Administrative: Unlocked	NT			The resource is available for use and has the capacity to accept services from another source.
Operational: Enabled Usage: Idle Administrative: Locked	NT		OSI.State.Administrative.Locked.Icon	The resource is available but is administratively prohibited from performing user services.
Operational: Enabled Usage: Active	CT			The resource is available for use and has the capacity to

OSI State Value	Primary State	Visual	Icon Properties (IltSettings)	Comment
Administrative: Unlocked				accept services from another source.
Operational: Enabled Usage: Active Administrative: Shutting down	CT		OSI.State.Administrative.Locked.Icon	The resource is administratively permitted to existing users only; it is shedding traffic.
Operational: Enabled Usage: Busy Administrative: Unlocked	CT		OSI.State.Usage.Busy.Icon	The resource is in use with no spare capacity.
Operational: Enabled Usage: Busy Administrative: Shutting down	CT		OSI.State.Usage.Busy.Icon OSI.State.Administrative.Locked.Icon	The resource is in use with no spare capacity; it is shedding traffic.
Other combinations				The resource is in an indeterminate state

Graphical representation of OSI secondary states

Secondary states are almost always represented graphically by adding an icon to the top left corner of the object base element. The only exception is for the “Not Installed” Availability state, which is denoted by a change in the base element visual (as shown in table *Other OSI secondary state representations*).

How to read the When Applicable column

The meaningful representation of the OSI secondary states depends on the eight valid OSI primary states. These primary states are grouped in three categories: *Out Of Service (OOS)*, *In Service and Carrying Traffic (CT)*, and *In Service and Carrying No Traffic (NT)*.

- ◆ Out Of Service (OOS):
 - Operational: Disabled, Usage: Idle, Administrative: Unlocked
 - Operational: Disabled, Usage: Idle, Administrative: Locked
- ◆ In Service, Carrying No Traffic (NT):
 - Operational: Enabled, Usage: Idle, Administrative: Unlocked
 - Operational: Enabled, Usage: Idle, Administrative: Locked
- ◆ In Service, Carrying Traffic (CT):
 - Operational: Enabled, Usage: Active, Administrative: Unlocked
 - Operational: Enabled, Usage: Active, Administrative: Shutting down
 - Operational: Enabled, Usage: Busy, Administrative: Unlocked
 - Operational: Enabled, Usage: Busy, Administrative: Shutting down

Secondary state names

In *Icon-based representations of OSI secondary states*, the symbolic name Secondary State Definition that appears after each Secondary State Name corresponds to the static secondary state definition. For example, the In Test secondary state is defined by `InTest`, which corresponds to the static definition `InTest`.

Secondary state icons













It is possible to change the icon associated with a secondary state by using global settings, see Using global settings. The icon property name to be used with `IltSettings.SetValue()` must include the secondary state group, the secondary state definition, and the primary state. For example:






















- ◆ `OSI.Repair.UnderRepair.OOS.Icon`
where:






- **Repair** is one of the five OSI secondary state groups
- **UnderRepair** is the secondary state definition of the “Under Repair” secondary state (the only secondary state in the Repair group)
- **OOS** is the corresponding OSI primary state group (Out Of Service)

For more information on how to use global settings to modify the OSI secondary state icons, see Customizing the OSI state system.


Icon-based representations of OSI secondary states

Secondary State Name Secondary State Definition	When Applicable: OOS, NT, CT			Comment
Procedural Secondary State				
Initialization Required InitializationRequired				Resource requires initialization before it can be made available.
Initializing Initializing				Resource is being initialized.
Reporting Reporting				Resource is initialized and test results are being returned.
Terminating Terminating				Resource is terminating.
Availability Secondary State				
Degraded Degraded				Service is degraded. This could adversely affect the usage state.
Dependency Dependency				The resource cannot operate because some other resource of which it depends (i.e. a resource not represented by the same managed object) is unavailable. For example, a device is not accessible because its controller is powered off. The operational state is Disabled.
Failed Failed				Resource is subject to a fault that prevents it from being used. In most cases, this secondary state is coupled with an alarm,

Secondary State Name Secondary State Definition	When Applicable: OOS, NT, CT			Comment
				an outstanding alarm, or a loss of connectivity.
In Test InTest				Resource is undergoing test.
Log Full LogFull				Log is full. Log service has been made unavailable.
Not Installed NotInstalled	See Table A.3			Resource is not installed.
Off Duty OffDuty				Service has been made unavailable because of an ongoing time schedule.
Off Line OffLine				The resource requires a routine operation to be performed to place it online and make it available for use. The operation may be manual or automatic, or both. The operational state is Disabled.
Power Off PowerOff				Resource requires power, but is not powered. Most often, this resource is coupled with an alarm, an outstanding alarm, or a loss of connectivity.
Control Secondary State				
Part of Services Locked PartOfServicesLocked				This value indicates whether a manager has administratively restricted a particular part of a service from the user(s) of a resource. The administrative state is Unlocked. Examples are: incoming service barred, outgoing service barred, write locked by media, read locked.
Reserved for Test ReservedForTest				Resource is reserved for test.
Subject to Test SubjectToTest				Resource is currently under test.
Suspended Suspended				The service has been administratively suspended to users of the resource. The resource may retain knowledge of the current users and/or request for usage, depending on the managed object class definition, but it does not resume performing services until the suspended

Secondary State Name Secondary State Definition	When Applicable: OOS, NT, CT			Comment
				condition is revoked. The administrative state is Unlocked.
Standby Secondary State				
Cold Standby ColdStandby				The backup resource is not providing service and cannot immediately take over the role of the primary resource.
Hot Standby HotStandby				The backup resource is not providing service, but can immediately take over the role of the primary resource.
Providing Service ProvidingService				The backup resource has been put into service. (It currently takes over the role of a primary resource.)
Warm Standby WarmStandby				The backup resource is not providing service, but can immediately or within a short delay take over the role of the primary resource. (Data is mirrored to the backup resource at regular intervals.)
Repair Secondary State				
Includes Under repair only. Outstanding alarm secondary states are considered as alarm representation cases and are represented as such.				
Under Repair UnderRepair				Resource is currently under repair.

Other OSI secondary state representations

Secondary State Name Secondary State Definition	Visual (OOS only)	Comment
Availability Secondary State		
Not Installed NotInstalled		Resource is not installed, is installed improperly, or is incompletely installed.

All other OSI primary state combinations indicate that the object is in an indeterminate state. In this case, the OSI secondary states are not applicable to the object.

The Bellcore state dictionary visuals

Describes the three primary states and various secondary states of the Bellcore state dictionary.

In this section

Graphical representation of the Bellcore primary states

Illustrates the graphical representation of the three Bellcore primary states.

Graphical representation of the Bellcore secondary states




Provides the graphical representations of Bellcore secondary states.

Graphical representation of the Bellcore primary states

These graphical representations are based on the base element states introduced in *Introducing business objects and data sources*. The same graphical representations are used for links, groups, and cards. Note that the primary state of empty slots is not represented graphically.

These states are represented graphically by a change in the appearance of the object base (hatched around the perimeter, flat base, or base in relief).

Graphical representation of Bellcore primary states

Bellcore State Value	Visual
Disabled/Idle	
Enabled/Idle	
Enabled/Active	

Graphical representation of the Bellcore secondary states

Secondary states are almost always represented graphically by adding an icon to the top left corner of the object base element.

The only exceptions concern the “Unassigned,” “Unequipped,” and “Pre-Post Service” secondary states for which the OOS representations are denoted by a main change in the base element visual (as shown in table *Other representations of Bellcore secondary states*).

How to read the When Applicable column

The meaningful representation of the secondary states depends on the current Bellcore primary state set to the telecom object. The Bellcore primary states are the following:

- ◆ OOS—Disabled/Idle
- ◆ NT—Enabled/Idle
- ◆ CT—Enabled/Active

The “When Applicable” column in table *Icon-based representations of Bellcore secondary states* lists the graphic representation of the Bellcore secondary states for a given primary state. A blank entry indicates that the secondary state has no meaning in that particular primary state.

Secondary state names

In *Icon-based representations of Bellcore secondary states*, the symbolic name “Secondary State Definition” that appears after each “Secondary State Name” corresponds to the static secondary state definition. For example, the “Inhibit In Progress” secondary state is defined by `InhibitInProgress`, which corresponds to the static definition `InhibitInProgress`.

Secondary state icons

It is possible to change the icon associated with a secondary state by using global settings, see *Using global settings*. The icon property name to be used with `ILTSettings.SetValue()` must include the secondary state definition and the primary state. For example:























- ◆ `Bellcore.SecState.IdleTransmit.NT.Icon`























where:




















- **IdleTransmit** is the secondary state definition of the “Idle Transmit” secondary state
- **NT** is the corresponding Bellcore primary state (Enabled/Idle)



































For more information on how to use global settings to modify the Bellcore secondary state icons, see *Customizing the Bellcore state system*.




















Icon-based representations of Bellcore secondary states


Secondary State Name Secondary State Definition	When Applicable: OOS, NT, CT			Comment
Blocked Blocked				No outgoing traffic is allowed on the entity. The restriction is imposed by the far-end network element.
Busy Busy				The entity is currently in use and has no spare operating capacity for further usage demand.
Cold Standby ColdStandby				The entity is about to back up another entity and is synchronized with the backed-up entity. It requires initialization before it can take over.
Combined Combined				This load-sharing entity has assumed the load of its mate entity in addition to its own.
Diagnostic Diagnostic				Diagnostic activity that affects the service is currently being performed on the entity.
Disconnected Disconnected				Subscribed service has been disconnected, but still exists in the database with a specific intercept treatment.
Exercise Exercise				Service-affecting exercise is currently being performed on the entity.
Facility Failure FacilityFailure				The associated transport facility is OOS.
Far End Processor Outage FarEndProcessorOutage				The processor at an associated far-end network element is OOS.
Fault Fault				The entity is OOS because it is faulty.
Forced Forced				The entity has been manually forced into the In Service state from an Out Of Service state

Secondary State Name Secondary State Definition	When Applicable: OOS, NT, CT			Comment
				that occurred under the normal procedure.
Hot Standby HotStandby				The entity is to back up another entity and is synchronized with the backed-up entity. It does not require initialization before it can take over.
Idle Idle				The entity is available to provide service, but is not currently used.
Idle Receive IdleReceive				Applicable to bidirectional termination points only. The receiving direction is not cross-connected.
Idle Transmit IdleTransmit				Applicable to bidirectional termination points only. The transmission direction is not cross-connected.
Inhibit In Progress InhibitInProgress				The entity is waiting for users to terminate before transitioning to OOS.
Locked Out LockedOut				No outgoing traffic is allowed on the entity. This restriction may be imposed for maintenance reasons.
Loopback Test LoopbackTest				A loopback activity is being performed on the entity.
Maintenance Maintenance				The entity has been manually removed from service for maintenance activity.
Maintenance Limited MaintenanceLimited				Normal trouble detection function is not provided for the entity because of defects developed in the entity or in an associated entity.
Mismatch Of Equipment MismatchOfEquipment				The entity is installed with improper equipment or circuit




Secondary State Name Secondary State Definition	When Applicable: OOS, NT, CT			Comment
				pack, or the correct equipment has improper attributes.
Monitor Monitor				The entity has reached a monitored performance level considered as abnormal.
Overflow Overflow				The entity has no more storage capacity for capturing additional information. In this state, the entity is read-only.
Performance Monitor Inhibited PerformanceMonitorInhibited				The performance monitoring function of the entity has been temporarily suspended.
Power Power				The entity is abnormal or OOS because there is a defect in the power supply.
Pre-Post Service PrePostService	See <i>Other representations of Bellcore secondary states</i>			The entity has been manually removed from service for the pre/post-service administration activity.
Protection Release Inhibited ProtectionReleaseInhibited				The entity is inhibited from protection release.
Protection Switch Exercise ProtectionSwitchExercise				A protection switching exercise is currently running on the entity.
Protection Switch Inhibited ProtectionSwitchInhibited				The regular protected entity is inhibited from switching to protection.
Protocol Protocol				The entity is OOS due to a Layer-2 or higher protocol violation.
Providing Service ProvidingService				The redundant (backup, protecting) entity is currently providing service. This value is mutually exclusive of cold-standby, warm-standby, and hot-standby. It is equivalent to the Working secondary state.
Rearrangement Rearrangement				The entity has been removed from service because of

Secondary State Name Secondary State Definition	When Applicable: OOS, NT, CT			Comment
				physical and/or logical rearrangement activity.
Red Lined RedLined				The entity is a special (red-lined) circuit.
Software Downloading SoftwareDownloading				Software download activity is currently being performed on the entity.
Software Transfer Inhibited SoftwareTransferInhibited				Software transfer is inhibited on the entity.
Software Transfer Only SoftwareTransferOnly				Only software transfer is allowed on the entity.
Software Uploading SoftwareUploading				Software upload activity is currently being performed on the entity.
Standby Inhibited StandbyInhibited				The standby entity is inhibited from taking over the role of the backed-up entity.
Standby Switched StandbySwitched				Indicates the state of a protection system, for example, switched on to provide service.
Supported Entity Absent SupportedEntityAbsent				The associated support entities are absent.
Supported Entity Exists SupportedEntityExists				The entity is currently supporting other entities.
Supporting Entity Absent SupportingEntityAbsent				The associated supporting entities are absent.
Supporting Entity Outage SupportingEntityOutage				There is an outage on the supporting entity.
Supporting Entity Swapped SupportingEntitySwapped				The associated supporting entity has swapped to a spare (backup) entity.
Suspend Both SuspendBoth				Any attempt to establish connection to or from the entity is administratively

Secondary State Name Secondary State Definition	When Applicable: OOS, NT, CT			Comment
				inhibited for non-maintenance reasons.
Suspend Origination SuspendOrigination				Any attempt to establish a connection with the entity is administratively prohibited for non-maintenance reasons. The consequences are the same as disabled, which is used for maintenance.
Suspend Termination SuspendTermination				Any attempt to establish a connection with the entity is administratively prohibited for non-maintenance reasons. The consequences are the same as disabled, which is used for maintenance.
Switched System Activity SwitchedSystemActivity				The entity is removed from service due to switching system activities.
Terminated-Both Terminated-Both				When applied to a termination point, the receiving signal is terminated and an insertion word is transmitted instead of the normal signal.
Terminated-From Terminated-From				When applied to a termination point, the receiving signal is terminated.
Terminated-To Terminated-To				When applied to a termination point, an insertion word is transmitted instead of the normal signal.
Test Test				Test activity, other than loopback, diagnostic, and exercise, is currently being performed on the entity.
Test Failure TestFailure				The object is OOS due to a test failure.
Transferred Transferred				The load sharing entity has transferred its normal load responsibility to its mate entity.
Warm Standby				The entity is to back up another entity and gets

Secondary State Name Secondary State Definition	When Applicable: OOS, NT, CT			Comment
WarmStandby				synchronized with the backed up entity at regular intervals. It does not require initialization before it can take over.
Working Working				The redundant (backup, protecting) entity is currently providing service. This value is mutually exclusive of cold-standby, warm-standby, and hot-standby.

Other representations of Bellcore secondary states

Secondary State Name Secondary State Definition	Visual (OOS only)	Comment
Pre-Post Service PrePostService		The entity has been manually removed from service for the pre/post-service administration activity.
Unassigned Unassigned		The entity has not been assigned the required provisioning data. No service or maintenance is permitted in this state since the necessary data has not been assigned.
Unequipped Unequipped		The equipment entity has not been equipped with the necessary hardware or the software entity has not been loaded with the necessary data or code.

The SNMP state dictionary visuals

Describes the five primary states and 40 secondary states of the SNMP state dictionary.

In this section

Graphical representation of SNMP primary states



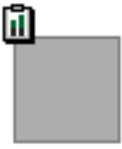
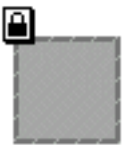
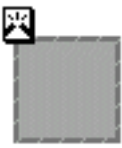
Illustrates the graphical representation of the SNMP primary states.

Graphical representation of SNMP secondary states

Illustrates the graphical representation of the SNMP secondary states.

Graphical representation of SNMP primary states

Graphical representation of SNMP primary states

SNMP State Value	Visual	Icon Properties (IltSettings)	Comment
Up			The resource is operable and available.
Down			The resource is not available.
Testing		SNMP.State.Testing.Icon	The resource is undergoing a test.
Shutdown		SNMP.State.Shutdown.Icon	The resource is not available and is administratively prohibited from performing user services.
Failed		SNMP.State.Failed.Icon	The resource is subject to a fault that prevents it from being used.

Graphical representation of SNMP secondary states

Secondary state names

In table *Graphical representation of SNMP secondary states*, the symbolic name “Secondary State Definition” that appears after each “Secondary State Name” corresponds to the static secondary state definition. For example, the “In Octets” secondary state is defined by `InOctets`, which corresponds to the static definition `InOctets`.

Secondary state icons

It is possible to change the icon associated with a secondary state by using global settings, see *Using global settings*. The icon property name to be used with `IltSettings.SetValue()` must include the secondary state group, the secondary state definition, and the decoration type. For example:




- ◆ `SNMP.Interface.InOctets.Gauge`
- ◆ `SNMP.UDP.InDatagrams.Chart`













where:











- **Interface** and **UDP** are two of the seven possible secondary state groups
- **InOctets** and **InDatagrams** are the secondary state definitions of the “In Octets” and “In Datagrams” secondary states
- **Gauge** and **Chart** are the decoration types defined by the `IltDecorationType` class












For more information on how to use global settings to modify the SNMP secondary state icons, see *Customizing the SNMP state system*.











