IBM ILOG JViews TGO V8.6

Context and Deployment Descriptor

Copyright

Copyright notice

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

The application context

Explains what the application context and the deployment descriptor are, and how to use them.

In this section

The concepts
Describes two key terms: context and deployment descriptor.

Initializing a context
Describes how to initialize a context either by loading a deployment descriptor file or through the API.

About internationalization
Describes how to define a context locale and how to access the localized resources.

The concepts

IBM® ILOG® JViews TGO provides contextual information and services that can be shared by a number of components in the broad sense of the term, that is, graphic components, data sources, and adapters. This common information can be accessed through a class that defines the application context, or in short, the context.

Context

The context encapsulates all common data and services in one object and provides access to information that is usually defined as singletons or static data.

All JViews TGO components can access contextual information from this single entry point called the context. In JViews TGO this idea of application context is similar to the Java™ BeansContext as defined in "Extensible Runtime Containment and Service Protocol for JavaBeans™". (For more information, see *http://java.sun.com*.) However, you do not have to be familiar with the BeansContext API to understand the concept of context in JViews TGO.

The context is defined by the `IlpContext` interface. This interface allows you to retrieve contextual information such as the locale and properties, as well as a number of services that are more than simple information providers. These services are the following:

- Type converter
- URL access service
- Image repository service
- Synchronization strategy
- Class manager
- Data Source manager
- Blinking manager
- Class Loader service
- Monitoring service

These services are described in detail in section *The context services*.

Generally, there is only one context per application. However, you can define multiple contexts if your application is to run in a multiuser environment or if different services should be accessible from various parts of the application. For more information, see *Creating a new context*.

Deployment descriptor

The deployment descriptor is an XML file used to initialize an application context. It is tagged with the XML element `<deployment>`. The context data and services to be initialized when the application is launched are defined between the opening and closing tags.

The deployment descriptor represents an easier and faster alternative for configuring a context than the API, as you do not have to write a single line of code nor recompile when you make a change.

For details on how to write and load a deployment descriptor, see *Writing a deployment descriptor* and *How to load a deployment descriptor using the parse method*.

Initializing a context

Describes how to initialize a context either by loading a deployment descriptor file or through the API.

In this section

Using a deployment descriptor
Describes how to initialize a context by loading a deployment descriptor file.

Using the API
Describes how to initialize a context by using the API.

Using a deployment descriptor

It is strongly recommend that you use the deployment descriptor alternative as it enables you to configure a context easily without having to write a single line of code. For more information, see *Writing a deployment descriptor* and *How to load a deployment descriptor using the parse method*.

You can initialize the default context by loading a deployment descriptor when calling the method `IltSystem`. There are several versions of the `Init` method, each with a different number of parameters.

The version with no parameter will attempt to load a default deployment parser named `cpldeploy.xml` by looking in the following places in this order:

1. The current directory
2. The user's home directory
3. The `user.home` system property
4. The `classpath`

For more information, see the class `IltSystem`.

How to configure the default context through a deployment descriptor

```
if (isApplet())
        IltSystem.Init(this, "deploy.xml");
else
        IltSystem.Init("deploy.xml");
```

You can only call `IltSystem` once, when you initialize the application for the first time. Any further call to this method is discarded.

If the method `IltSystem.Init` has not been called before `IltSystem.GetDefaultContext` is called, the latter will implicitly call `IltSystem.Init()` (the version with no parameter) to initialize the default context. This has the following two effects, one a drawback and one a feature:

- If you call an `IltSystem.Init` method with parameters, you must make sure that instances of classes using the default context are not created before `IltSystem.Init` is invoked. For example these instances should not be created during static initialization.
- If you are working in an IDE, you can configure the default context by changing the default deployment descriptor at design time. Your context will then be automatically loaded when you instantiate the Beans at run time.

Writing a deployment descriptor

The deployment descriptor is defined by an XML file that describes how to initialize the contextual information and services.