Graphical representation of SNMP secondary states

Secondary State Name Secondary State Definition	Graphic	Graphic Frame Files	Comment
Interface Group			
In Octets InOctets		ilt_in_octets1	Total number of octets received in the interface, including framing characters.
		ilt_in_octets2	
		ilt_chart_in_octets1	
		ilt_chart_in_octets2	
In Subnetwork Unicast Packets InUcastPkts		ilt_in_ucastpkts1	The number of subnetwork unicast packets delivered to
		ilt_in_ucastpkts2	







Secondary State Name Secondary State Definition	Graphic	Graphic Frame Files	Comment
		ilt_chart_in_ ilt_chart_in_ucastpkts2	a higher-layer protocol.
In Non-Unicast Packets InNUcastPkts		ilt_in_nucastpkts1 ilt_in_nucastpkts2	The number of non-unicast packets delivered to a higher-layer protocol.
		ilt_chart_in_nucastpkts1 ilt_chart_in_nucastpkts2	
In Discards InDiscards		ilt_in_discards1 ilt_in_discards2	The number of inbound packets that were chosen to be discarded, even though no errors had been detected, to prevent their being deliverable to a higher-layer protocol.
		ilt_chart_in_discards1 ilt_chart_in_discards2	
In Errors InErrors		ilt_in_errors1 ilt_in_errors2	The number of inbound packets that contained errors, preventing them from being deliverable to a higher-layer protocol.
		ilt_chart_in_errors1 ilt_chart_in_errors2	
In Unknown Protocol InUnknownProtos		ilt_in_unknown_protos1 ilt_in_unknown_protos2	The number of packets received through the interface that were discarded because of an unknown or unsupported protocol.
		ilt_chart_in_unknown_protos1 ilt_chart_in_unknown_protos2	
Out Octets OutOctets		ilt_out_octets1 ilt_out_octets2	Total number of octets transmitted from the interface, including framing characters.
		ilt_chart_out_octets1 ilt_chart_out_octets2	
Out Unicast Packets OutUcastPkts		ilt_out_ucastpkts1 ilt_out_ucastpkts2	The total number of packets that higher-level













Secondary State Name Secondary State Definition	Graphic	Graphic Frame Files	Comment
		ilt_chart_out_ucastpkts1 ilt_chart_out_ucastpkts2	protocols requested be transmitted to a subnetwork-unicast address, including those that were discarded or not sent.
Out Non-Unicast Packets OutNUcastPkts		ilt_out_nucastpkts1 ilt_out_nucastpkts2	The total number of packets that higher-level protocols requested be transmitted to a non-unicast address, including those that were discarded or not sent.
		ilt_chart_out_nucastpkts1 ilt_chart_out_nucastpkts2	
Out Discards OutDiscards		ilt_out_discards1 ilt_out_discards2	The number of outbound packets that were chosen to be discarded even though no errors had been detected to prevent them being transmitted.
		ilt_chart_out_discards1 ilt_chart_out_discards2	
Out Errors OutErrors		ilt_out_errors1 ilt_out_errors2	The number of outbound packets that could not be transmitted because of errors.
		ilt_chart_out_errors1 ilt_chart_out_errors2	
IP Group			
In Receives InReceives		ilt_ip_inreceives1 ilt_ip_inreceives2	The total number of input datagrams received from interfaces, including those received in error.
		ilt_chart_ip_inreceives1 ilt_chart_ip_inreceives2	
In Header Errors InHdrErrors		ilt_ip_inhdrerrors1 ilt_ip_inhdrerrors2	The number of input datagrams discarded due to




Secondary State Name Secondary State Definition	Graphic	Graphic Frame Files	Comment
		ilt_chart_ip_inhdrerrors1 ilt_chart_ip_inhdrerrors2	errors in their IP headers.
In Address Errors InAddrError		ilt_ip_inaddrerrors1 ilt_ip_inaddrerrors2	The number of input datagrams discarded because the IP address in their IP header destination field was not a valid address to be received at this entity.
		ilt_chart_ip_inaddrerrors1 ilt_chart_ip_inaddrerrors2	
Forwarded Datagrams ForwDatagrams		ilt_ip_forwdatagrams1 ilt_ip_forwdatagrams2	The number of input datagrams for which this entity was not their final IP destination,.As a result, an attempt was made to find a route to forward them to that final destination.
		ilt_chart_ip_forwdatagrams1 ilt_chart_ip_forwdatagrams2	
In Unknown Protocols InUnknownProtos		ilt_ip_unknownprotos1 ilt_ip_unknownprotos2	The number of locally addressed datagrams received successfully, but discarded because of an unknown or unsupported protocol.
		ilt_chart_ip_unknownprotos1 ilt_chart_ip_unknownprotos2	
In Discards InDiscards		ilt_ip_indiscards1 ilt_ip_indiscards2	The number of input IP datagrams for which no problems were encountered to prevent them being processed, but which were discarded.
		ilt_chart_ip_indiscards1 ilt_chart_ip_indiscards2	
In Delivers InDelivers		ilt_ip_indelivers1 ilt_ip_indelivers2	The total number of input datagrams successfully delivered to IP user
		ilt_chart_ip_indelivers1	

Secondary State Name Secondary State Definition	Graphic	Graphic Frame Files	Comment
		ilt_chart_ip_indelivers2	protocols (including ICMP).
Out Requests OutRequests		ilt_ip_outrequests1 ilt_ip_outrequests2	The total number of IP datagrams that local IP user protocols (including ICMP) supplied to IP in requests for transmission.
		ilt_chart_ip_outrequests1 ilt_chart_ip_outrequests2	
Out Discards OutDiscards		ilt_ip_outdiscards1 ilt_ip_outdiscards2	The number of output IP datagrams for which no problem was encountered to prevent their transmission to their destination, but which were discarded.
		ilt_chart_ip_outdiscards1 ilt_chart_ip_outdiscards2	
Out No Routes OutNoRoutes		ilt_ip_noroutes1 ilt_ip_noroutes2	The number of IP datagrams discarded because no route could be found to transmit them to their destination.
		ilt_chart_ip_noroutes1 ilt_chart_ip_noroutes2	
Forwarding Forwarding		SNMP.IP.Forwarding.Icon (IltSettings)	The indication of whether this entity is acting as an IP gateway for forwarding datagrams received by this entity, but not addressed to it.
ICMP Group			
In Messages InMsgs		ilt_icmp_inmsgs1 ilt_icmp_inmsgs2	The total number of ICMP messages that the entity received.
		ilt_chart_icmp_inmsgs1 ilt_chart_icmp_inmsgs2	
In Errors InErrors		ilt_icmp_inerrors1 ilt_icmp_inerrors2	The number of ICMP messages

Secondary State Name Secondary State Definition	Graphic	Graphic Frame Files	Comment
		ilt_chart_icmp_inerrors1 ilt_chart_icmp_inerrors2	that the entity received, but determined as having ICMP-specific errors.
Out Messages OutMsgs		ilt_icmp_outmsgs1 ilt_icmp_outmsgs2	The total number of ICMP messages that this entity attempted to send.
		ilt_chart_icmp_outmsgs1 ilt_chart_icmp_outmsgs2	
Out Errors OutErrors		ilt_icmp_outerrors1 ilt_icmp_outerrors2	The number of ICMP messages that this entity did not send due to problems, such as a lack of buffers, discovered within ICMP.
		ilt_chart_icmp_outerrors1 ilt_chart_icmp_outerrors2	
TCP Group			
Current Established CurrentEstablished		ilt_tcp_current1 ilt_tcp_current2	The number of TCP connections currently established.
		ilt_chart_tcp_current1 ilt_chart_tcp_current2	
In Segments InSegs		ilt_tcp_insegs1 ilt_tcp_insegs2	The total number of segments received, including those received in error.
		ilt_chart_tcp_insegs1 ilt_chart_tcp_insegs2	
Out Segments OutSegs		ilt_tcp_outsegs1 ilt_tcp_outsegs2	The total number of segments sent, including those on current connections, but excluding those containing only retransmitted octets.
		ilt_chart_tcp_outsegs1 ilt_chart_tcp_outsegs2	
In Errors InErrors		ilt_tcp_inerrors1 ilt_tcp_inerrors2	The total number of segments received

Secondary State Name Secondary State Definition	Graphic	Graphic Frame Files	Comment
		ilt_chart_tcp_inerrors1 ilt_chart_tcp_inerrors2	in error (for example, bad TCP checksums).
Retransmitted Segments RetranSegs		ilt_tcp_retransmitted1 ilt_tcp_retransmitted2	The total number of segments retransmitted; that is, the number of TCP segments transmitted containing one or more previously transmitted octets.
		ilt_chart_tcp_retransmitted1 ilt_chart_tcp_retransmitted2	
UDP Group			
In Datagrams InDatagrams		ilt_udp_indatagrams1 ilt_udp_indatagrams2	The total number of UDP delivered datagrams.
		ilt_chart_udp_indatagrams1 ilt_chart_udp_indatagrams2	
In Errors InErrors		ilt_udp_inerrors1 ilt_udp_inerrors2	The number of received UDP datagrams that could not be delivered for

Secondary State Name Secondary State Definition	Graphic	Graphic Frame Files	Comment
		ilt_chart_udp_inerrors1 ilt_chart_udp_inerrors2	reasons other than the lack of an application at the destination port.
Out Datagrams OutDatagrams		ilt_udp_outdatagrams1 ilt_udp_outdatagrams2	The total number of UDP datagrams sent from this entity.
		ilt_chart_udp_outdatagrams1 ilt_chart_udp_outdatagrams2	
EGP Group			
In Messages InMsgs		ilt_egp_inmsgs1 ilt_egp_inmsgs2	The number of EGP messages received without error.
		ilt_chart_egp_inmsgs1 ilt_chart_egp_inmsgs2	
In Errors InErrors		ilt_egp_inerrors1 ilt_egp_inerrors2	The number of EGP messages received that proved to be in error.
		ilt_chart_egp_inerrors1 ilt_chart_egp_inerrors2	
Out Messages OutMsgs		ilt_egp_outmsgs1 ilt_egp_outmsgs2	The total number of locally generated EGP messages.
		ilt_chart_egp_outmsgs1 ilt_chart_egp_outmsgs2	
Out Errors OutErrors		ilt_egp_outerrors1 ilt_egp_outerrors2	The number of locally generated EGP messages not sent due to resource limitations within an EGP entity.
		ilt_chart_egp_outerrors1 ilt_chart_egp_outerrors2	
SNMP Group			
In Packets InPkts		ilt_snmp_inpkts1 ilt_snmp_inpkts2	The total number of messages delivered to the SNMP entity

Secondary State Name Secondary State Definition	Graphic	Graphic Frame Files	Comment
		ilt_chart_snmp_inpkts1 ilt_chart_snmp_inpkts2	from the transport service.
Out Packets OutPkts		ilt_snmp_outpkts1 ilt_snmp_outpkts2	The total number of SNMP messages that were passed from the SNMP protocol entity to the transport service.
		ilt_chart_snmp_outpkts1 ilt_chart_snmp_outpkts2	

The Misc state dictionary visuals

Describes the secondary states of the Misc state dictionary, which are used to complement those of the other state dictionaries.

In this section

Graphical representation of Misc secondary states

Provides the graphical representations of Misc secondary states.

Graphical representation of Misc secondary states

Misc secondary states are always represented graphically by adding an icon to the top left corner of the object base element.

State applicability

Unlike other state systems, table *Icon-based representations of Misc secondary states* does not specify the conditions in which the secondary states are applicable when they are combined with primary states. Use of these secondary states depends on the specific situations and the network technologies used for building the network. As a consequence, precise rules governing their usage are defined at the individual application level.

Secondary state names

In table *Icon-based representations of Misc secondary states*, the symbolic name “Secondary State Symbol” that appears after each “Secondary State Name” corresponds to the static secondary state definition. For example, the “Mismatched Card” secondary state is defined by `MismatchedCard`, which corresponds to the static definition `MismatchedCard`.

Secondary state icons

It is possible to change the icon associated with a secondary state by using global settings, see *Using global settings*. The icon property name to be used with `IltSettings.SetValue()` must only include the secondary state definition. For example:




◆ `Misc.SecState.UnknownCard.Icon`











where:

- **UnknownCard** is the secondary state definition of the “Unknown Card” secondary state

For more information on how to use global settings to modify the Misc secondary state icons, see *Customizing the Miscellaneous state system*.

Icon-based representations of Misc secondary states

Secondary State Name Secondary State Definition	Icon	Comment
Door Ajar <code>DoorAjar</code>		Customers expressed concern regarding vandalism.
High Temperature Warning <code>HighTemperatureWarning</code>		In hot weather, large numbers of minor alarms are generated due to high temperatures. The concern is that the craft may miss a potentially important minor alarm embedded in the high temperature minor alarms.
Low Temperature Warning <code>LowTemperatureWarning</code>		In cold weather, large numbers of minor alarms are generated due to low temperatures. The concern is

Secondary State Name	Icon	Comment
Secondary State Definition		
		that the craft may miss a potentially important minor alarm embedded in the low temperature minor alarms.
Mismatched Card MismatchedCard		A mismatched card is reported.
Plan to Remove PlanToRemove		The entity is subject to removal.
Software Download SoftwareDownload		Software downloading activity is being performed.
Software Limit Exceeded SoftwareLimitExceeded		The software limit has been exceeded.
Software Upload SoftwareUpload		Software uploading activity is being performed.
Test Failed TestFailed		Provides high-level notification of a failed test.
Test Passed TestPassed		Provides high-level notification of a successful test.
Threshold Crossing ThresholdCrossing		Threshold crossing of some entity performance or operational data is reported.
Under Repair UnderRepair		The resource is currently under repair.
Unknown Card UnknownCard		An unknown card is required or accessed.

The Performance state dictionary visuals

Describes the secondary states of the Performance state dictionary.

In this section

Graphical representation of Performance secondary states

Provides the graphical representations of Performance secondary states.

Graphical representation of Performance secondary states

Secondary state names

In table *Gauge and chart-based representations of Performance states*, the symbolic name “Secondary State Definition” that appears after each “Secondary State Name” corresponds to the static state definition. For example, the “Out Gb” secondary state is defined by `Out_Gb`, which corresponds to the static definition `Out_Gb`.

Secondary state icons

It is possible to change the icon associated with a secondary state by using global settings, see *Using global settings*. The icon property name to be used with `IltSettings.SetValue()` must include the secondary state definition and the decoration type. For example:

◆ `Performance.SecState.In_Kb.Gauge`


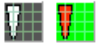












◆ `Performance.SecState.Temperature.Chart`





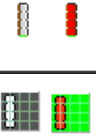


where:



- **In_Kb** and **Temperature** are the secondary state definitions of the “In Kb” and “Temperature” performance secondary states
- **Gauge** and **Chart** are the decoration types defined by the `IltDecorationType` class

For more information on how to use global settings to modify the Performance secondary state icons, see *Customizing the Performance State System*.

Gauge and chart-based representations of Performance states

Secondary State Name Secondary State Definition	Graphic	Graphic Frame Files	Comment
Input Input		Input-gauge1 Input-gauge2	A generic state for modeling any input.
		Input-chart1 Input-chart2	
In In		In-gauge1 In-gauge2	A state for modeling any input of data in bytes.
		In-chart1 In-chart2	
In Kb In_Kb		In_Kb-gauge1 In_Kb-gauge2	A state for modeling any input of data in kilobytes.
		In_Kb-chart1 In_Kb-chart2	
In Mb In_Mb		In_Mb-gauge1 In_Mb-gauge2	A state for modeling any input of data in megabytes.
		In_Mb-chart1 In_Mb-chart2	
In Gb In_Gb		In_Gb-gauge1 In_Gb-gauge2	A state for modeling any input of data in gigabytes.
		In_Gb-chart1 In_Gb-chart2	
Output Output		Output-gauge1 Output-gauge2	A generic state for modeling any output.
		Output-chart1 Output-chart2	
Out Out		Out-gauge1 Out-gauge2	A state for modeling any output of data in bytes.
		Out-chart1 Out-chart2	

Secondary State Name Secondary State Definition	Graphic	Graphic Frame Files	Comment
Out Kb Out_Kb		Out_Kb-gauge1 Out_Kb-gauge2 Out_Kb-chart1 Out_Mb-chart2	A state for modeling any output of data in kilobytes.
Out Mb Out_Mb		Out_Mb-gauge1 Out_Mb-gauge2 Out_Mb-chart1 Out_Mb-chart2	
Out Gb Out_Gb		Out_Gb-gauge1 Out_Gb-gauge2 Out_Gb-chart1 Out_Gb-chart2	A state for modeling any output of data in gigabytes.
Print Print		Print-gauge1 Print-gauge2 Print-chart1 Print-chart2	A state for representing the portion of a document that has already printed out.
Generic Generic		Generic-gauge1 Generic-gauge2 Generic-chart1 Generic-chart2	A state for representing any numeric value.
Power Power		Power-gauge1 Power-gauge2 Power-chart1 Power-chart2	A state for representing the power or voltage of an item of equipment.
Temperature Temperature		Temperature-gauge1 Temperature-gauge2 Temperature-chart1 Temperature-chart2	A state for representing the temperature of an item of equipment.

Secondary State Name Secondary State Definition	Graphic	Graphic Frame Files	Comment
Bandwidth Bandwidth		Bandwidth-gauge1 Bandwidth-gauge2	A generic state for modeling the bandwidth.
		Bandwidth-chart1 Bandwidth-chart2	

The SAN state dictionary visuals

Describes the secondary states of the SAN state dictionary.