Following is a basic example that shows you what a deployment descriptor file looks like. For a more detailed description of its syntax, see the class `IlpDeploymentParser`.

```
<?xml version="1.0"?>
<deployment xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation = "ilog/cpl/schema/deploy.xsd">
   <urlAccess verbose="true">
        <relativePath>data/images</relativePath>
   </urlAccess>
   <classManager>
        <file>model.xml</file>
   </classManager>
</deployment>
```

A deployment descriptor is delimited by the XML element `<deployment>`. This element can contain a number of subelements each corresponding to the contextual information and services as described in *The concepts*.

The elements that you can use to define your deployment descriptor in XML format can be found in the XML schema file ***<installdir>* /data/ilog/cpl/schema/deploy.xsd**.

Note: XML elements are not necessarily read in the order in which they appear in the file. For example, the `<urlAccess>` element will be systematically read before the `<classManager>` element even if it is placed after it in the deployment descriptor file. Therefore, you can arrange these subelements in any order.

In the example, the deployment descriptor is composed of two subelements:

- The `<urlAccess>` element is read before the elements that potentially use it, that is, elements that require files to be loaded (the `<classManager>`, for example).
- The `<classManager>` element includes a list of file elements that contain filenames. These files are read in the order in which they appear in the file.

How to load a deployment descriptor using the parse method

Once you have created a context, you can initialize it from an XML deployment descriptor using the `parse` method of `IlpDeploymentParser`.

The following example shows how to use this method to load a deployment descriptor:

```
IlpDefaultURLAccessService urlBootService = new IlpDefaultURLAccessService();
URL url = urlBootService.getFileLocation("deploy.xml");
IlpDeploymentParser parser = new IlpDeploymentParser(url);
IltDefaultContext context = new IltDefaultContext();
try {
  parser.parse(null, context);
} catch (Exception e) {
```

```
  e.printStackTrace();
}
```

In this code, `IlpDefaultURLAccessService` retrieves the URL to the `deploy.xml` file, then a new context is created and passed to the `parse` method. Note that the first parameter of the `parse` method is not used here.

Extending the deployment descriptor

If you want the deployment descriptor to perform specific initialization that is not part of the JViews TGO contextual information and services, you can extend its XML format with custom elements and write a handler to process these elements. Custom elements will be passed to the handler as DOM elements.

Here is a very basic example of a deployment descriptor with only one custom element:

```
<?xml version="1.0"?>
<deployment xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation = "ilog/cpl/schema/deploy.xsd">
  <custom handlerClass="UserHandler"/>Joe Williams</custom>
</deployment>
```

Following is a basic handler class that sets the user name and stores it as a context property:

```
public class UserHandler implements IlpDeploymentParser.IlpCustomHandler {
  public UserHandler() {}
  public void handleCustomElement(Element customElement,
    IlpDefaultContext context) {
      String name = customElement.getTextTrim();
      context.setProperty("userName",name);
  }
}
```

The deployment parser processes `<custom>` elements in the order in which they appear, after dealing with standard elements. It tries to load the Java™ class specified in the `handlerClass` attribute and instantiate it using the default constructor. Then it calls the method `handleCustomElement(org.jdom.Element, ilog.cpl.service.IlpDefaultContext)`.

Note that the custom handler must implement `IlpDeploymentParser.IlpCustomHandler` and provide a default constructor (with no parameter).

Using the API

In more complex cases—if you need to add your own properties or services to the context—you will have to use the context API. For details, see *Creating a new context* and *Modifying a context*.

Note: You can use the context API in conjunction with a deployment descriptor. In other words, you can extend the syntax of the deployment descriptor and process added XML elements with a dedicated handler. For more information, see *Extending the deployment descriptor*.

You can initialize the default context through the API `IltDefaultContext`, that is, the default implementation of the interface `IlpContext`.

When you initialize the JViews TGO library (by calling `IltSystem.Init`), an instance of the default context is created. You can retrieve it by using the method `GetDefaultContext()`. This method is invoked by the default constructor of classes using contextual information and services (`IltDefaultDataSource`, for example).

How to retrieve the default context through the API

```
IlpContext context = IltSystem.GetDefaultContext();
```

Creating a new context

Generally, you are led to create new contexts when your application is to run in a multiuser environment and you want each user to have his/her own context.