In this section

Graphical representation of SAN secondary states

Provides the graphical representations of SAN secondary states.

Graphical representation of SAN secondary states

Secondary state names

In table *Gauge and chart based representations of SAN secondary states*, the symbolic name “Secondary State Definition” that appears after each “Secondary State Name” corresponds to the static state definition. For example, the “Back Recovery” secondary state is defined by `BackRecovery`, which corresponds to the static definition `BackRecovery`.

Secondary state icons

It is possible to change the icon associated with a secondary state by using global settings, see *Using global settings*. The icon property name to be used with `IlSettings.SetValue()` must include the secondary state definition and the decoration type. For example:

◆ `SAN.SecState.LostData.Gauge`

◆ `SAN.SecState.IO.Chart`



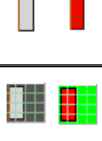



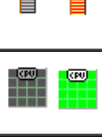

where:










- **LostData** and **IO** are the secondary state definitions of the “Lost Data” and “I/O” SAN secondary states
- **Gauge** and **Chart** are the decoration types defined by the `IlDecorationType` class

For more information on how to use global settings to modify the SAN secondary state icons, see *Customizing the SAN state system*.

Gauge and chart based representations of SAN secondary states

Secondary State Name	Graphic	Graphic Frame Files	Comment
Secondary State Definition			

Secondary State Name	Graphic	Graphic Frame Files	Comment
Secondary State Definition Allocated Allocated		Allocated-gauge1 Allocated-gauge2 Allocated-chart1 Allocated-chart2	A state to model allocated storage space.
Available Available		Available-gauge1 Available-gauge2 Available-chart1 Available-chart2	
Back Recovery BackRecovery		BackRecovery-gauge1 BackRecovery-gauge2 BackRecovery-chart1 BackRecovery-chart2	A state to model back recovery.
Bandwidth BandWidth		Bandwidth-gauge1 In_Mb-gauge2 Bandwidth-chart1 Bandwidth-chart2	
Capacity Capacity		Capacity-gauge1 Capacity-gauge2 Capacity-chart1 Capacity-chart2	A state to model storage capacity.
Capacity Utilization CapacityUtilization		CapacityUtilization-gauge1 CapacityUtilization-gauge2 CapacityUtilization-chart1 CapacityUtilization-chart2	
CPU Power CPU		CPU-gauge1 CPU-gauge2 CPU-chart1 CPU-chart2	A state to model CPU power.
Data Access Delay DataAccessDelay		DataAccessDelay-gauge1 DataAccessDelay-gauge2	

Secondary State Name Secondary State Definition	Graphic	Graphic Frame Files	Comment
		DataAccessDelay-chart1 DataAccessDelay-chart2	
Fragmentation Fragmentation		Fragmentation-gauge1 Fragmentation-gauge2	A state to model disk fragmentation.
		Fragmentation-chart1 Fragmentation-chart2	
I/O IO		IO-gauge1 IO-gauge2	A state to model I/O.
		IO-chart1 IO-chart2	
Lost Data LostData		LostData-gauge1 LostData-gauge2	A state to model lost data.
		LostData-chart1 LostData-chart2	
Usage Usage		Usage-gauge1 Usage-gauge2	A state to model disk usage.
		Usage-chart1 Usage-chart2	

The SONET state dictionary visuals

Describes the six primary states and six secondary states of the SONET state dictionary.

In this section

Graphical representation of SONET primary states

Illustrates the graphical representation of the SONET primary states.

Graphical representation of SONET secondary states

Illustrates the graphical representation of the SONET secondary states, or protection switch request indicators.

Graphical representation of SONET primary states







Common pairs of SONET primary states describes the graphical representations of the common double SONET states.

SONET primary states are rendered graphically by changes in the base colors, line styles, and relief.






State names

In these tables, the symbolic name “Link State Symbol” that appears after each primary state name corresponds to the last part of the variable name holding the state. For example, the “ActiveProtecting” state, which has `ActiveProtecting` for symbol, has the variable `ActiveProtecting` associated with it in the library.

Graphical representation of SONET primary states

Link State	Link Base
Link State Symbol	
Disabled Disabled	
Inactive Inactive	
Active Active	
Active and protecting ActiveProtecting	
Troubled and protected TroubledProtected	
Troubled and unprotected TroubledUnprotected	

Common pairs of SONET primary states

Link State	Link Base
Link State Symbols	
Disabled in both directions Disabled Disabled	
Disabled in one direction, active in the other direction Disabled Active	
Inactive in both directions Inactive Inactive	
Inactive in one direction, active in the other direction Inactive Active	
Active in both directions Active	

Link State Link State Symbols	Link Base
Active	
Troubled and protected in one direction, active in the other direction TroubledProtected Active	
Active and protecting in both directions ActiveProtecting ActiveProtecting	
Active and protecting in one direction, inactive in the other direction ActiveProtecting Inactive	
Troubled and protected in both directions TroubledProtected TroubledProtected	
Troubled and protected in one direction, inactive in the other direction TroubledProtected Inactive	
Troubled and protected in one direction, troubled and unprotected in the other direction TroubledProtected TroubledUnprotected	
Troubled and unprotected in both directions TroubledUnprotected TroubledUnprotected	
Troubled and unprotected in one direction, active in the other direction TroubledUnprotected Active	

Graphical representation of SONET secondary states

The protection switch indicators are always represented by icons that appear near the end of the link on which they are set.

Secondary state names

In *Icon-based representations of SONET secondary states*, the symbolic name “Secondary State Definition” that appears after each “Secondary State Name” corresponds to the static secondary state definition. For example, the “Manual Switch” secondary state is defined by `ManualSwitch`, which corresponds to the static definition `ManualSwitch`.

Secondary state icons

It is possible to change the icon associated with a secondary state by using global settings, see *Using global settings*. The icon property name to be used with `IlSettings.SetValue()` must only include the secondary state definition. For example:







◆ `SONET.Protection.ManualSwitch.Icon`

where:

- `ManualSwitch` is the secondary state definition of the “Manual Switch” state

For more information on how to use global settings to modify the SONET secondary state icons, see *Customizing the SONET state system*.

Icon-based representations of SONET secondary states

Secondary State Name Secondary State Definition	Icon	Comment
Exercisor Exercisor		Facility is currently undergoing test.
Forced switch ForcedSwitch		Forced protection switch request has been triggered.
Locked Locked		No traffic is allowed on the facility. This restriction may be imposed for maintenance reasons.
Manual switch ManualSwitch		Manual protection switch request has been triggered.
Pending Pending		Pending protection switch request.
Wait-to-restore WaitToRestore		Waiting for the facility service to be restored.

States

Introduces the visual dictionaries that are used for displaying state changes in predefined telecom business objects. Also introduces the object state classes and explains how to set states to predefined business objects.

In this section

Graphical representations of predefined business object states

Presents the visual cues used to represent states or alarms on predefined business objects.

State dictionaries: an overview

Introduces each state dictionary as well as the concepts of primary and secondary states.

The OSI state dictionary

Describes the states and statuses that are defined in the OSI state dictionary.

The Bellcore state dictionary

Describes the states that are defined in the Bellcore state dictionary.

The SNMP state dictionary

Describes the states that are defined in the SNMP state dictionary.

Miscellaneous states: the Misc state dictionary

Provides information and pointers to the Misc secondary state dictionary.

Performance states: the Performance state dictionary

Provides information and pointers to the Performance secondary state dictionary.

SAN states: the SAN state dictionary

Provides information and pointers to the SAN secondary state dictionary.

Link states: the SONET state dictionary

Describes the SONET primary and secondary states associated with links.

Alarm states

Describes the characteristics of graphical representations of telecom object alarm conditions. Also explains how to define an alarm state with the API and in XML.

Trap states

Describes the different trap types with their graphical representation and explains how to define trap states with the API and in XML.

Managing states

Explains how to change and withdraw the states associated with JViews TGO objects, by showing how to use the state dictionaries.

Defining states in XML

Explains how to define object states in the XML format and load them in a data source.

Information window

Shows how secondary states are represented in a telecom object depending on their number.

System window

Shows how the presence of system attributes is represented in a telecom object.

Customizing the representation of states and alarms

Provides a pointer to the section of the documentation which explains how to customize the representation of states and alarms.

Graphical representations of predefined business object states

IBM® ILOG® JViews TGO provides a comprehensive, user-friendly interface designed to illustrate changes in network telecom equipment states.

Within this framework, JViews TGO includes a wide range of visual techniques to identify changes in equipment states and alarms. These visual cues are a combination of:

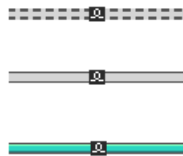
- ◆ Several base representations for network elements, links, and cards.

The following illustration shows a network element in three of the fundamental states, namely Out Of Service (OOS), In service but carrying No Traffic (NT), and In service and Carrying Traffic (CT).



Network management workstation in different states

- ◆ Variable visual parameters for links and regions (thickness, line style, color, pattern).



Links showing disabled, inactive, and active states

- ◆ Sets of decorations (icons, labels, balloons, gauges).



Network elements and links with icons

In addition, a number of graphical properties have been developed to notify the telecommunication network operator that alarms are present. When a new alarm is identified on an element, four visual cues are added to its graphical representation:

- ◆ An alarm counter is displayed in the network element base
- ◆ An alarm balloon appears above the network element displaying another alarm count.
- ◆ The object base turns a vibrant color (red, orange, or yellow) depending on the alarm severity.
- ◆ A colored outline is associated with the object base.



A network element in three different alarm states

Passive devices

Predefined business objects that do not have a specific object state are considered as *passive devices*. These devices do not report information about their current states and alarms. Passive devices are graphically represented with an icon located at the same position as alarm counts in the case of alarms.

The passive icon is displayed when the object is in its regular representation, but it is hidden when the network element is collapsed. The following example illustrates the use of passive devices with network elements.

How to use passive devices

```
object."ilog.tgo.model.IltNetworkElement" {
  collapsed: true;
  passiveIconVisible: false;
}
object."ilog.tgo.model.IltNetworkElement":selected {
  collapsed: false;
  passiveIconVisible: true;
}
```

The result is shown in the following images:



Passive device expanded



Passive device collapsed (passive icon not visible)

State dictionaries: an overview

To group graphic operations for displaying states and to make sets of relevant states available, JViews TGO provides several state dictionaries based on worldwide standards. JViews TGO associates with each state in a dictionary the drawing mode of the telecom object holding that state. Therefore, the drawing is updated when the state of the graphic object is modified. This association between a state and a graphic drawing is detailed throughout this section.

The dictionaries available in the JViews TGO library are presented with their definitions and icons. The development of these dictionaries is based on current telecom standards combined with hands-on experience.

- ◆ The OSI state dictionary is based on the standard ISO/IEC 10164-2, ITU-T X.731: *State Management Function*.
- ◆ The Bellcore state dictionary is based on Bellcore GR-1093: *Generic State Requirements for Network Elements*.
- ◆ The SNMP state dictionary is based on RFC 1213: *Management Information Base for Network Management of TCP/IP based internets: MIB-II*.
- ◆ The SONET state dictionary is dedicated to the states of reporting transport links.
- ◆ The Misc state dictionary includes additional states that are absent from other standards, but are useful to several network operators.
- ◆ The Performance state dictionary includes additional states that are absent from other standards, but are useful to model numeric values such as performance indicators or service levels.
- ◆ The SAN state dictionary includes additional states that are absent from other standards, but are useful to model numeric values such as storage indicators.
- ◆ The Alarm state dictionary is the state model proposed by JViews TGO to display the set of alarms assigned to a telecom object.
- ◆ The Trap state dictionary is the state model proposed by JViews TGO and based on RFC 1157 - *A Simple Network Management Protocol (SNMP)* - to display the set of traps assigned to a telecom object.

Reference tables for the graphical representations of these state dictionaries are contained in *Lookup tables for state visuals*.

Primary and secondary states

The OSI, Bellcore, SNMP and SONET state dictionaries all contain the notion of primary and secondary states. The difference between a primary and a secondary state is that a telecom object will usually carry one and only one primary state, whereas it can carry a number of secondary states.

Dictionary nomenclatures

While the dictionaries described in the following sections contain the notion of primary and secondary states, the terms traditionally used to describe them are not always the same.

- ◆ The Bellcore and SNMP dictionaries use the terms ‘primary state’ and ‘secondary state.’
- ◆ The OSI dictionary uses the terms ‘state’ and ‘status.’
- ◆ The SONET dictionary ties the notion of ‘secondary state’ to protection switch request indicators.

For our purposes, we will use the terms *primary state* and *secondary state* in a generic sense.

Primary states allowing secondary states

The applicability of secondary states in the OSI and Bellcore models depends on primary states. Most secondary states can be set on a telecom object only if the object is already in a predefined state. For instance, the OSI Power-Off status can be set only on an object that is out-of-service, that is, with the OSI Disabled primary state.

The three conditions that determine whether a given secondary state applies to a telecom object are:

- ◆ Out Of Service (OOS)
- ◆ In service, Carrying No Traffic (NT)
- ◆ In service, Carrying Traffic (CT)

These conditions represent a combination of primary states in all state dictionaries.

The applicability of secondary states on given primary states can be seen in *Icon-based representations of OSI secondary states* .

The OSI state dictionary

The OSI state dictionary is based on the OSI SMF 10164-2 standard, which defines the primary state of a telecom object as a combination of three values, and also introduces a number of statuses.

OSI states

An OSI state is a triplet including the following states:

◆ *Operational*, which can be one of the following:

- Disabled
- Enabled

◆ *Usage*, which can be one of the following:

- Idle
- Active
- Busy

◆ *Administrative*, which can be one of the following:

- Unlocked
- Shutting Down
- Locked

Valid OSI states

While the OSI state system definition above allows 18 combinations of states (2x3x3), only eight of them are meaningful and thus legal. These are:

1. Operational: Disabled; Usage: Idle; Administrative: Unlocked
2. Operational: Disabled; Usage: Idle; Administrative: Locked
3. Operational: Enabled; Usage: Idle; Administrative: Unlocked
4. Operational: Enabled; Usage: Idle; Administrative: Locked
5. Operational: Enabled; Usage: Active; Administrative: Unlocked
6. Operational: Enabled; Usage: Active; Administrative: Shutting down
7. Operational: Enabled; Usage: Busy; Administrative: Unlocked
8. Operational: Enabled; Usage: Busy; Administrative: Shutting down

OSI statuses

In addition to the states already mentioned, the OSI SMF standard includes a status property, which is used to complement the primary state. JViews TGO provides a comprehensive set of status values for which a graphical interpretation is available. This status set is divided into five groups:

- ◆ *Procedural* is used to report whether the managed object has been properly or improperly initialized or is finally reporting.
- ◆ *Availability* is used to determine the availability status of the managed object.
- ◆ *Control* is used to determine if a managed object is reserved for test or subject to test.
- ◆ *Standby* is used to identify a managed resource that does not provide a service, but which can immediately take over the role of a primary resource.
- ◆ *Repair* is used to determine whether the managed resource is under repair.

The OSI states and statuses are individually described in the following reference tables:

- ◆ *Graphical representation of the eight valid OSI primary states*
- ◆ *Icon-based representations of OSI secondary states*
- ◆ *Other OSI secondary state representations*

For information on how to customize OSI states, refer to Customizing the OSI state system.

The Bellcore state dictionary

In the Bellcore state dictionary, a primary state is defined as holding one of the following values:

- ◆ Disabled/Idle
- ◆ Enabled/Idle
- ◆ Enabled/Active

The Bellcore dictionary also includes numerous secondary states. All of the secondary states, in addition to the primary states, are individually described in the following reference tables:

- ◆ *Graphical representation of Bellcore primary states* .
- ◆ *Icon-based representations of Bellcore secondary states* .
- ◆ *Other representations of Bellcore secondary states*

For information on how to customize Bellcore states, refer to Customizing the Bellcore state system.

The SNMP state dictionary

The SNMP state dictionary is based on RFC 1213 - Management Information Base for Network Management of TCP/IP-based internets - MIB-II. This document defines a management information base, which is organized in nine groups:

- ◆ *System* Provides general information about the managed system.
- ◆ *Interfaces* Provides generic information about the physical interfaces of the entity, including configuration information and statistics on the events occurring at each interface.
- ◆ *IP* Contains information relevant to the implementation and operation of IP at a node.
- ◆ *ICMP* Contains information relevant to the implementation and operation of ICMP at a node.
- ◆ *TCP* Contains information relevant to the implementation and operation of TCP at a node.
- ◆ *UDP* Contains information relevant to the implementation and operation of UDP at a node.
- ◆ *EGP* Contains information relevant to the implementation and operation of EGP at a node;
- ◆ *Transmission* Contains objects that provide details about the underlying transmission medium for each interface on a system;
- ◆ *SNMP* Contains information relevant to the implementation and operation of SNMP.

Primary state

In the SNMP State Dictionary, the primary state is based on the valid combinations of the Administrative and Operational Status, as they are defined in the Interfaces Group. This primary state holds one of the following values:

- ◆ *Up* Administrative Status is Up/Operational Status is Up
- ◆ *Down* Administrative Status is Down/Operational Status is Down
- ◆ *Testing* Administrative Status is Testing/Operational Status is Testing
- ◆ *Failed* Administrative Status is Up/Operational Status is Down
- ◆ *Shutdown* Administrative Status is Down/Operational Status is Up

You can extend this primary state to take into account your own state definitions and their associated representations. More information on extensions is provided in Customizing object states.