To create an instance of `IltDefaultContext`, simply call:

```
IltDefaultContext context = new IltDefaultContext();
```

To use this new context, you must pass the `context` parameter to the constructor of each class making use of it in order to bypass the default context returned by `GetDefaultContext()`. For example, to create a data source, you should not use the default constructor `new IltDefaultDataSource()`, because using it would initialize the default context, but use instead the constructor that takes a context as its parameter, that is, `new IltDefaultDataSource(context)`.

How to create and initialize a context through the API

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

Modifying a context

You may want to modify an existing context to add your own services or properties.

How to change a service in a context

You can add a service to a context or replace an existing service with a new one using the `addService` method.

If you want to change the synchronization strategy to use regular Java™ synchronization and not the default strategy that uses the event thread, write the following line:

```
context.addService(IlSynchronizationStrategy.class,
                   new IlSynchronizeOnLockStrategy());
```

The first parameter of the `addService` method is a class that defines the service to be changed (usually this parameter is an interface because a service is generally defined by an interface). The second parameter is an implementation of this service. In other words, the second parameter is supposed to implement the class specified by the first parameter.

How to remove a service from a context

To remove a service from the context, you use the method `removeService` with a single parameter to indicate the service to be removed:

```
context.removeService(IlSynchronizationStrategy.class)
```

The default context implementation allows you to store properties. The following methods are available to help you handle the context properties:

- `public void setProperty(String name, Object property)` Sets the value of a property in the context.
- `public Object getProperty(String name)` Returns the value of a property in the context or `null` if the property is not defined.
- `public Map getProperties()` Returns the collection of all properties defined in the context.

  You can directly modify this Map by adding or removing your own context properties.

Creating a context in background

It is possible to initialize TGO in a background thread, which automatically invokes the `IltSystem` method along with the creation of a new context. It also initializes the TGO static resources that are usually only required at rendering time. This improves the graphic component initialization time (usually done at the Event Dispatch thread) and also provides some control over the initialized TGO static resources.

How to initialize TGO in background

When the JViews TGO application context is created (using the method `IltSystem`), many predefined resources are initialized, such as alarm states and default values for severities.

Other resources such as the different predefined object types are initialized on an as needed basis, when they are first referenced by the application. This process takes time and can slow down your application display time, for example, when you try to display a large number of different predefined nodes for the first time.

You can prevent such slowdowns by warming up TGO with an extended context initialization that is carried out on a background thread by the `IltSystemInitializer` utility class. This is useful for applications that perform time consuming tasks during startup and/or initialization (such as establishing remote connections, waiting for user authentication, and so on).

The `IltSystemInitializer` class creates the TGO application context (by internally invoking the `IltSystem.Init` method) extending the initialization to some resources usually lazily initialized by the rendering of predefined objects. This is done on a separate thread, concurrently with the time consuming part of your application initialization.

The following example illustrates the process:

Before starting a routine that is time consuming, start the TGO initialization:

```
// Start initialization on a worker thread
IltSystemInitializer.Init("my/deployment/file.xml");
```

When you are ready to start your UI, check for correct TGO initialization and continue:

```
// Wait for TGO initialization
// This is a blocking method!
IltSystemInitializer.WaitInitialization();

// You can check for initialization errors
Throwable error = IlpSystemInitializer.GetError());

// The context is already initialized
// There is no need to invoke IltSystem.Init()
_context = IltSystem.GetDefaultContext();
_tree = new IlpTree(_context);
```

Note that this technique is useful when the `IltSystemInitializer` method executes in parallel with a time consuming task.

About internationalization

The context can also store locale information for internationalization. The locale can be defined through the deployment descriptor or through the API.

How to define a context locale through the deployment descriptor

The following example illustrates how you can customize the context locale in the deployment descriptor file:

```
<locale>
     <language>fr</language>
     <country>FR</country>
</locale>
```

where:

- `language` is a lowercase two-letter ISO-639 code.
- `country` is an uppercase two-letter ISO-3166 code.

See `java.util.Locale` for more information on locale arguments.