For visuals, see *Graphical representation of SNMP primary states*.

Secondary states

The SNMP State Dictionary also includes numerous secondary states. The secondary states are divided according to the group where they are referenced in MIB-II.

Most of the variables defined in MIB-II are counters, holding a numeric value ranging from 0 to 232-1. The valid interval for the SNMP secondary numeric states can be configured, using the `IltLimitedNumericState` API. The graphic representation of each state can be configured as a Gauge, Counter or Chart. Customization is explained in detail in .

The valid interval for the SNMP secondary numeric states can be configured, as shown in this code extract:

```
IltSNMP.Interface.InDiscards.setMaxValue (new Float (100.0));
```

For visuals, see *Graphical representation of SNMP secondary states*.

For information on how to customize SNMP states, refer to *Customizing the SNMP state system*.

System group

As well as defining the secondary states, which are represented graphically as gauges or charts, MIB-II also defines a group called System. This group is responsible for storing general information (such as location, contact person, and description) about the object being managed.

In the SNMP State Dictionary, this group is represented by a set of attributes. The attributes present in an object are mapped graphically to a window called System window. (See *System window*.)

Attributes can be added to the System Group and represented in the System window with their values. More information on this extension is provided in *Creating a new attribute in the System group*.

Miscellaneous states: the Misc state dictionary

The Misc State Dictionary provides secondary state values that can be used to complement OSI, Bellcore or SNMP standards. The secondary states included in this dictionary are often used in telecom network supervision applications.

This dictionary can be extended by the JViews TGO user to take into account his own state definitions and their associated icons. More information about extensions is provided in Customizing the secondary state icons.

All of the Misc states are individually described in the reference table *Icon-based representations of Misc secondary states* .

For information on how to customize Miscellaneous states, refer to Customizing the Miscellaneous state system.

Performance states: the Performance state dictionary

The Performance State Dictionary provides secondary state values that can be used to complement any other standard state system such as OSI, Bellcore, SNMP, or SONET. The secondary states included in this dictionary can be used to model and represent any state with a numeric value. For example, any numeric value corresponding to performance information or a service level can be modeled using this state system. Any secondary state in the Performance State Dictionary can be represented graphically by a gauge, a chart or a numeric counter.

The Performance secondary numeric states hold a numeric value ranging from 0 to $2^{32}-1$. The valid interval for the Performance secondary numeric states can be configured through the `IltLimitedNumericState` API. The graphic representation of each state can also be configured as a gauge, chart or counter.

Customization is explained in detail in [Customizing the Performance State System](#).

The valid interval for the Performance secondary numeric states can be configured, as shown in this code extract:

```
IltPerformance.SecState.In.setMaxValue(new Integer(100));
```

This dictionary can be extended by the JViews TGO user to take into account his own state definitions and their associated decorations. More information about extensions is provided in [Creating new Performance secondary states](#).

All of the Performance states are individually described in the reference table *Gauge and chart-based representations of Performance states*.

SAN states: the SAN state dictionary

The SAN (Storage Area Network) State Dictionary provides secondary state values that can be used to complement any other standard state system such as OSI, Bellcore, SNMP or SONET. The secondary states included in this dictionary can be used to model and represent any state with a numeric value. For instance, any numeric value corresponding to SAN information can be modeled using this state system. Any secondary state in the SAN State Dictionary can be represented graphically by a gauge, a chart, or a numeric counter.

The SAN secondary numeric states hold a numeric value ranging from 0 to $2^{32}-1$. You can configure the valid interval for the SAN secondary numeric states using the `IlLimitedNumericState` API. You can also configure the graphic representation of each state as a gauge, chart or counter.

Customization is explained in detail in [Customizing the SAN state system](#).

The valid interval for the SAN secondary numeric states can be configured, as shown in this code extract:

```
IlSAN.SecState.Allocated.setMaxValue(new Integer(100));
```

This dictionary can be extended by the JViews TGO user to take into account his own state definitions and their associated decorations. More information about extensions is provided in [Creating new SAN secondary states](#).

All of the SAN states are individually described in the reference table *Gauge and chart based representations of SAN secondary states*.

Link states: the SONET state dictionary

The SONET State Dictionary groups states and indicators that are used most often to display transport links with the protection process. Such link state graphics are useful only in applications in which the end user must be informed about link states and protection switching information (as in a fiber transport network, for example).

This state system should not always be used, for the following reasons:

- ◆ The states are not necessarily adapted to the specific application case. For example, you may want to ignore the issue regarding the protection facility.
- ◆ In some network representation cases, you may decide not to use these graphics, since they might interfere with other managed object state graphics.

Primary states

The SONET State Dictionary includes a certain number of primary states involving changes in the drawing of the link base.

Although this set of states is independent of OSI or Bellcore standards, it shares the following common subset with the following three main states:

- ◆ *Disabled*: the facility is Out Of Service (OOS).
- ◆ *Inactive*: the working facility is in service, carrying No Traffic (NT).
- ◆ *Active*: the working facility is in service, Carrying Traffic (CT).

The other states determine the state of the link in relation to the usage and state of the protection facility:

- ◆ *Active and Protecting*: the working facility is carrying traffic and the protection facility is enabled.
- ◆ *Troubled and Protected*: the working facility is troubled (failure conditions are reported) and the protection facility is active (the failure conditions triggered the protection switch).
- ◆ *Troubled and Unprotected*: the working facility is troubled and the protection facility is unable to protect.

Secondary states

The SONET State Dictionary includes a number of secondary states, or *protection switch request indicators*, that deal with the protection process. These states can be applied either at both link ends or at one link end only. Programming such states is explained in *Setting link states*.

You can extend the list of the SONET secondary states to take into account your own state definitions and their associated icons. More details about extensions are provided in *Customizing the SONET state system*.

The SONET primary and secondary states are described individually in the following reference tables:

- ◆ *Graphical representation of SONET primary states*
- ◆ *Common pairs of SONET primary states*
- ◆ *Icon-based representations of SONET secondary states*

Alarm states

Describes the characteristics of graphical representations of telecom object alarm conditions. Also explains how to define an alarm state with the API and in XML.

In this section

Graphical representation of alarm conditions

Describes the characteristics of graphical representations of telecom object alarm conditions.

Setting the alarm counters

Describes the different approaches to set alarm counters.

Defining alarm states with the API

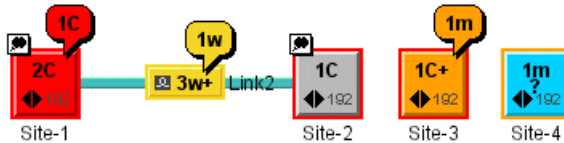
Explains how to set alarm states using the API.

Loading alarm states in XML

Provides a pointer on how to load alarm states in XML.

Graphical representation of alarm conditions

The network component provides a graphical representation for the alarm state of managed objects. New raw alarms are represented by a round rectangle alarm balloon; new impact alarms are represented by an impact alarm cloud. In addition, a secondary alarm state representation is provided for cases when an object has both impact and raw alarms.



Representation of alarm states in a network

Graphical cues for alarm states

The main graphical cues for alarms are:

- ◆ A color associated with the object base element (the alarm coding color scheme is illustrated in *Raw alarm color coding scheme*).
- ◆ An alarm count summary displayed on the object base element (the contents of the summary is explained below).
- ◆ A colored alarm balloon displaying also an alarm count summary.
- ◆ A colored outline displayed around the object base element.
- ◆ A secondary state icon for the secondary alarm state (see *Primary and secondary states*.)

Alarm state details

Alarms can be either new or acknowledged. The term outstanding alarms includes both new and acknowledged alarms.

The graphical representation of a telecom object alarm condition shows the number and highest severity of new alarms and the number and highest severity of outstanding alarms in the following manner:

- ◆ The color of the base element and the color of the alarm balloon are those associated with the most severe new alarm (see *Raw alarm color coding scheme*).
- ◆ The color of the outline around the base element is the one associated with the most severe outstanding alarm.
- ◆ The alarm count summary in the base element displays the number of outstanding alarms.
- ◆ The alarm count in the alarm balloon displays the number of new alarms.

Alarm count summary

An alarm count summary displays the number and highest severity of new or outstanding alarms. It is composed of:

- ◆ A number indicating the amount of most severe new or outstanding alarms.
- ◆ One or more letters indicating the highest severity of new or outstanding alarms (see *Raw alarm color coding scheme*).
- ◆ An optional icon to represent the highest severity of new or outstanding alarms.
- ◆ A plus sign indicating that there are also less severe new or outstanding alarms.

Primary and secondary alarm states

A telecom object can have raw and impact alarms at the same time, which introduces the concepts of primary and secondary alarm states. The primary alarm state has a more detailed representation than the secondary alarm state. By default, the primary alarm state is the raw alarm state.

The primary alarm state is identified by the following:

- ◆ The color of the base element and the color of the alarm balloon, which correspond to the most severe new alarm.
- ◆ The shape of the alarm balloon.
- ◆ The color of the outline of the base element, which corresponds to the most severe outstanding alarm.
- ◆ The alarm count summary in the base element displaying the number of outstanding alarms.
- ◆ The alarm count in the alarm balloon displaying the number of new alarms.

The secondary alarm state is identified by the following:

- ◆ The secondary alarm state icon.



LEFT: primary alarm state for raw alarms. RIGHT: primary alarm state for impact alarms

Alarm severity coding

JViews TGO provides two types of alarms: raw and impact alarms.

Raw alarms have a range of six severity levels, including the Cleared severity. These levels and their associated color and short text are shown in *Raw alarm color coding scheme*.

Raw alarm color coding scheme

Severity	Color	Text
Critical	Red	C
Major	Red	M
Minor	Orange	m
Warning	Yellow	w
Unknown	Grey	u
Cleared	Not represented	Not represented

Note: The Cleared severity is never represented in alarm states.

Impact alarms have a range of ten severity levels, including the Cleared severity. These levels and their associated color, short text and icon are shown in *Impact alarm color coding scheme*.

Impact alarm color coding scheme



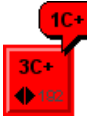



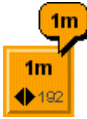











Severity	Color	Text	Icon
Critical High	Red	!!	🚨
Critical Low	Red	!	🚨
Major High	Red	!!	🚨
Major Low	Red	!	🚨
Minor High	Orange	!!	🚨
Minor Low	Orange	!	🚨
Warning High	Yellow	!!	🚨
Warning Low	Yellow	!	🚨
Unknown	Grey	!	🚨
Cleared	Not represented	Not represented	Not represented







Note: The Cleared severity is never represented in alarm states.

You can extend the default severity levels of raw and impact alarms. You can add new severity levels and associate them with a color and label. Functions for extending severity levels, as well as examples, are provided in Customizing alarm severities.

Alarm state graphical representations provides some alarm state examples and their associated visual representations.

Alarm state graphical representations

Alarm State Values	Visual in Network and Equipment	Visual in Table and Tree	Comment
New Critical			The resource has one new critical alarm.
Outstanding Critical			The resource has one new critical alarm, plus less severe new alarms, and one acknowledged critical alarm.
New Major			The resource has one new major alarm, plus less severe new alarms.
New Minor			The resource has one new minor alarm.
Acknowledged Minor			The resource has two acknowledged minor alarms, plus acknowledged less severe alarms.
Outstanding Warning			The resource has three new warning alarms, plus two acknowledged warning alarms.
New Unknown			The resource has a new unknown alarm.
New Critical High Impact			The resource has a new critical high impact alarm; the primary alarm state is for impact alarms.
Impact Alarms			The resource has some impact alarms; the primary alarm state is for raw alarms.

Alarm State Values	Visual in Network and Equipment	Visual in Table and Tree	Comment
New Minor Low Impact and Raw Alarms			The resource has a new minor low impact alarm and some raw alarms; the primary alarm state is for impact alarms.
Loss of Connectivity			The alarm collection process has been shut down or alarm counts are not reliable.
Not Reporting			Alarm reporting has been suspended because the resource was purposely taken offline, for example for repairs.

Setting the alarm counters

JViews TGO offers two approaches to setting the alarm counters:

- ◆ *The back end computes the alarm state counters*
- ◆ *JViews TGO computes alarm state counters*

It is possible to combine the two computation models, having JViews TGO compute the alarm counters for some managed objects and the back end for the other managed objects.

The back end computes the alarm state counters

To set the counters:

1. Remove from the data source all alarms related to the managed object.
2. Set the `computedFromAlarmList` property of the alarm state to `false`.
3. Retrieve the counters from the back end and set the values in the alarm state.

To update the counter values according to changes in the back end:

1. Subscribe to the notification of alarm counter changes by the back end.
2. Update the alarm counter in the alarm state according to the back-end notifications.

JViews TGO computes alarm state counters

JViews TGO can compute alarm state counters for a given managed object based on the list of alarms for this object.

To set the counters:

1. Use an `IltDefaultDataSource` for the data source.
2. Set the `computedFromAlarmList` property of the alarm state to `true`.
3. Retrieve the list of alarms for the managed object and create the corresponding `IltAlarm` objects. In the `IltAlarm` objects, set the `managedObjectInstance` attribute value to the object identifier of the managed object.
4. Add the `IltAlarm` objects in the same data source as the `IltObject` corresponding to the managed object.

To update the alarm state according to changes in the back end:

1. Subscribe to the notification of alarms and alarm list changes for the object.
2. Add, remove or update the `IltAlarm` business objects according to the changes.

The `handlingAlarmReferences` property of `IltDefaultDataSource` controls whether the alarm state consolidation occurs or not in a given data source. By default the consolidation is active.

Combining alarm counter computation models

You can have JViews TGO compute the alarm counters for some managed objects, and the back end compute the counters for the other managed objects. You can also switch from one computation model to the other for a given managed object.

Note: The concurrent use of the two models on the same managed object would result in inconsistent and inaccurate counters. Use `IltAlarm.State.setComputedFromAlarmList` to switch between the two modes. When the alarm counters are computed from the associated list of alarms, attempts to change the alarm counters directly with the alarm state APIs would trigger an `IllegalStateException`. Instead, update the associated `IltAlarm` objects.

How to switch from alarm counters computed by the back end to alarm counters computed by JViews TGO

1. Unsubscribe to the notification of alarm counter changes by the back end.
2. Disable the automatic alarm counter computation of the alarm state using the method `IltAlarm.State.setComputedFromAlarmList`. This will reset the counters to zero.
3. Retrieve the list of alarms for the managed object and create the corresponding `IltAlarm` objects. In the `IltAlarm`, set the `ManagedObjectInstanceAttribute` value to the object identifier of the managed object.
4. Add the `IltAlarm` objects in the same data source as the `IltObject` corresponding to the managed object.
5. Subscribe to alarm list change notifications by the back end.
6. Add, remove or update `IltAlarm` objects according to the back-end notifications.

How to switch from alarm counters computed by JViews TGO to alarm counters computed by the back end

1. Unsubscribe to the notification of alarm list changes by the back end.
2. Remove from the data source all the alarms related to the managed object.
3. Enable the automatic alarm counter computation of the alarm state using the method `IltAlarm.State.setComputedFromAlarmList`. This will reset the counters to zero.
4. Retrieve the counters from the back end and set the values in the alarm state.
5. Subscribe to the notification of alarm counter changes by the back end.
6. Update the alarm counters in the alarm state according to the back-end notifications.

Defining alarm states with the API

Alarms carried by telecom objects are described in the same way as states. Alarms are held by one of the `IltObjectState` object attributes and are modeled using an instance of the class `IltAlarm.State`, which is a subclass of `IltState`. The alarm state object includes the following information:

- ◆ For each alarm severity:
 - The number of new alarms of this severity
 - The number of acknowledged alarms of this severity
- ◆ Boolean indicators for special conditions:
 - Not Reporting: the equipment has ceased reporting alarms.
 - Loss of Connectivity: the connection with the equipment has been lost, thus making the recorded number of alarms unreliable.

The alarm model is the same for most of the existing object state classes. When the alarm information is based on the alarm model, you can retrieve this information by calling the method `IltObject.getAlarmState()`.

How to set alarm counters

The API for managing alarms is directly available from the `IltObject` class.

The following code line retrieves the object state corresponding to the alarms for an object named `paris`.

```
IltAlarm.State alarms = paris.getAlarmState();
```

The following code line shows how to set the count of new alarms to the object.

```
alarms.setNewAlarmCount(IltAlarm.Severity.Critical, 2);
```

To set the count of acknowledged alarms, use the following code:

```
alarms.setAcknowledgedAlarmCount(IltAlarm.Severity.Critical, 2);
```

Alarms can be set either directly, as shown above, or incrementally, as in the following code where a new critical alarm is added to the same network element.

How to set alarms incrementally

```
alarms.addNewAlarm(IltAlarm.Severity.Critical);
```

```
System.out.println("Critical alarms on Paris: "  
    + alarms.getNewAlarmCount(IltAlarm.Severity.Critical));
```

The printed output is the following:

```
Critical alarms on Paris: 3 new
```

How to set special alarm statuses

Special alarm statuses are set and unset using dedicated APIs.

To switch to the loss-of-connectivity mode, use the following:

```
alarms.setLossOfConnectivity(true);
```

To switch to the not-reporting mode, use the following:

```
alarms.setNotReporting(true);
```

Loading alarm states in XML

For details on how to load alarm states in XML, refer to *Alarm states*.