How to define a context locale through the API

The following code shows how to create a context with the French locale:

```
IlpContext context = new IltDefaultContext(Locale.FRENCH);
```

How to create a localized version of JViews TGO resources

By default, JViews TGO is localized for the `en_US` locale. To create a localized version for another language, do the following:

1. Extract the following property files from `jviews-tgo-all.jar`:
   - `ilog/cpl/messages/EquipmentMessages.properties`
   - `ilog/tgo/messages/JTGOMessages.properties`
   - `ilog/cpl/messages/NetworkMessages.properties`
   - `ilog/cpl/messages/TableMessages.properties`
2. Create a new localized version of these files.

   Observe the naming conventions for localized property files as defined in `java.util.PropertyResourceBundle`.
3. Make sure that the localized files are accessible to JViews TGO by adding them to the application classpath.

How to access localized resources using the API

The following code extract shows how you can access the JViews TGO resource bundle and retrieve its messages according to the locale that is currently set in your application context:

```
Locale locale = context.getLocale();
ResourceBundle bundle =
   IlpI18NUtil.GetResourceBundle("ilog.tgo.messages.JTGOMessages", locale,
   "JViews TGO");
String value = bundle.getString("ilog.tgo.Alarm_Critical");
```

How to access localized resources using CSS

You can also retrieve localized resources from a CSS configuration file.

To do so, use the CSS function `'resource'` (`IlpResourceFunction`) as illustrated below:

```
object."ilog.tgo.model.IltAlarm/perceivedSeverity"[perceivedSeverity=Critical]

{
  label : '@|resource("ilog.tgo.messages.JTGOMessages",
"ilog.tgo.Alarm_Critical", "Critical")';
}
```

The context services

Describes the IBM® ILOG® JViews TGO context services in detail and how to customize them.

In this section

Type converter
Describes how to use the type converter service and how to customize it.

URL access service
Describes how to use the URL access service and how to customize it.

Image repository service
Describes how to use the image repository service and how to customize it.

Synchronization strategy
Describes how to use the synchronization strategy and how to customize it.

Class manager
Describes how to use the class manager and how to customize it.

Data source manager
Describes how to use the data source manager and how to customize it.

Blinking manager
Describes how to use the blinking manager and how to customize it.

Class loader
Describes how to use the class loader service and how to customize it.

Monitoring service
Describes how to use the monitoring service and how to customize it.

Type converter

The type converter service defined by the interface `IlpTypeConverter` is mainly used to read and write XML files that contain data source information. All the attributes of business objects are converted using this service. Refer to Type conversion in the *Business Objects and Data Sources* documentation for more information.

The main methods of the `IlpTypeConverter` interface are the following:

- `createJavaInstance(Class type, String value)`. Creates a Java™ instance from a class type. The value is initialized with the supplied string value. `IlEnum`, dates, colors and fonts are specifically supported. Icons are also specifically supported to add a full path to the icon name. All Java basic types and any Java class which has a string constructor are supported.
- `String createStringValue(Class type, Object value)`. Creates a `String` value from a general Java `Object`. This method produces string values that can be used in an XML file. Specifically, dates, colors, and fonts produce strings that are correctly formatted for use in data storage XML files. Any types that are not specifically handled will return the result of calling `toString()` on the value.

By default, the type converter service is implemented by the class `IlpDefaultTypeConverter`. This class can convert basic classes into Strings or Strings into basic classes. It handles the following basic types by default:

- `java.lang.Byte`
- `java.lang.Character`
- `java.lang.Double`
- `java.lang.Float`
- `java.lang.Integer`
- `java.lang.Long`
- `java.lang.Short`
- `java.lang.String`
- `java.awt.Color`
- `java.awt.Image`
- `java.awt.Font`
- `javax.swing.SwingConstants`
- `javax.swing.KeyStroke`
- `ilog.util.IlEnum`
- `ilog.util.Date`
- `ilog.cpl.graphic.IlpGraphicRenderer`

- `ilog.cpl.interactor.IlpInteractor`

The default type converter implementation uses the following two mechanisms to convert values from String to instance and instance to String:

1. A String-based constructor: This mechanism looks in the target class using the reflection API to retrieve a String-based constructor. This constructor is then used to convert String values to instances of a given class. Conversely, when converting instance values to String values, the method `toString` is used. The easiest way to automatically convert a user class using the default type converter implementation is to provide both a String-based constructor and a `toString` method in the user class.
2. Property editors: The default type converter implementation also allows you to register property editors (see `java.beans.PropertyEditor`) for a given class. You can register your own property editor using the following method:

   ```
   IlpDefaultTypeConverter.registerEditor (Class targetType, Class editorClass)
   ```

   The property editor for the following classes must not be overridden: `Date`, `Boolean`, `Color`, `Font`, `IlpPoint`, `IlpRect` and `IlpShelfItemPosition`. The default type converter class implements a specific treatment for these classes which cannot be overridden.