Trap states

JViews TGO provides another concept of alarm that is based on RFC 1157 - A Simple Network Management Protocol (SNMP). Basically, a trap represents something unusual that occurs in an object. Traps, as well as alarms, are represented graphically using the alarm balloon and alarm count decorations. The graphical cues described in the *Graphical cues for alarm states* are also valid when representing traps. Another similarity with alarms is the concept of new, acknowledged and outstanding traps, which were explained in *Alarm state details*.

Trap type coding

JViews TGO provides the following range of trap types and their associated graphic representations. These types are based on the generic traps defined in RFC 1157.



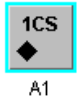

Trap color coding scheme

Type	Color	Letter
Link Failure	Red	LF
Authentication Failure	Red	AF
Cold Start	Blue	CS
Warm Start	Orange	WS
EGP Neighbor Loss	Yellow	NL

You can add new trap types and associate them with a color and label. Functions for extending traps, as well as examples, are provided in Customizing trap types.

Trap state graphical representations illustrates some alarm status examples and their associated visual representations.

Trap state graphical representations

Trap Status	Visual	Comment
New Link Failure		The resource has received a link failure trap.
New Link Failure		The resource has received one new link failure trap, plus other less severe new traps.
Acknowledged Cold Start		The resource has received one new link failure trap, plus other less severe new traps.
Acknowledged Link Failure traps and New Neighbor loss traps		The resource has acknowledged link failure traps, as well as new EGP neighbor loss traps.

Trap types

JViews TGO defines five trap types based on RFC 1157; they are the elements of the `IltTrap.Type` enumeration. These instances of `IltTrap.Type` are statically allocated and are stored in static data members of `IltTrap`.

Available severity values are:

- ◆ `IltTrap.Type.LinkFailure`
- ◆ `IltTrap.Type.AuthenticationFailure`
- ◆ `IltTrap.Type.EGPNeighborLoss`
- ◆ `IltTrap.Type.ColdStart`
- ◆ `IltTrap.Type.WarmStart`

Other types can be defined to extend the default trap model. See Customizing trap types for an explanation of how to do this.

Defining trap states with the API

Traps carried by telecom objects are described in the same way as states. Traps are held by one of the `IltObjectState` object attributes and are modeled using an instance of the class `IltTrap.State`, which is a subclass of `IltState`.

Make sure output of the apilink is `IltTrap.State`

For each trap type, the alarm object includes the following information:

- ◆ the number of new traps
- ◆ the number of acknowledged traps.

The trap model is the alarm representation for SNMP-based objects. To retrieve the information about traps, you should use the method `IltObject.getTrapState()`.

The following code shows how traps are managed on the `paris` network element. The API for managing traps is directly available from the `IltObject` class.

How to set traps directly

The following code line retrieves the object state corresponding to the alarms for Paris:

```
IltTrap.State alarms = paris.getTrapState();
```

The following code line shows how to set new alarms on the object.

```
alarms.setNewAlarmCount(IltTrap.Type.LinkFailure, 2);
```

To set acknowledged traps, use the following code:

```
alarms.setAcknowledgedAlarmCount(IltTrap.Type.LinkFailure, 2);
```

Traps can be set either directly as shown above, or incrementally as in the following code sample, where a new *Link Failure* trap is added to the same network element.

How to set traps incrementally

```
alarms.addNewAlarm(IltTrap.Type.LinkFailure);
System.out.println("Link Failure traps on Paris: "
    + alarms.getNewAlarmCount(IltTrap.Type.LinkFailure));
```

The printed output is the following:

```
Link Failure traps on Paris: 3 new
```

Defining trap states in XML

For details on how to load alarm states in XML, refer to *Trap states*.

Managing states

Explains how to change and withdraw the states associated with JViews TGO objects, by showing how to use the state dictionaries.

In this section

State values, state classes, and state systems

Introduces the object state classes and explains the naming principles of the states listed in the dictionaries.

Object states

Provides a class diagram of the object states classes and lists the dictionaries associated with each object state.

The object state classes

Describes how states are structured and named using the OSI, Bellcore, SNMP, Misc, SONET, Alarm, and Trap object state classes.

Modifying states and statuses

Provides examples illustrating how to manage the states and statuses for network elements and links.

Accessing and removing states

Explains how to access and how to remove states with the API.

State values, state classes, and state systems

The Java™ classes introduced by JViews TGO to implement the concepts in state models are `IltState` and `IltStateSystem`.

State classes are subclasses of `IltState`. Their instances are the state values that will be used to describe the state and alarm condition of a telecom object. Examples of state values are: the primary state of a telecom object, each of its secondary states, or its alarm state.

The state dictionaries are embodied by instances of `IltStateSystem`, which gather the related state classes as nested classes.

JViews TGO includes seven state systems:

- ◆ `IltOSI` is the gathering class for the OSI state dictionary. The classes nested in `IltOSI` are:
 - `IltOSI.State` This class implements the OSI primary states. Its instances are triplets, with a member of each of:
 - `IltOSI.State.Operational`
 - `IltOSI.State.Usage`
 - `IltOSI.State.Administrative`
 - `IltOSI.Procedural`, `IltOSI.Availability`, `IltOSI.Control`, `IltOSI.Standby`, and `IltOSI.Repair` These classes implement the OSI secondary states.
- ◆ `IltBellcore` is the gathering class for the Bellcore state dictionary. The classes nested in `IltBellcore` are:
 - `IltBellcore.State` This class implements the Bellcore primary states.
 - `IltBellcore.SecState` This class implements the Bellcore secondary states.
- ◆ `IltSNMP` is the gathering class for the SNMP state dictionary. The classes nested in `IltSNMP` are:
 - `IltSNMP.State` This class implements the SNMP primary state and its values.
 - `IltSNMP.Interface` This class contains the definition of the MIB-II Interface Group states.
 - `IltSNMP.IP` This class contains the definition of the states present in the MIB-II IP Group.
 - `IltSNMP.TCP` This class contains the definition of the states present in the MIB-II TCP Group.
 - `IltSNMP.UDP` This class contains the definition of the states present in the MIB-II UDP Group.
 - `IltSNMP.EGP` This class contains the definition of the states present in the MIB-II EGP Group.

- `IltSNMP.ICMP` This class contains the definition of the states present in the MIB-II ICMP Group.
- `IltSNMP.SNMP` This class contains the definition of the states present in the MIB-II SNMP Group.
- `IltSNMP.System` This class models the MIB-II System Group.
- ◆ `IltSONET` is the gathering class for the SONET state dictionary. The classes nested in `IltSONET` are:
 - `IltSONET.State` This class implements the SONET primary states.
 - `IltSONET.Protection` This class implements the SONET secondary states, also called protection switch request indicators.
- ◆ `IltMisc` is the gathering class for the Misc state dictionary. The only class nested in `IltMisc` is:
 - `IltMisc.SecState` This class implements the additional secondary states brought in by the Misc state dictionary.
- ◆ `IltPerformance` is the gathering class for the Performance state dictionary. The only class nested in `IltPerformance` is:
 - `IltPerformance.SecState` This class implements the additional secondary states brought in by the Performance state dictionary.
- ◆ `IltSAN` is the gathering class for the SAN state dictionary. The only class nested in `IltSAN` is:
 - `IltSAN.SecState` This class implements the additional secondary states brought in by the SAN state dictionary.
- ◆ `IltAlarm` is the gathering class for the Alarm state dictionary. The classes nested in `IltAlarm` are:
 - `IltAlarm.State` This class implements the alarm state of an object, with the number of new and acknowledged alarms of each severity, plus the Not Reporting and Loss Of Connectivity conditions.
 - `IltAlarm.Severity` This class is not a state class. It is an extensible enumeration implementing the raw alarm severities known to JViews TGO.
 - `IltAlarm.ImpactSeverity` This class is not a state class. It is an extensible enumeration implementing the impact alarm severities known to JViews TGO.
- ◆ `IltTrap` is the gathering class for the Trap state dictionary. The classes nested in `IltTrap` are:
 - `IltTrap.State` This class implements the trap state of an object, with the number of new and acknowledged traps for each type of trap.
 - `IltTrap.Type` This class represents the enumeration with the traps known to JViews TGO.

Composite state values, such as the OSI primary state triplet, or the Alarm collection of integers and Boolean values, are normal Java™ objects. All other state values are statically allocated Java objects implementing symbolic state values. They include:

- ◆ The operational, usage, and administrative components of the OSI primary state.
- ◆ The Bellcore, SONET, and SNMP primary states.
- ◆ The OSI, Bellcore, SNMP, SONET, Performance, and Misc secondary states.

The names of the static data members storing the state values are listed in the tables in *Lookup tables for state visuals*.

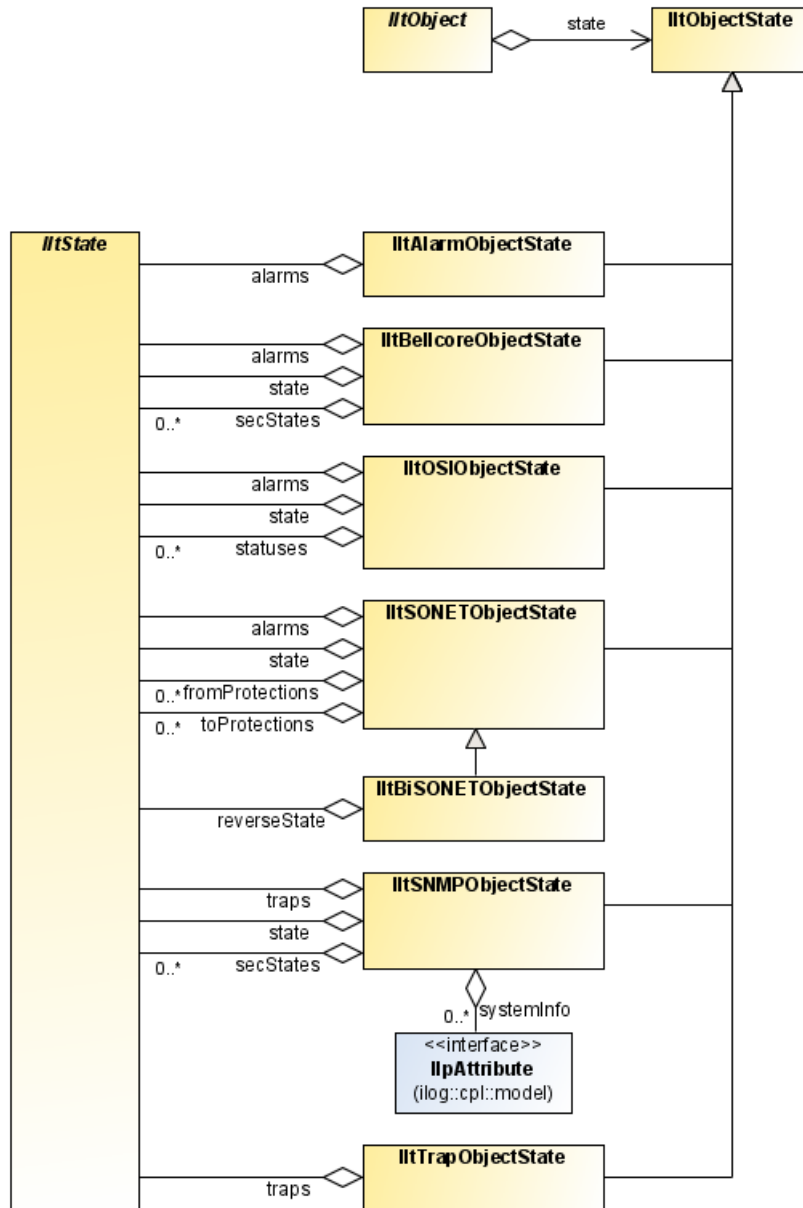
Object states

The state of a telecom object is described by an `IltObjectState` instance stored in the telecom object. This instance is called the *object state* of the telecom object. It contains state values corresponding to the primary state, the secondary states, and the alarm or trap state of the telecom object.

When the object state is modified to reflect a change in the telecom equipment state, the graphic representation of the telecom object is automatically recomputed and redisplayed.

Object state classes

There are seven subclasses of `IltObjectState` in JViews TGO. Subclasses are based on the OSI, Bellcore, SNMP and SONET state systems as illustrated in the figure that follows.



Object state classes

This figure shows that each type of object state can hold states and alarms. Alarms systematically complement states with the class `IltAlarmObjectState`, dedicated to the objects to which only alarms, and no states, are assigned. This figure also shows that each type of object state can hold states and traps. Traps systematically complement states with

the class `IltTrapObjectState`, dedicated to the objects to which only traps, and no states, are assigned.

Depending on the state systems (OSI, Bellcore, SNMP, SONET, Alarm, Trap, Misc) that compose the object state, state modeling is performed differently. For example, there are three state attributes in the SONET state system, whereas there are only two in the other state systems. The terminology employed, as well as the structure of the state models themselves, is developed in the next section.

Dictionaryes and object states

To choose the set of state dictionaryes used to describe the state and alarm condition of a telecom object, select a subclass of `IltObjectState` to store the object state. The seven subclasses of `IltObjectState` are listed below with the state dictionaryes that they gather.

- ◆ `IltOSIObjectState` OSI, SAN, Alarm, Performance, and Misc dictionaryes.
- ◆ `IltBellcoreObjectState` Bellcore, SAN, Alarm, Performance, and Misc dictionaryes.
- ◆ `IltSNMPObjectState` SNMP, SAN, Trap, Performance, and Misc dictionaryes.
- ◆ `IltSONETObjectState` and `IltBiSONETObjectState` SONET, SAN, Alarm, and Performance dictionaryes.
- ◆ `IltAlarmObjectState` Alarm dictionary only.
- ◆ `IltTrapObjectState` Trap dictionary only.

The object state system is supplied as an argument to the constructor of the telecom business object, as in the following code example.

How to supply the state system to the business object

```
IltNetworkElement berlin =
    new IltNetworkElement
        ("Berlin",
         IltNetworkElement.Type.NE,
         IltNetworkElement.Function.TransportCrossConnect,
         IltNetworkElement.Family.OC12,
         new IltBellcoreObjectState());
```

The object state classes

The OSI-based object state class

`IltOSIObjectState` is an object state class that provides an OSI primary state, a set of secondary states to be taken from the OSI, Misc, and Performance state dictionaries, and an alarm state from the Alarm state dictionary.

An instance of `IltOSIObjectState` thus holds:

- ◆ The object primary state as an instance of `IltOSI.State`
- ◆ The object `IltOSI.Control` secondary states as a list of state values which can be instances of `IltOSI.Procedural`, `IltOSI.Availability`, `IltOSI.Control`, `IltOSI.Standby`, `IltOSI.Repair`, `IltSAN.SecState`, `IltMisc.SecState`, or `IltPerformance.SecState`.
- ◆ The object alarms as an instance of `IltAlarm.State`.

The Bellcore-based object state class

`IltBellcoreObjectState` is an object state class that provides a Bellcore primary state, a set of secondary states to be taken from the Bellcore, Performance, and Misc state dictionaries, and an alarm state from the Alarm state dictionary.

An instance of `IltBellcoreObjectState` thus holds:

- ◆ The object primary state as an instance of `IltBellcore.State`
- ◆ The object secondary states as a list of state values that can be instances of `IltBellcore.SecState`, `IltSAN.SecState`, `IltMisc.SecState`, or `IltPerformance.SecState`
- ◆ The alarms of the object as an instance of `IltAlarm.State`.

The SNMP-based object state class

`IltSNMPObjectState` is an object state class that provides an SNMP primary state, a set of secondary states to be taken from the SNMP, Performance and Misc state dictionaries, and a trap state from the Trap state dictionary.

An instance of `IltSNMPObjectState` thus holds:

- ◆ The object primary state as an instance of `IltSNMP.State`
- ◆ The object secondary states as a list of state values that can be instances of `IltLimitedNumericState`, `IltSAN.SecState`, `IltMisc.SecState` or `IltPerformance.SecState`.
- ◆ The alarms of the object as an instance of `IltTrap.State`.

The SONET-based object state class

`IltSONETObjectState` and `IltBiSONETObjectState` are object state classes that provide one or two SONET primary states respectively, two sets of protection switch request indicators, a secondary state taken from the Performance state dictionary if needed and one alarm state. They are dedicated to links, typically links in SONET rings.

An instance of `IltSONETObjectState` thus holds:

- ◆ The link primary state as an instance of `IltSONET.State`
- ◆ The protection switch request indicators that have been set on one end of the link, as a list of instances of `IltSONET.Protection`
- ◆ The protection switch request indicators that have been set on the other end of the link, as a second list of instances of `IltSONET.Protection`
- ◆ The object secondary states as a list of state values that are instances of `IltSAN.SecState` or `IltPerformance.SecState`.
- ◆ The link alarms as an instance of `IltAlarm.State`

An instance of `IltBiSONETObjectState` holds:

- ◆ A second instance of `IltSONET.State`, to store the state of the link in the reverse direction.

On a link with a BiSONET object state, the inherited primary state is considered to be the state of the direct direction and is graphically represented by the inner part of the link base and the “to” arrow. The reverse state is represented by the outer part of the link base and the “from” arrow.

The Alarm-only object state class

`IltAlarmObjectState` is an object state class that provides only an alarm state from the Alarm state dictionary. It is intended for objects with no states.

An instance of `IltAlarmObjectState` thus holds only an instance of `IltAlarm.State`.

The Trap-only object state class

`IltTrapObjectState` is an object state class that provides only a trap state from the Trap state dictionary. It is intended for objects with no states.

An instance of `IltTrapObjectState` thus holds only an instance of `IltTrap.State`.

Modifying states and statuses

Setting a network element state in the OSI state system

The following code shows how to modify the usage state of the `neast` node and set it to the active value:

```
neast.setState(IltOSI.State.Usage.Active);
```

The `Active` symbol is the name of a global variable that contains a state value. This value is an instance of the `IltOSI.State.Usage` class (which is itself an `IltState` subclass).

Setting network element states in the Bellcore state system

The following code fragment shows how to change the primary state of the `berlin` node to `EnabledActive` and how to set the `Busy` secondary state on this node:

```
berlin.setState(IltBellcore.State.EnabledActive);  
berlin.setState(IltBellcore.SecState.Busy);
```

Note that the use of the functional interface is the same for all the telecom object classes and the state systems.

Setting network element states in the SNMP state system

The following code fragment shows how to change the primary state of the `berlin` node to `Shutdown` and how to set the `Interface.InOctets` secondary state on this node. This code is an extract from the `snmp` sample in the distribution.

How to change the primary state of a node and set the secondary state

```
berlin.setState(IltSNMP.State.Shutdown);  
berlin.setState(IltMisc.SecState.TestFailed);  
berlin.set(IltSNMP.Interface.InOctets, new Integer(123));
```

In the SNMP state dictionary, most of the states have numeric values. In order to set the values for these states you can use either `IltSNMPObjectState` or `IltObject`. Both of these APIs provide methods to set and retrieve the value of a numeric state.

- ◆ `set(ilog.tgo.model.IltState, java.lang.Object)` sets the value of the given state.
- ◆ `get(ilog.tgo.model.IltState)` returns the value of the given state.

Besides the numeric and Boolean states present in the SNMP state dictionary, there is also a set of information that is related to the system being managed and that is defined in the MIB-II System Group. The information in the group is displayed in a `System` window (explained in detail in *System window*). The graphical representation of this information is

an icon that, when clicked, opens the System window containing the list of all the attributes set in the telecom object. For information regarding the insertion of new attributes, see [Creating a new attribute in the System group](#).

The following code extract shows how to change the values of the attributes defined by the System Group.

How to change the attribute values defined by the system group

```
IltSNMPObjectState objstate = (IltSNMPObjectState) berlin.getObjectState();
IltSNMP.SystemInfo sysinfo = objstate.getSystemInfo();
Sysinfo.setDescription ("Berlin station");
Sysinfo.setContact ("John Doe");
```

Setting link states

In this example, we modify the `berlin_west` link, which uses the SONET state system.

How to set link states

```
berlin_west.setState(IltSONET.State.Active);
```

Accessing and removing states

In addition to the API used to assert states, another API is available for accessing and removing states. Most of the functions for accessing states are available from the class `IltObject`.

For example, the Boolean predicate `hasState` can be used to verify whether the telecom object is in a given state. In all the following examples, `neast` is the variable that contains the network element North East.

```
neast.hasState(IltOSI.State.Usage.Idle);
```

Sometimes, specific accesses are granted using the functional interfaces on the classes `IltObjectState` and `IltState`. The following code shows how to access the state object and then set the accessors for retrieving a given state.

How to access a state object and set accessors for retrieving a state

```
IltOSIObjectState osiObjectState =  
    (IltOSIObjectState)neast.getObjectState();  
IltOSI.State osiState = osiObjectState.getState();  
osiState.getOperationalState();  
osiState.getUsageState();  
osiState.getAdministrativeState();
```

When primary or secondary states are added, the same functional interface is used, but with different consequences. The primary state is unique (in the OSI state system, it is a triplet; in the Bellcore state system, it is a single value), as opposed to secondary states (for example, the values associated with the Misc dictionary can be added without limitation). Consequently, when the member function `setState` is used, this function *replaces* the primary state, although it adds one more secondary state. There are two ways to remove states created in this way:

- ◆ You can use the `clearState` member function, which removes secondary states *or* resets the primary state to its default value.
- ◆ You can use the `resetState` member function, which removes all the secondary states *and* resets the primary state to its default value.

Defining states in XML

Explains how to define object states in the XML format and load them in a data source.

In this section

Overview

Explains the use of a data source to load objects with states defined in XML.

OSI states

Describes the XML elements that you can use in the OSI state system.

Belcore states

Describes the XML elements that you can use in the Bellcore state system.

SNMP states

Describes the XML elements that you can use in the SNMP state system.

Miscellaneous states

Describes the XML elements that you can use in the Miscellaneous state system.

Performance states

Describes the XML elements that you can use to read and write Performance states.

SAN states

Describes the XML elements that you can use in the SAN state system.

SONET states

Describes the XML elements that you can use in the SONET state system.

BiSONET states

Describes the XML elements that you can use in the BiSONET state system.

Alarm states

Describes the XML elements that you can use in the Alarm state system.

Trap states

Describes the XML elements that you can use in the Trap state system.

Overview

To load objects with states defined in the XML format, all you have to do is create a data source using the data source default implementation defined by `IltDefaultDataSource` and pass the XML file containing the object description to the `parse` method of the data source, as shown below:

```
dataSource = new IltDefaultDataSource();
dataSource.parse("OSIXMLFile.xml");
```

To define states when you add predefined business objects to the data source, set the value of attribute `objectState` to one of the JViews TGO object states:

- ◆ `IltOSIObjectState`
- ◆ `IltBellcoreObjectState`
- ◆ `IltSNMPObjectState`
- ◆ `IltSONETObjectState`
- ◆ `IltBiSONETObjectState`
- ◆ `IltAlarmObjectState`
- ◆ `IltTrapObjectState`

How to create a predefined business object with states and alarms using XML

The following example shows an XML extract that you can load in a data source to create a rectangular group. States and alarms can be set in this object by declaring the value of attribute `objectState` when adding the business object:

```
<addObject id="RectGroup">
  <class>ilog.tgo.model.IltRectGroup</class>
  <attribute name="name">RectGroup</attribute>
  <attribute name="position" javaClass="ilog.cpl.graphic.IlpRect">
    <x>489</x> <y>356</y> <width>80</width> <height>60</height>
  </attribute>
  <attribute name="objectState" javaClass="ilog.tgo.model.IltOSIObjectState">

    <state>
      <administrative>Locked</administrative>
      <operational>Enabled</operational>
      <usage>Idle</usage>
    </state>
    <availability>PowerOff</availability>
    <control>ReservedForTest</control>
    <alarms>
```

```

    <new severity="Raw.Critical">5</new>
    <ack severity="Raw.Warning">12</ack>
  </alarms>
</attribute>
</addObject>

```

How to set states and alarms to an existing object using XML

The following example shows how to set the states and alarms of an object that already exists in the data source. You can achieve this by using the XML tag `<updateObject>` to modify the attribute `objectState`:

```

<updateObject id="RectGroup">
  <attribute name="objectState" javaClass="ilog.tgo.model.IltOSIObjectState">
    <state>
      <administrative>ShuttingDown</administrative>
      <operational>Enabled</operational>
      <usage>Busy</usage>
    </state>
    <availability>PowerOff</availability>
    <misc>HighTemperatureWarning</misc>
    <alarms>
      <new severity="Raw.Critical">2</new>
    </alarms>
  </attribute>
</updateObject>

```

The `<attribute>` tag is used to modify the value of an attribute in a business object. When you use it to modify the value of attribute `objectState`, the old object state definition is completely replaced by the new one.

JViews TGO also provides support to update the object state of a predefined business object to allow you to perform incremental changes. This support is made available through the tag `<updateState>`:

Within an `<updateState>` block, you can modify the states, alarms and traps present in the business object state, adding and removing states, adding, removing and setting alarms and traps.

How to update states and alarms incrementally using XML

```

<updateObject id="NE1">
  <updateState>
    <state>
      <operational>Enabled</operational>
      <usage>Active</usage>
    </state>
    <procedural operation="remove">Initializing</procedural>
    <misc operation="remove">DoorAjar</misc>
    <alarms>

```

```
<new severity="Raw.Critical" operation="set">2</new>
<new severity="Raw.Major" operation="add">1</new>
<ack severity="Raw.Warning" operation="remove">5</ack>
</alarms>
</updateState>
</updateObject>
```

The tables in this section list, for each state system, the XML elements that you can use to describe states in the XML format.

OSI states

The class `ILtOSIObjectStateSAXInfo` is the XML serialization class that allows you to read and write OSI object states in the XML format.

The following table describes the XML elements that can be used.

XML elements in the OSI state system

XML Element	Attributes	Possible Values	Description
<administrative>		Locked, Unlocked, ShuttingDown	OSI Administrative state
	operation	add, remove	This attribute is optional. It specifies whether the state should be added to/removed from the object state. The default value is <code>add</code> indicating that the state will be set in the object state. This attribute is used within an <updateState> element.
<operational>		Enabled, Disabled	OSI Operational state
	operation	add, remove	This attribute is optional. It specifies whether the state should be added to/removed from the object state. The default value is <code>add</code> indicating that the state will be set in the object state. This attribute is used within an <updateState> element.
<usage>		Idle, Active, Busy	OSI Usage state
	operation	add, remove	This attribute is optional. It specifies whether the state should be added to/removed from the object state. The default value is <code>add</code> indicating that the state will be set in the object state. This attribute is used within an <updateState> element.
<procedural>		InitializationRequired, Initializing, Reporting, Terminating	OSI Procedural status
	operation	add, remove	This attribute is optional. It specifies whether the state should be added to/removed from the object state. The default value is <code>add</code> indicating that the state will be set in the object

XML Element	Attributes	Possible Values	Description
			state. This attribute is used within an <updateState> element.
<availability>		Degraded, Dependency, Failed, InTest, LogFull, NotInstalled, OffDuty, OffLine, PowerOff	OSI Availability status
	operation	add, remove	This attribute is optional. It specifies whether the state should be added to/removed from the object state. The default value is add indicating that the state will be set in the object state. This attribute is used within an <updateState> element.
<control>		PartOfServicesLocked, ReservedForTest, SubjectToTest, Suspended	OSI Control status
	operation	add, remove	This attribute is optional. It specifies whether the state should be added to/removed from the object state. The default value is add indicating that the state will be set in the object state. This attribute is used within an <updateState> element.
<standby>		InStandby	OSI Stand-by status
	operation	add, remove	This attribute is optional. It specifies whether the state should be added to/removed from the object state. The default value is add indicating that the state will be set in the object state. This attribute is used within an <updateState> element.
<repair>		UnderRepair	OSI Repair status
	operation	add, remove	This attribute is optional. It specifies whether the state should be added to/removed from the object state. The default value is add indicating that the state will be set in the object

XML Element	Attributes	Possible Values	Description
			state. This attribute is used within an <updateState> element.

How to add a network element with OSI states defined in XML

```
<addObject id="NE1">
  <class>ilog.tgo.model.IltNetworkElement</class>
  <attribute name="name">NE1</attribute>
  <attribute name="family">OC12</attribute>
  <attribute name="type">MD</attribute>
  <attribute name="position" javaClass="ilog.cpl.graphic.IlpPoint">
    <x>68</x> <y>61</y>
  </attribute>
  <attribute name="objectState" javaClass="ilog.tgo.model.IltOSIObjectState">
    <state>
      <administrative>ShuttingDown</administrative>
      <operational>Enabled</operational>
      <usage>Active</usage>
    </state>
    <procedural>Reporting</procedural>
    <repair>UnderRepair</repair>
  </attribute>
</addObject>
```

How to set OSI states to an existing object using XML

The following example shows how to set OSI states to an object that already exists in the data source. You can achieve this by using the XML tag <updateObject> to modify the attribute objectState:

```
<updateObject id="RectGroup">
  <attribute name="objectState" javaClass="ilog.tgo.model.IltOSIObjectState">
    <state>
      <administrative>ShuttingDown</administrative>
      <operational>Enabled</operational>
      <usage>Busy</usage>
    </state>
    <availability>PowerOff</availability>
    <procedural>Terminating</procedural>
  </attribute>
</updateObject>
```

How to update OSI states incrementally using XML

```
<updateObject id="RectGroup">
  <updateState>
    <state>
      <operational>Enabled</operational>
      <usage>Active</usage>
    </state>
    <procedural operation="remove">Initializing</procedural>
    <availability>InTest</availability>
  </updateState>
</updateObject>
```

Bellcore states

The class `IlBellcoreObjectStateSAXInfo` is the XML serialization class that allows you to read and write Bellcore object states in the XML format.

The following table describes the XML elements that can be used.

XML elements in the Bellcore state system

XML Element	Attributes	Possible Values	Description
<state>	None	DisabledIdle, EnabledIdle, EnabledActive	BellCore primary state
<secState>		Blocked, Busy, ColdStandby, Combined, Diagnostic, Disabled, Disconnected, Exercise, FacilityFailure, FarEndProcessorOutage, Fault, Forced, HotStandby, Idle, IdleReceive, IdleTransmit, InhibitInProgress, LockedOut, LoopbackTest, Maintenance, MaintenanceLimited, MismatchOfEquipmen, Monitor, Overflow, PerformanceMonitorInhibited, Power, PrePostService, ProtectionReleaseInhibited, ProtectionSwitchExercise, ProtectionSwitchInhibited, Protocol, Rearrangement, RedLined, SoftwareDownloading, SoftwareTransferInhibited, SoftwareTransferOnly, SoftwareUploading, StandbyInhibited, StandbySwitched, SupportedEntityAbsent, SupportedEntityExists, SupportingEntityOutage, SupportingEntitySwapped, SuspendBoth, SuspendOrigination, SuspendTermination, SwitchedSystemActivity, TerminatedBoth, TerminatedFrom, TerminatedTo, Test, TestFailure, Transferred, Unassigned, Unequipped, Working	BellCore secondary states
	operation	add, remove	This attribute is optional. It specifies whether the state should be added

XML Element	Attributes	Possible Values	Description
			to/removed from the object state. The default value is add indicating that the state will be set in the object state. This attribute is used within an <updateState> element.

How to add a network element with Bellcore states defined in XML

```
<addObject id="NE3">
  <class>ilog.tgo.model.IltNetworkElement</class>
  <attribute name="name">NE3</attribute>
  <attribute name="type">Router</attribute>
  <attribute name="position" javaClass="ilog.cpl.graphic.IlpPoint">
    <x>151</x> <y>512</y>
  </attribute>
  <attribute name="objectState"
    javaClass="ilog.tgo.model.IltBellcoreObjectState">
    <state>EnabledActive</state>
    <secState>TestFailure</secState>
  </attribute>
</addObject>
```

How to set Bellcore states to an existing object using XML

The following example shows how to set Bellcore states to an object that already exists in the data source. You can achieve this by using the XML tag <updateObject> to modify the attribute objectState:

```
<updateObject id="NE1">
  <attribute name="objectState"
    javaClass="ilog.tgo.model.IltBellcoreObjectState">
    <state>EnabledActive</state>
    <secState>Blocked</secState>
    <secState>Busy</secState>
  </attribute>
</updateObject>
```

How to update Bellcore states incrementally using XML

```
<updateObject id="NE1">
  <updateState>
```

```
<state>EnabledIdle</state>  
  <secState operation="remove">Blocked</secState>  
  <secState operation="remove">Busy</secState>  
  <secState>HotStandby</secState>  
</updateState>  
</updateObject>
```

SNMP states

The class `ILT SNMPObjectStateSAXInfo` is the XML serialization class that allows you to read and write SNMP object states in the XML format.

The following table describes the XML elements that can be used.

XML elements in the SNMP state system

XML Element	Attributes	Possible Values	Description
<state>	None	Down, Failed, Shutdown, Testing, Up	SNMP primary state
<interface>	state	InOctets, InUcastPkts, InDiscards, InErrors, InUnknownProtos, OutOctets, OutUcastPkts, OutNUcastPkts, OutDiscards, OutErrors	This attribute is mandatory. It defines the secondary state that will be set.
	isArray	true or false	This attribute is optional. It defines whether the state value is an array or not.
	value	java.lang.Float	This attribute is optional. When describing array values, each value in the array is enclosed in a <value> element.
	javaClass		This attribute is optional. It defines the value of the state. The default value is <code>java.lang.Float</code> .
	operation	add, remove	This attribute is optional. It specifies whether the state should be added to/removed from the object state. The default value is <code>add</code> indicating that the state will be set in the object state.