When a default context instance is created, it is initialized with a type converter that is an instance of `IlpDefaultTypeConverter`. You can change this configuration through the deployment descriptor file or through the API.

How to initialize the type converter service through the deployment descriptor file

You can initialize the type converter service in the deployment descriptor file using the tag `<typeConverter>` as follows:

```
[<typeConverter javaClass="class name"/>]
```

where the `javaClass` indicates the name of a Java class that implements the `IlpTypeConverter` interface.

The following example illustrates the customization of the type converter service through the deployment descriptor:

```
<?xml version="1.0"?>
<deployment xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation = "ilog/cpl/schema/deploy.xsd">
  <typeConverter javaClass = "package.MyTypeConverter" />
</deployment>
```

How to initialize the type converter service using the API

```
IlpDefaultContext context = ...
IlpTypeConverter converter = new IlpDefaultTypeConverter(context);
context.addService(IlpTypeConverter.class, converter);
```

How to use the type converter service through the API

```
IlpTypeConverter converter = context.getTypeConverter();
Integer value = (Integer) converter.createJavaInstance(Integer.class, "10");
```

and

```
Integer value = ...;
String valueString = converter.createStringValue(null, value);
```

The default type converter can be extended to support complex XML types (made up of several XML tags) through the interface `IlpSAXSerializable`. For more information, refer to Complex types.

URL access service

The URL access service is defined by the interface `IlpURLAccessService`.

It turns relative URLs into absolute URLs or pathnames and can be used to deploy applications and applets without hard coding pathnames or URLs.

More specifically, it can be used to access configuration files, data source files, and resource files.

`IlpURLAccessService`provides the method `getFileLocation(java.lang.String)` which returns the URL of a file in the path, or `null` if the file is not found.

This method takes as its parameter a relative URL or pathname as follows:

```
public java.net.URL getFileLocation (String filename);
```

Optionally, this service also provides the method `getFileLocation` with two optional parameters which specify whether the file shall be searched for in the CLASSPATH (directories and jars) or not, as follows:

```
public java.net.URL getFileLocation (String filename,
                                     boolean searchClassPathFirst,
                                     boolean searchClassPathLast)
```

How to initialize the URL access service through the deployment descriptor file

The URL access service can be initialized in the deployment descriptor file using the tag `<urlAccess>`.

```
[<urlAccess [verbose="bool"] [documentBase="xxx"]>
   [<relativePath>xxx</relativePath>]*
   [<absolutePath>xxx</absolutePath>]*
</urlAccess>]
```

For:

- `documentBase`: set the base URL.
- `<relativePath>`: add a path relative to the document base.
- `<absolutePath>`: add an absolute path.

How to customize the URL access service through the deployment descriptor

The following example illustrates the customization of the URL access service through the deployment descriptor:

```
<urlAccess>
    <!-- Add relative path to sample root directory -->
    <relativePath>../..</relativePath>
</urlAccess>
```

How to use the URL access service in an application

The following code extract shows you how you can use the URL access service in your application:

```
IlpContext context = IltSystem.GetDefaultContext();
IlpNetwork networkComponent = new IlpNetwork(context);

IlpURLAccessService urlService = context.getURLAccessService();
URL url = urlService.getFileLocation("europe.ivl");
if (url != null) {
  networkComponent.addBackgroundURL(url);
} else {
  System.err.println("Background file could not be found.");
}
```

By default, the URL access service is implemented by the class `IlpDefaultURLAccessService`. This default implementation uses one base URL that is usually defined according to the deployment model:

- Applets use `setDocumentBase` to register their base URL.
- Applications can also use `setDocumentBase` to register their base URL but, by default, they use the current directory. Please be aware that, in the case of a web application, the current directory might not be the root directory of the web application. Instead, it may be the directory of the JVM booted from by the web server. So, it is necessary to add relative paths to properly access the files in the root web application directory.

You can specify additional paths to the locations to be searched for a given filename. These additional paths can be:

- Paths relative to the document base. You set them using the method `addRelative`.
- Absolute paths. You set them using the method `addAbsolute`.
- JAR files defined by a URL relative to the document base. You set them using the method `addRelative` with a JAR filename as parameter.
- JAR files defined by an absolute URL. You set them using the method `addAbsolute` with a JAR filename as parameter.

How to use the URL access service through the API

The following example illustrates the use of the URL access service through the API:

```
IlpDefaultURLAccessService accessService = new IltDefaultURLAccessService();
```

```
accessService.setDocumentBase(new URL("file:C:/myfiles/"));
accessService.addRelative("myname/");
acesssService.addAbsolute(new URL("file://nfs/shared/resources/"));
accessService.addRelative("my.jar");
accessService.addAbsolute(new URL("file://nfs/shared/resources.jar"));
```

A call to `getFileLocation("myresource")` will return one of the following URLs:

```
file:C:/myfiles/myname/myresource
file://nfs/shared/resources/myresource
jar:file:C:/myfiles/my.jar!/myresource
jar:file://nfs/shared/resources.jar!/myresource
```

For details about the `file` URL syntax, refer to *http://www.w3.org/Addressing/rfc1738.txt*.

For details about the `jar` URL syntax, refer to *http://java.sun.com/javase/6/docs/api/java/net/JarURLConnection.html*.

Image repository service

The image repository service is defined by the interface `IlpImageRepository`. It is used to load all JViews TGO images. This service will return the requested image if it is already in the repository. Otherwise, it will use the URL access service to locate the image. If the image cannot be found, a default image is returned.

The image repository serves two main purposes:

1. It matches an image and the file location of this image.
2. It provides memory management, that is, reuse of images that have already been loaded.

   By keeping one reusable instance of an image in a repository, the image does not need to be loaded in memory multiple times.

The `IlpImageRepository`interface provides five methods to manage the image repository:

- `getImage(java.lang.String)`. Gets an image by location. The location is described in the form of a string presenting the relative path of the image. If the image is not already in the repository, it will be retrieved from the given location (as resolved by the URL access service).
- `getImageLocation(java.awt.Image)`. Gets the location string associated with an image in the repository. This method searches the repository for the specified image and returns the location of the image if it is found or `null` otherwise.
- `getImageLocations()`. Gets a list of all the image locations currently in the repository.
- `getImageURL(java.awt.Image)`. Gets the location of an image in the repository as a URL. This method searches the repository for the specified image and returns the location of the image as a URL if it is found or `null` otherwise.
- `removeImage(java.lang.String)`. Removes an image from the repository.

How to use the image repository service through the API

The following code extract shows you how you can use this service in your application:

```
IlpContext context = IltSystem.GetDefaultContext();
IlpImageRepository repository = context.getImageRepository();
Image img = repository.getImage("ilog/tgo/check.png");
```

By default, the image repository service is implemented by the class `IltDefaultImageRepository`.

How to customize the image repository service through the API

The following code extract shows you how you can customize the context to use your own image repository implementation:

```
context.addService(IlpImageRepository.class, myImageRepository);
```

Synchronization strategy

The synchronization strategy, named after the strategy design pattern, is defined by the class `IlSynchronizationStrategy`. It makes it possible to access shared resources simultaneously with a choice of synchronization schemes.