XML Element	Attributes	Possible Values	Description
			This attribute is used within an <updateState> element.
<ip>	state	InReceives, InHdrErrors, InAddrError, ForwDatagrams, InUnknownProtos, InDiscards, InDelivers, OutRequests, OutDiscards, OutNoRoutes, Forwarding	See <interface>
	isArray	true or false	See <interface>
	value	java.lang.Float	See <interface>
	javaClass		See <interface>
	operation	add, remove	This attribute is optional. It specifies whether the state should be added to/removed from the object state. The default value is add indicating that the state will be set in the object state. This attribute is used within an <updateState> element.
<icmp>	state	InMsgs, InErrors, OutMsgs, OutErrors	See <interface>
	isArray	true or false	See <interface>
	value	java.lang.Float	See <interface>
	javaClass		See <interface>
	operation	add, remove	This attribute is optional. It specifies whether the state should be added to/removed from the object state. The default value is add indicating that the state will be set in the object state. This attribute is used within an <updateState> element.
<tcp>	state	CurrentEstablished, InSegs, OutSegs, InErrors, RetranSegs	See <interface>
	isArray	true or false	See <interface>
	value	java.lang.Float	See <interface>
	javaClass		See <interface>
	operation	add, remove	This attribute is optional. It specifies whether the state should be added to/removed from the object state. The

XML Element	Attributes	Possible Values	Description
			default value is <code>add</code> indicating that the state will be set in the object state. This attribute is used within an <code><updateState></code> element.
<code><udp></code>	state	<code>InDatagrams, InErrors, OutDatagrams</code>	See <code><interface></code>
	<code>isArray</code>	<code>true</code> or <code>false</code>	See <code><interface></code>
	<code>value</code>	<code>java.lang.Float</code>	See <code><interface></code>
	<code>javaClass</code>		See <code><interface></code>
	<code>operation</code>	<code>add, remove</code>	This attribute is optional. It specifies whether the state should be added to/removed from the object state. The default value is <code>add</code> indicating that the state will be set in the object state. This attribute is used within an <code><updateState></code> element.
<code><egp></code>	state	<code>InMsgs, InErrors, OutMsgs, OutErrors</code>	See <code><interface></code>
	<code>isArray</code>	<code>true</code> or <code>false</code>	See <code><interface></code>
	<code>value</code>	<code>java.lang.Float</code>	See <code><interface></code>
	<code>javaClass</code>		See <code><interface></code>
	<code>operation</code>	<code>add, remove</code>	This attribute is optional. It specifies whether the state should be added to/removed from the object state. The default value is <code>add</code> indicating that the state will be set in the object state. This attribute is used within an <code><updateState></code> element.
<code><snmp></code>	state	<code>InPkts, OutPkts</code>	See <code><interface></code>
	<code>isArray</code>	<code>true</code> or <code>false</code>	See <code><interface></code>
	<code>value</code>	<code>java.lang.Float</code>	See <code><interface></code>
	<code>javaClass</code>		See <code><interface></code>
	<code>operation</code>	<code>add, remove</code>	This attribute is optional. It specifies whether the state should be added to/removed from the object state. The default value is <code>add</code> indicating that the state will be set in the object state.

XML Element	Attributes	Possible Values	Description
			This attribute is used within an <code><updateState></code> element.
<code><system></code>	None		Defines the system information for the object. It can be used to define the system attributes, such as location, description, contact.

How to add a group with SNMP states defined in XML

```

<addObject id="RectGroup">
  <class>ilog.tgo.model.IltRectGroup</class>
  <attribute name="name">RectGroup</attribute>
  <attribute name="position" javaClass="ilog.cpl.graphic.IlpRect">
    <x>489</x> <y>356</y> <width>80</width> <height>60</height>
  </attribute>
  <attribute name="objectState" javaClass="ilog.tgo.model.IltSNMPObjectState">

    <state>Down</state>
    <ip state="InDiscards">50</ip>
    <ip state="Forwarding" javaClass="java.lang.Boolean">true</ip>
    <interface state="InOctets" isArray="true" javaClass="java.lang.Integer">

      <value>50</value>
      <value>70</value>
      <value>58</value>
      <value>60</value>
      <value>58</value>
      <value>62</value>
    </interface>
  </attribute>
</addObject>

```

How to set SNMP states to an existing object using XML

The following example shows how to set SNMP states to an object that already exists in the data source. You can achieve this by using the XML tag `<updateObject>` to modify the attribute `objectState`:

```

<updateObject id="NE1">
  <attribute name="objectState" javaClass="ilog.tgo.model.IltSNMPObjectState">

    <state>Up</state>
    <interface state="InOctets">100</interface>
    <ip state="InDiscards">50</ip>
    <system>
      <attribute name="location">San Francisco</attribute>
    </system>
  </attribute>
</updateObject>

```



```
    <attribute name="description">Test machine</attribute>
  </system>
</attribute>
</updateObject>
```

How to update SNMP states incrementally using XML

```
<updateObject id="NE1">
  <updateState>
    <state>Up</state>
    <interface state="InOctets">80</interface>
    <ip state="InDiscards" operation="remove"/>
    <system>
      <attribute name="location">Los Angeles</attribute>
      <attribute name="description" null="true"/>
    </system>
  </updateState>
</updateObject>
```

Miscellaneous states

The class `IltObjectStateSAXInfo` is the XML serialization class that allows you to read and write miscellaneous states in the XML format.

The following table describes the XML elements that can be used.

XML elements in the Miscellaneous state system

XML Element	Attributes	Possible Values	Description
<code><misc></code>		SoftwareUpload, SoftwareDownload, SoftwareLimitExceeded, MismatchedCard, UnknownCard, DoorAjar, LowTemperatureWarning, HighTemperatureWarning, TestPassed, TestFailed, ThresholdCrossing, PlanToRemove, UnderRepair	Miscellaneous state
	<code>operation</code>	add, remove	This attribute is optional. It specifies whether the state should be added to/removed from the object state. The default value is <code>add</code> indicating that the state will be set in the object state. This attribute is used within an <code><updateState></code> element.

How to add a network element with Miscellaneous states defined in XML

```
<addObject id="NE1">
  <class>ilog.tgo.model.IltNetworkElement</class>
  <attribute name="position" javaClass="ilog.cpl.graphic.IlpPoint">
    <x>68</x> <y>61</y>
  </attribute>
  <attribute name="objectState" javaClass="ilog.tgo.model.IltOSIObjectState">
    <state>
      <administrative>ShuttingDown</administrative>
      <operational>Enabled</operational>
      <usage>Active</usage>
    </state>
    <misc>LowTemperatureWarning</misc>
    <misc>DoorAjar</misc>
  </attribute>
</addObject>
```

How to set Miscellaneous states to an existing object using XML

The following example shows how to set miscellaneous states to an object that already exists in the data source. You can achieve this by using the XML tag `<updateObject>` to modify the attribute `objectState`:

```
<updateObject id="NE1">
  <attribute name="objectState"
  javaClass="ilog.tgo.model.IltBellcoreObjectState">
    <state>EnabledActive</state>
    <misc>DoorAjar</misc>
  </attribute>
</updateObject>
```

How to update Miscellaneous states incrementally using XML

```
<updateObject id="NE1">
  <updateState>
    <misc operation="remove">DoorAjar</misc>
    <misc>LowTemperatureWarning</misc>
  </updateState>
</updateObject>
```

Performance states

The class `IltObjectStatesSAXInfo` is the XML serialization class that allows you to read and write Performance states in the XML format.

The following table describes the XML elements that can be used.

XML elements in the Performance state system

XML Element	Attributes	Possible Values	Description
<code><performance></code>	<code>state</code>	<code>Input, In, In_Kb, In_Mb, In_Gb, Output, Out, Out_Kb, Out_Mb, Out_Gb, Print, Generic, Power, Temperature, Bandwidth</code>	SAN state
	<code>isArray</code>	<code>true or false</code>	This attribute is optional. It defines whether the state value is an array or not.
	<code>value</code>	<code>java.lang.Float</code>	This attribute is optional. When describing array values, each value in the array is enclosed in an element.
	<code>javaClass</code>		This attribute is optional. It defines the value of the state. The default value is <code>java.lang.Float</code> .
	<code>operation</code>	<code>add, remove</code>	This attribute is optional. It specifies whether the state should be added to/removed from the object state. The default value is <code>add</code> indicating that the state will be set in the object state. This attribute is used within an <code><updateState></code> element.

How to add a network element with Performance states defined in XML

```
<addObject id="NE1">
  <class>ilog.tgo.model.IltNetworkElement</class>
  <attribute name="position" javaClass="ilog.cpl.graphic.IlpPoint">
    <x>68</x> <y>61</y>
  </attribute>
  <attribute name="objectState" javaClass="ilog.tgo.model.IltOSIObjectState">
    <state>
      <administrative>ShuttingDown</administrative>
      <operational>Enabled</operational>
    </state>
  </attribute>
</addObject>
```

```

    <usage>Active</usage>
  </state>
  <performance state="In_Kb" isArray="true" javaClass="java.lang.Integer">
    <value>50</value>
    <value>70</value>
    <value>58</value>
    <value>60</value>
    <value>58</value>
    <value>62</value>
    <value>54</value>
    <value>24</value>
    <value>56</value>
    <value>85</value>
    <value>58</value>
    <value>65</value>
    <value>12</value>
    <value>35</value>
  </performance>
  <performance state="Input">200</performance>
</attribute>
</addObject>

```

How to set Performance states to an existing object using XML

The following example shows how to set Performance states to an object that already exists in the data source. You can achieve this by using the XML tag `<updateObject>` to modify the attribute `objectState`:

```

<updateObject id="NE1">
  <attribute name="objectState"
javaClass="ilog.tgo.model.IltBellcoreObjectState">
    <state>EnabledActive</state>
    <performance state="Input">200</performance>
  </attribute>
</updateObject>

```

How to update Performance states incrementally using XML

```

<updateObject id="NE1">
  <updateState>
    <performance state="Input" operation="remove"/>
    <performance state="Bandwidth">50</performance>
  </updateState>
</updateObject>

```

SAN states

The class `IltObjectStateSAXInfo` is the XML serialization class that allows you to read and write SAN states in the XML format.

The following table describes the XML elements that can be used.

XML elements in the SAN state system

XML Element	Attributes	Possible Values	Description
<SAN>	state	IO, Allocated, Available, BackRecovery, Bandwidth, Capacity, CapacityUtilization, CPU, DataAccessDelay, Fragmentation, LostData, Usage	SAN state
	isArray	true or false	This attribute is optional. It defines whether the state value is an array or not.
	value	java.lang.Float	This attribute is optional. When describing array values, each value in the array is enclosed in an element.
	javaClass		This attribute is optional. It defines the value of the state. The default value is <code>java.lang.Float</code> .
	operation	add, remove	This attribute is optional. It specifies whether the state should be added to/removed from the object state. The default value is <code>add</code> indicating that the state will be set in the object state. This attribute is used within an <code><updateState></code> element.

How to add a network element with SAN states defined in XML

```
<addObject id="NE1">
  <class>ilog.tgo.model.IltNetworkElement</class>
  <attribute name="position" javaClass="ilog.cpl.graphic.IlpPoint">
    <x>68</x> <y>61</y>
  </attribute>
  <attribute name="objectState" javaClass="ilog.tgo.model.IltOSIObjectState">
    <state>
      <administrative>ShuttingDown</administrative>
      <operational>Enabled</operational>
      <usage>Active</usage>
    </state>
  </attribute>
</addObject>
```

```
</state>
  <SAN state="Bandwidth" isArray="true" javaClass="java.lang.Integer">
    <value>50</value>
    <value>70</value>
    <value>30</value>
  </SAN>
</attribute>
</addObject>
```

How to set SAN states to an existing object using XML

The following example shows how to set SAN states to an object that already exists in the data source. You can achieve this by using the XML tag `<updateObject>` to modify the attribute `objectState`:

```
<updateObject id="NE1">
  <attribute name="objectState"
javaClass="ilog.tgo.model.IltBellcoreObjectState">
    <state>EnabledActive</state>
    <SAN state="Allocated">88</SAN>
  </attribute>
</updateObject>
```

How to update SAN states incrementally using XML

```
<updateObject id="NE1">
  <updateState>
    <SAN state="Allocated" operation="remove"/>
    <SAN state="Fragmentation">75</SAN>
  </updateState>
</updateObject>
```

SONET states

The class `IltSONETObjectStateSAXInfo` is the XML serialization class that allows you to read and write SONET states in the XML format.

The following table describes the XML elements that can be used:

XML elements in the SONET state system

XML Element	Attributes	Possible Values	Description
<code><state></code>	None	Disabled, Inactive, Active, ActiveProtecting, TroubledProtected, TroubledUnprotected	SONET primary state
<code><protection></code>		Exercisor, ForcedSwitch, Locked, ManualSwitch, Pending, WaitToRestore	SONET protection state
	from	true or false	This is an optional attribute. It defines whether the protection is set in the "from" end point of the link. By default, the value is true.
	to	true or false	This is an optional attribute. It defines whether the protection is set in the "to" end point of the link. By default, the value is true.

How to add a link with a SONET state defined in XML

```
<addObject id="Link1">
  <class>ilog.tgo.model.IltLink</class>
  <link> <from>PolyGroup</from> <to>NE3</to> </link>
  <attribute name="name">Link1</attribute>
  <attribute name="media">Fiber</attribute>
  <attribute name="objectState" javaClass="ilog.tgo.model.IltSONETObjectState">
    <state>Active</state>
    <protection from="true">Exercisor</protection>
    <protection to="true">Locked</protection>
  </attribute>
</addObject>
```

How to set SONET states to an existing object using XML

The following example shows how to set SONET states to an object that already exists in the data source. You can achieve this by using the XML tag `<updateObject>` to modify the attribute `objectState`:


```
<updateObject id="Link1">
  <attribute name="objectState" javaClass="ilog.tgo.model.IltSONETObjectState">
    <state>Active</state>
    <protection from="true" to="false">Exercisor</protection>
    <protection>Locked</protection>
  </attribute>
</updateObject>
```

How to update SONET states incrementally using XML

```
<updateObject id="Link1">
  <updateState>
    <protection from="false" to="true">Exercisor</protection>
    <protection from="false" to="false">Locked</protection>
  </updateState>
</updateObject>
```

BiSONET states

The class `IltBiSONETObjectStateSAXInfo` is the XML serialization class that allows you to read and write BiSONET states in the XML format.

The following table describes the XML elements that can be used:

XML elements in the BiSONET state system

XML Element	Attributes	Possible Values	Description
<code><state></code>	None	Disabled, Inactive, Active, ActiveProtecting, TroubledProtected, TroubledUnprotected	BiSONET primary state
<code><reverseState></code>	None	Disabled, Inactive, Active, ActiveProtecting, TroubledProtected, TroubledUnprotected	BiSONET reverse state
<code><protection></code>		Exercisor, ForcedSwitch, Locked, ManualSwitch, Pending, WaitToRestore	BiSONET protection state
	from	true or false	This is an optional attribute. It defines whether the protection is set in the "from" end point of the link. By default, the value is <code>true</code> .
	to	true or false	This is an optional attribute. It defines whether the protection is set in the "to" end point of the link. By default, the value is <code>true</code> .

How to add a link with a BiSONET state defined in XML

```
<addObject id="Link1">
  <class>ilog.tgo.model.IltLink</class>
  <link> <from>PolyGroup</from> <to>NE3</to> </link>
  <attribute name="name">Link1</attribute>
  <attribute name="media">Fiber</attribute>
  <attribute name="objectState"
    javaClass="ilog.tgo.model.IltBiSONETObjectState">
    <state>Active</state>
    <reverseState>ActiveProtecting</reverseState>
    <protection from="true">Exercisor</protection>
    <protection to="true">Locked</protection>
  </attribute>
</addObject>
```

How to set BiSONET states to an existing object using XML

The following example shows how to set BiSONET states to an object that already exists in the data source. You can achieve this by using the XML tag `<updateObject>` to modify the attribute `objectState`:

```
<updateObject id="Link1">
  <attribute name="objectState"
  javaClass="ilog.tgo.model.IltBiSONETObjectState">
    <state>Active</state>
    <reverseState>ActiveProtecting</reverseState>
    <protection from="true" to="false">Exercisor</protection>
    <protection>Locked</protection>
  </attribute>
</updateObject>
```

How to update BiSONET states incrementally using XML

```
<updateObject id="Link1">
  <updateState>
    <reverseState>TroubledProtected</reverseState>
    <protection from="false" to="true">Exercisor</protection>
    <protection from="false" to="false">Locked</protection>
  </updateState>
</updateObject>
```

Alarm states

The class `IttAlarmObjectStateSAXInfo` is the XML serialization class that allows you to read and write alarm states in the XML format.

The following table describes the XML elements that can be used.

XML elements in the Alarm state system

XML Element	Attributes	Possible Values	Description
<alarms>			Delimits alarm state definition
	<code>notReporting</code>	<code>true</code> or <code>false</code>	This is an optional attribute. It defines whether the managed object is currently not reporting its alarm state.
	<code>lossOfConnectivity</code>	<code>true</code> or <code>false</code>	This is an optional attribute. It defines whether the managed object is currently without connectivity.
<new>			Structural element that lets you specify new alarms for an object
	<code>severity</code>	<code>Raw.Critical</code> , <code>Raw.Major</code> , <code>Raw.Minor</code> , <code>Raw.Warning</code> or <code>Raw.Unknown</code> <code>Impact.CriticalHigh</code> , <code>Impact.CriticalLow</code> , <code>Impact.MajorHigh</code> , <code>Impact.MajorLow</code> , <code>Impact.MinorHigh</code> , <code>Impact.MinorLow</code> , <code>Impact.WarningHigh</code> , <code>Impact.WarningLow</code> or <code>Impact.Unknown</code>	This attribute is mandatory. It defines the severity of the new alarm to be set.
	<code>operation</code>	<code>set</code> , <code>add</code> , <code>remove</code>	This attribute is optional. It specifies whether the alarm should be set, added to or removed from the object state. The default value is <code>set</code> indicating that the alarm will be set in the object state. This attribute is

XML Element	Attributes	Possible Values	Description
			used within an <updateState> element.
<ack>			Structural element that lets you specify acknowledged alarms for an object
	severity	Raw.Critical, Raw.Major, Raw.Minor, Raw.Warning Or Raw.Unknown Impact.CriticalHigh, Impact.CriticalLow, Impact.MajorHigh, Impact.MajorLow, Impact.MinorHigh, Impact.MinorLow, Impact.WarningHigh, Impact.WarningLow Or Impact.Unknown	This attribute is mandatory. It defines the severity of the acknowledged alarm to be set.
	operation	set, add, remove	This attribute is optional. It specifies whether the alarm should be set, added to or removed from the object state. The default value is set indicating that the alarm will be set in the object state. This attribute is used within an <updateState> element.