The `IlSynchronizationStrategy` class provides five methods to manage synchronized access to resources:

- `SetDefault(ilog.mt.IlSynchronizationStrategy)`. Defines the default strategy to use. The default locking behavior of an application can be changed just by changing this default.
- `GetDefault()`. Gets the default synchronization strategy to use.
- `synchronizeRun(java.lang.Runnable, java.lang.Object)`. Synchronizes the current thread and calls `run` on the specified `Runnable` object. Depending on the implementation of the strategy, the synchronization may be global (such as in a Swing application), or may use the object passed as the `lock` parameter to synchronize on.
- `readLock(java.lang.Runnable, java.lang.Object)`. Acquires a read lock for the current thread and calls `run` on the specified `Runnable` object. Some implementations may not support read/write locks. For this reason, the default implementation will call the `synchronizeRun` method.
- `writeLock(java.lang.Runnable, java.lang.Object)`. Acquires a write lock for the current thread and calls `run` on the specified `Runnable` object. Some implementations may not support read/write locks. For this reason, the default implementation will call the `synchronizeRun` method.

How to use the synchronization strategy through the API

The following code extract shows you how you can use the synchronization strategy in your application:

```
IlpContext context = IltSystem.GetDefaultContext();
IlSynchronizationStrategy strategy = context.getSynchronizationStrategy();
Runnable task = new Runnable() {
  public void run() {
  ...
  }
}
IlpNetwork networkComponent = new IlpNetwork(context);
strategy.writeLock(task, networkComponent.getModel());
```

JViews TGO provides two standard implementations of this service:

- `IlSwingThreadSyncStrategy` for Swing-based applications
- `IlSynchronizeOnLockStrategy` to synchronize on a mutex

How to customize the synchronization strategy through the deployment descriptor

The default synchronization strategy service can be customized through the deployment descriptor file using the tag `<synchronizationStrategy>`.

```
[<synchronizationStrategy type="application"|"servlet"/>]
```

where `type`:

- `application` creates a synchronization strategy for Swing-based applications using the `IlSwingThreadSyncStrategy` class.
- `servlet` creates a synchronization strategy for servlet applications using the `IlSynchronizeOnLockStrategy` class.

The following code extract shows you how you can customize the synchronization strategy through the API:

How to customize the synchronization strategy through the API

The only thing you can do through the API is to change the synchronization strategy used in the context.

```
IlpDefaultContext context = ...
IlSynchronizationStrategy strategy = new IlSwingThreadSyncStrategy();
context.addService(IlSynchronizationStrategy.class, strategy);
```

Class manager

The class manager is defined by the interface `IlpClassManager`. It handles a hierarchy of business classes. Some of the classes stored in the class manager can be loaded dynamically from a file where they are defined. For details, see Business class manager API .

The `IlpClassManager` interface provides five methods to manage a class hierarchy:

- `getRootClasses()` Returns the root classes of the model in a collection of `IlpClass` instances. Root classes are classes that do not inherit from another class.
- `getClass(String name)` Returns the class called `name` or `null` if this class does not exist.
- `getClasses()` Returns the classes of the model as a collection.
- `hasClass(IlpClass aClass)` Returns `true` if the given class belongs to this model.
- `hasClass(String name)` Returns whether a class with the given `name` is registered in the manager.

How to retrieve information about business class IltNetworkElement

The following code extract shows how to retrieve information about the business class `IltNetworkElement`:

```
IlpContext context = IltSystem.GetDefaultContext();
IlpClassManager classMgr = context.getClassManager();
IlpClass bclass = classMgr.getClass("ilog.tgo.model.IltNetworkElement");
```

By default, the class manager service is implemented by the class `IltDefaultClassManager`. This default implementation stores dynamic classes, makes it possible to load classes from XML files and creates dynamic classes to represent JavaBean™ classes using introspection.

The default class manager service can be customized through the deployment descriptor file using the tag `<classManager>`.

All the files listed in the `classManager` scope are loaded in the default class manager implementation, and are therefore recognized as business classes within the application context.

How to customize the default class manager through the deployment descriptor

The following example is an extract of `<installdir>/samples/table/customClasses/deploy.xml` and illustrates how you can customize your class manager in the deployment descriptor file:

```
<classManager>
    <!-- Register custom classes -->
```

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

where `<file>` is the business class file that will be loaded in the class manager at initialization time.