How to add a network element and a link with alarms defined in XML

```

<addObject id="NE2">
  <class>ilog.tgo.model.IltNetworkElement</class>
  <attribute name="name">NE2</attribute>
  <attribute name="type">Mainframe</attribute>
  <attribute name="position" javaClass="ilog.cpl.graphic.IlpPoint">
    <x>229</x> <y>126</y>
  </attribute>
  <attribute name="objectState"
    javaClass="ilog.tgo.model.IltBellcoreObjectState">
    <state>EnabledActive</state>
    <alarms>
      <new severity="Raw.Critical">5</new>
      <ack severity="Impact.WarningLow">2</ack>
    </alarms>
  </attribute>
</addObject>

```

```

<addObject id="Link2">
  <class>ilog.tgo.model.IltLink</class>
  <link> <from>PolyGroup</from> <to>NE3</to> </link>
  <attribute name="name">Link2</attribute>
  <attribute name="media">Fiber</attribute>
  <attribute name="objectState" javaClass="ilog.tgo.model.IltSONETObjectState">

    <state>Active</state>
    <alarms notReporting="true" />
  </attribute>
</addObject>

```

How to set alarms to an existing object using XML

The following example shows how to set alarms to an object that already exists in the data source. You can achieve this by using the XML tag `<updateObject>` to modify the attribute `objectState`:

```

<updateObject id="NE1">
  <attribute name="objectState" javaClass="ilog.tgo.model.IltOSIObjectState">

    <state>
      <operational>Disabled</operational>
      <usage>Idle</usage>
      <administrative>Unlocked</administrative>
    </state>
    <alarms>
      <new severity="Raw.Critical">2</new>
      <new severity="Raw.Warning">1</new>
      <ack severity="Raw.Minor">3</ack>
    </alarms>
  </attribute>
</updateObject>

```

How to update alarms incrementally using XML

```

<updateObject id="NE1">
  <updateState>
    <alarms>
      <new severity="Raw.Critical" operation="remove">1</new>
      <new severity="Raw.Warning" operation="add">1</new>
      <ack severity="Raw.Critical" operation="add">1</ack>
      <ack severity="Raw.Minor" operation="remove">3</ack>
    </alarms>
  </updateState>
</updateObject>

```

Trap states

The class `IltTrapObjectStateSAXInfo` is the XML serialization class that allows you to read and write traps in the XML format.

The following table describes the XML elements that can be used.

XML elements in the Trap state system

XML Element	Attributes	Possible Values	Description
<traps>	None		Delimits trap state definition
<new>			Structural element that lets you specify new traps for an object
	type	ColdStart, WarmStart, LinkFailure, AuthenticationFailure, EGPNeighborLoss	This attribute is mandatory. It defines the type of the new trap to be set.
	operation	set, add, remove	This attribute is optional. It specifies whether the trap should be set, added to or removed from the object state. The default value is <code>set</code> indicating that the trap will be set in the object state. This attribute is used within an <code><updateState></code> element.
<ack>			Structural element that lets you specify acknowledged traps for an object
	type	ColdStart, WarmStart, LinkFailure, AuthenticationFailure, EGPNeighborLoss	This attribute is mandatory. It defines the severity of the acknowledged trap to be set.
	operation	set, add, remove	This attribute is optional. It specifies whether the trap should be set, added to or removed from the object state. The default value is <code>set</code> indicating that the trap will be set in the object state. This attribute is used within an <code><updateState></code> element.

How to add a network element with traps defined in XML

```
<addObject id="NE2">
  <class>ilog.tgo.model.IltNetworkElement</class>
  <attribute name="name">NE2</attribute>
  <attribute name="type">Mainframe</attribute>
  <attribute name="position" javaClass="ilog.cpl.graphic.IlpPoint">
    <x>229</x> <y>126</y>
  </attribute>
</addObject>
```

```
<attribute name="objectState" javaClass="ilog.tgo.model.IltSNMPObjectState">
  <state>Up</state>
  <traps>
    <new type="LinkFailure">5</new>
    <ack type="AuthenticationFailure">2</ack>
  </traps>
</attribute>
</addObject>
```

How to set traps to an existing object using XML

The following example shows how to set traps to an object that already exists in the data source. You can achieve this by using the XML tag `<updateObject>` to modify the attribute `objectState`:

```
<updateObject id="NE1">
  <attribute name="objectState" javaClass="ilog.tgo.model.IltSNMPObjectState">
    <state>Up</state>
    <traps>
      <new type="LinkFailure">2</new>
      <ack type="AuthenticationFailure">3</ack>
    </traps>
  </attribute>
</updateObject>
```

How to update traps incrementally using XML

```
<updateObject id="NE1">
  <updateState>
    <traps>
      <new type="ColdStart">1</new>
      <new type="LinkFailure" operation="remove">1</new>
      <new type="AuthenticationFailure" operation="add">1</new>
      <ack severity="LinkFailure" operation="add">1</ack>
      <ack severity="AuthenticationFailure" operation="remove">3</ack>
    </traps>
  </updateState>
</updateObject>
```


Information window

In most cases, secondary states are displayed as small icons in the top left corner of the base or plinth of the telecom object graphic.

When an object holds several such secondary states, they are represented as follows:

- ◆ Two secondary state icons can be displayed simultaneously as in the figure below:

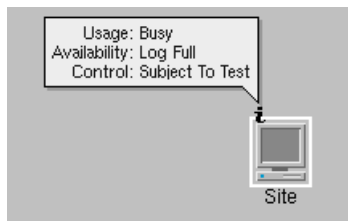


- ◆ When three or more secondary states are to be displayed, an Information icon replaces the secondary state icons as shown in the figure below:



- ◆ This icon provides an interactor, used to expand an information box that provides a list of existing secondary states when you click with the left mouse button (see the following figure):

Note: This interactor is an object interactor. As such it is active only if the view interactor set to the view delegates events to object interactors. To have a view interactor delegate event processing to an object interactor, use the method `setUsingObjectInteractor(boolean)` of the class `IlpViewInteractor`.



- ◆ Once the Information window is displayed, the rectangle that encompasses the Information icon becomes transparent and remains so until any new secondary state change is notified

to the telecom object. After the Information window has been closed (by clicking on the Information icon with the left mouse button), the Information icon looks like this:



The display of a collection of secondary states is replaced with the display of an information icon when there are more than a threshold number of icons; the default is two icons. The information window representation as well as the threshold number of icons can be customized through CSS. For details on using CSS to customize information windows, refer to .

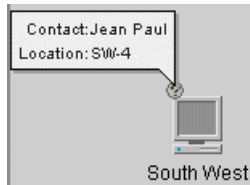
System window

In the SNMP State Dictionary there is a group called System, which is responsible for storing information about the system being managed. This group is based on the MIB-II System Group that contains attributes such as description, location, and contact. The System Window represents this information graphically, listing the contents of the attributes in a textual format.

- ◆ When there are attributes present in the object, the System icon is present as shown in the figure below:



- ◆ This icon provides an interactor, used to expand a System window that provides a list of existing attributes when you click the icon (see the following figure):



Note: This interactor is an object interactor. As such, it is active only if the view interactor set to the view delegates events to object interactors. To have a view interactor delegate event processing to an object interactor, use the method `setUsingObjectInteractor(boolean)` of the class `IlpViewInteractor`.

- ◆ Once the System window is displayed, the System icon background becomes gray and remains so until notification of an attribute change is sent to the telecom object. After the System window has been closed (by clicking the System icon), the System icon looks like this:



You can customize the way the System attributes are presented in the System Window, as well as add new attributes. For more information, see [Adding a user-defined business attribute to the system window](#).

Customizing the representation of states and alarms

For information on how to customize the graphic representation of states and alarms, refer to Customizing object states.

Index

A

- alarms **25**
 - class **264**
 - color coding **271**
 - count **343**
 - creating with the API **269, 350**
 - customizing **271, 404**
 - defined in XML **267, 396**
 - graphical cues **342**
 - in a table **271**
 - in a tree **271**
 - in subnetworks **187**
 - loading **267**
 - object **350**
 - properties **26**
 - representation **187, 271**
 - severity **343**
 - states **342, 343**

B

- Bellcore
 - graphical representations **284**
 - in XML **378**
 - primary states **284**
 - secondary states **404**
- Bellcore State Dictionary **333**
 - visuals **284**
- BISONETObjectState class **169**
- BTS (Base Transceiver Station) **23**
 - antennas **250**
 - class **250**
 - creating with the API **254**
 - customizing **256**
 - defined in XML **252**
 - equipment **251**
 - loading **252**
 - representation in a table **256**
 - representation in a tree **256**

- business class manager **66**
- business classes **11, 44**
 - predefined **30**
- business model **44**
 - API **58**
 - defining in XML **48**
 - dynamic classes **48, 68, 72**
 - integrating **45**
 - loading **56**
- business objects **44**
 - adding **97, 98**
 - defining **101**
 - removing **99**

C

- card carriers **218**
 - class **219**
 - creating with the API **222**
 - defined in XML **220**
 - definition **218**
 - loading **220**
- card items **226**
 - class **227**
- cards **20, 202**
 - class **203**
 - creating with the API **206**
 - customizing **248**
 - defined in XML **204**
 - loading **204**
 - representation in a table **247**
 - representation in a tree **247**
- cluster network element **122**
- complex types **51**

D

- data sources **12**
 - API **82**
 - batches **106**
 - implementing **110**

parsing **108**
writing **108**

E

empty slots **210**
class **211**
creating with the API **214**
defined in XML **212**
graphic representation **214**
loading **212**

G

groups
class **172**
creating with the API **177**
customizing **179**
defined in XML **175**
loading **175**
representation in a table **179**
representation in a tree **179**
shapes **173**

I

IlpAbstractClass class **59**
IlpAttribute interface **62**
IlpAttributeGroup interface **63**
IlpAttributeValueChangeSupport interface **75**
IlpAttributeValueHolder interface **63**
IlpAttributeValueProvider interface **64**
IlpBeansAttribute class **62**
IlpBeansClass class **59**
IlpBeansObject class **60, 97**
IlpChild interface **104**
IlpClass interface **44, 59**
IlpClassManager interface **66**
IlpComputedAttribute class **62, 64**
IlpContainer interface **104**
IlpDataSource interface **81, 82**
IlpDataSourceLoader class **108**
IlpDataSourceOutput class **108, 109**
IlpDefaultAttribute class **62**
IlpDefaultAttributeGroup class **63**
IlpDefaultClass class **30, 59**
IlpDefaultClassManager class **66**
IlpDefaultDataSource class **108**
IlpDefaultObject class **30**
IlpExtendedAttributeGroup class **63**
IlpLink interface **101, 104**
IlpLinkExtremity interface **104**
IlpMakeLinkInteractor class **163**
IlpMutableAttributeGroup interface **63**
IlpMutableClass interface **59**
IlpMutableDataSource interface **82**
IlpObject interface **44, 60, 81**
IlpObjectReferenceAttribute class **62**
IlpObjectSupport class **60**

IlpPolyline class **173**
IlpRect class **173**
IlpReferenceAttribute class **62**
IlpSAXSerializable interface **51**
IlpShelfItemPosition class **202**
IlpStaticAttribute class **62**
IltAlarm class **358**
IltAlarm.ImpactSeverity class **358**
IltAlarm.Severity class **358**
IltAlarm.State class **358, 364, 365**
IltAlarmObjectState class **361, 365**
IltBellcore class **358**
IltBellcore.SecState class **364**
IltBellcore.State class **364**
IltBellcoreObjectState class **364**
IltBiSONETObjectState class **160, 365**
IltBTS class **30, 250**
IltBTSAntenna class **30, 250**
IltCard class **30, 203**
IltCardCarrier class **219**
IltCardItem class **227**
IltDefaultDataSource class **82, 98, 110**
IltEmptySlot class **211**
IltGroup class **30, 172**
IltLed class **30, 203, 231**
IltLed.Type class **235**
IltLimitedNumericState class **338, 364**
IltLinearGroup class **173**
IltLink class **30, 148, 151, 154**
IltLinkBundle class **30**
IltLinkLayout class **163**
IltLinkPort class **163**
IltLinkSet class **30**
IltMisc class **358**
IltMisc.SecState class **358, 364**
IltNetworkElement class **30, 41, 116, 250**
IltObject class **34, 38, 154, 368**
IltObjectInfo class **41**
IltObjectState class **350, 361, 368**
IltOffPageConnector class **30, 258**
IltOSI class **358**
IltOSI.Availability class **364**
IltOSI.Procedural class **364**
IltOSI.Repair class **364**
IltOSI.Standby class **364**
IltOSI.State class **364**
IltPerformance class **358**
IltPolygon class **173**
IltPolyGroup class **173**
IltPort class **30, 203, 238, 239**
IltRectGroup class **173**
IltSAN class **358**
IltShelf class **30, 195**

- IltShelfItem class **202**
- IltShortLinkLayout class **163**
- IltSNMP class **358**
- IltSNMP. EGP class **358**
- IltSNMP. ICMP class **358**
- IltSNMP. Interface class **358**
- IltSNMP. IP class **358**
- IltSNMP. SNMP class **358**
- IltSNMP. State class **358, 364**
- IltSNMP. System class **358**
- IltSNMP. TCP class **358**
- IltSNMP. UDP class **358**
- IltSNMPObjectState class **364**
- IltSONET class **358**
- IltSONET. Protection class **358, 365**
- IltSONET. State class **358, 365**
- IltSONETObjectState class **365**
- IltSONETObjectStateSAXInfo class **392**
- IltState class **350, 358, 368**
- IltStateSystem class **358**
- IltTrap class **358**
- IltTrap. State class **358, 364, 365**
- IltTrap. Type class **358**
- IltTrapObjectState class **361, 365**

J

- JavaBeans
 - design patterns **68**

L

- LED (Light Emitting Diode) **230**
 - class **231**
 - creating with the API **234**
 - defined in XML **232**
 - loading **232**
 - predefined **235**
- link bundles
 - class **154**
 - creating with the API **156**
 - defined in XML **154**
 - loading **154**
- link sets **151**
 - class **151**
 - creating with the API **152**
 - defined in XML **151**
 - loading **151**
- links **148**
 - class **148**
 - connection ports **163**
 - creating with the API **149**
 - customizing **161**
 - defined in XML **148**
 - graphical representations **158**
 - link media **159**
 - loading **148**
 - network element **367**

- oriented **160**
- programming **167**
- representation in a table **162**
- representation in a tree **162**
- self-links **161**

M

- Misc State Dictionary **336**
 - icons **404**
- Miscellaneous states
 - in XML **386**

N

- network elements **14, 116**
 - creating **120**
 - customizing **143**
 - defined in XML **118**
 - families **139**
 - functions **135**
 - groups **17**
 - links **16**
 - loading **118**
 - partial **14, 140**
 - representation **122**
 - representation in a table **143**
 - representation in a tree **143**
 - shortcuts **15, 141**
 - sizes **142**
 - subnetworks **19**
 - types **122**

O

- object states **358**
 - Bellcore **364**
 - defining in XML **371**
 - dictionaries **363**
 - information window **401**
 - OSI **364**
- off-page connectors **24, 258**
 - class **258**
 - creating with the API **260**
 - customizing **261, 262**
 - defined in XML **259**
 - loading **259**
 - representation in a table **262**
 - representation in a tree **262**
- OSI
 - graphical representations **276**
 - in XML **374**
 - object states **364**
- OSI State Dictionary **276, 331**

P

- Performance State Dictionary **337**
- Performance states
 - in XML **388**
- ports **20, 238**

- class **239**
- creating with the API **242**
- defined in XML **240**
- graphical representation **187**
- loading **240**
- predefined **243**
- predefined business classes **30**
 - extending **39, 54, 75**
- predefined business objects
 - adding **96**
 - attributes **34**
 - BTS **250**
 - card carriers **218**
 - card items **226**
 - cards **20, 202**
 - computed attributes **38**
 - empty slots **210**
 - graphical representations **327**
 - groups **172**
 - LEDs **20, 230**
 - links **16**
 - network elements **14, 116**
 - off-page connectors **24, 258**
 - passive devices **328**
 - ports **20, 238**
 - shelves **20, 194**
 - subnetworks **182**

S

- SAN
 - in XML **390**
- SAN State Dictionary **338**
 - visuals **316**
- shelves **20, 194**
 - cards **202**
 - class **195**
 - creating with the API **197**
 - customizing **248**
 - defined in XML **196**
 - empty slots **210**
 - loading **196**
 - representation in a table **247**
 - representation in a tree **247**
- SNMP
 - graphical representations **295, 404**
 - in XML **381**
- SNMP State Dictionary **334**
 - system window **403**
- SONET
 - graphical representations **320, 323**
 - in XML **392**
- SONET State Dictionary **339**
 - secondary state **339, 404**
 - visuals **320**
- state dictionaries **329**
 - Bellcore **333**

- Misc **336**
- OSI **331**
- Performance **337**
- primary states **329**
- SAN **338**
- secondary states **329**
- SNMP **334**
- SONET **339**
- states **26**
 - alarm **26, 342**
 - base elements **26**
 - classes **358**
 - customizing **404**
 - modifiers **26**
 - systems **358**
 - trap **353**
 - values **358**
- subnetworks **19, 182**
 - creating with the API **185**
 - customizing **182**
 - defined in XML **183**
 - loading **183**
 - representing alarms **187**

T

- traps
 - color coding **271**
 - graphical representations **271**
 - in XML **399**
 - states **353**
 - types **353, 354**
 - values **355**