How to initialize the class manager service through the API

The following code extract shows how to intialize the Class Manager service through the API:

```
IlpDefaultContext context = ...;
// Create a new class manager instance
IltDefaultClassManager classMgr = new IltDefaultClassManager();

// Set the new service in the context
context.addService(IlpClassManager.class, classMgr);
```

How to customize the class manager service through the API

The following code extract shows how to customize the Class Manager service through the API:

```
// Load a model file in the class manager. To do this, the class manager
// needs to use two services: type converter, URL access service - which can
// be retrieved from the context
try {
  classMgr.parse("model.xml", context.getTypeConverter(),
      context.getURLAccessService());
}
```

Data source manager

The data source manager defined by the interface `IlpDataSourceManager` provides access to the data sources that it handles.

It provides the following methods:

- `getDataSources()` Returns the data sources created in the context of this data source manager.
- `hasDataSource(IlpDataSource aDataSource)` Returns `true` if the given data source belongs to this manager.

By default, the data source manager is implemented by the class `IlpDefaultDataSourceManager`. This default implementation maintains a collection of data sources which are referenced through weak references that do not prevent garbage collection.

How to use the data source manager through the API

```
IlpContext context = ...
IlpDataSourceManager dsMgr = context.getDataSourceManager();
```

or

```
IlpDataSourceManager dsMgr = (IlpDataSourceMananger)
   context.getService(IlpDataSourceManager.class);
```

How to Customize the Data Source Manager Through the API

```
IlpDefaultContext context = ...
IlpDataSourceManager dsMgr = MyDataSourceManager();
context.addService(IlpDataSourceManager.class, dsMgr);
```

Blinking manager

The blinking manager, defined by the interface `IlpBlinkingManager`, manages the blinking of colors.

The `IlpBlinkingManager` interface provides six methods to manage the blinking of colors:

- `addColor(IlpBlinkingColor color)` Adds a blinking color to an object. A blinking color must be added to a blinking manager for the object to blink.
- `removeColor(IlpBlinkingColor color)` Removes the blinking color from the object.
- `setOnPeriod(long onPeriod)` Sets the default duration, in milliseconds, of the period the color is on.
- `setOffPeriod(long offPeriod)` Sets the default duration, in milliseconds, of the period the color is off.
- `getOnPeriod()` Returns the default duration, in milliseconds, of the period the color is on.
- `getOffPeriod()` Returns the default duration, in milliseconds, of the period the color is off.

How to customize the blinking manager service through the deployment descriptor

The following code extract shows how to customize the on and off periods of blinking colors in the blinking manager.

```
[<blinkingManager>
     <onPeriod>111</onPeriod>
     <offPeriod>111</offPeriod>
</blinkingManager>]
```

where:

- `<onPeriod>` sets the duration, in miliseconds, of the period when the color is on.
- `<offPeriod>` sets the duration, in miliseconds, of the period when the color is off.

How to customize the blinking manager through the API

```
IlpContext context = IltSystem.GetDefaultContext();
IlpBlinkingManager blinkingMgr = context.getBlinkingManager();
blinkingMgr.setOnPeriod(1000);
blinkingMgr.setOffPeriod(500);
```

The on/off periods in this example are used for `IlpBlinkingColor` instances that do not have their own on/off periods defined.

By default, the blinking manager service is implemented by the class `IltDefaultBlinkingManager`.

The default blinking manager service can be customized through the deployment descriptor file using the tag `<blinkingManager>`.

Class loader

The Class Loader service is defined by the interface `IlpClassLoaderService`.

It enables JViews TGO to load classes used in the definition of the business model and in the styling of the components and objects. This is performed with a class loader that is different from the default class loader provided by the Java™ framework. By default, it is done with the class loader service defined by the class `IlpDefaultClassLoaderService` which tries to load classes in the following order:

1. Using the thread context class loader
2. Using the class loader used to load this service
3. Using the system class loader

`IlpClassLoaderService` provides the method `loadClass(java.lang.String)` which returns the Java class that corresponds to the given class name. If the class name cannot be loaded, this method should throw a `ClassNotFoundException`.

```
public Class loadClass (String name) throws ClassNotFoundException
```

How to initialize the class loader service through the deployment descriptor file

You can initialize the class loader service in the deployment descriptor file using the tag `<classLoader>` as follows:

```
[<classLoader javaClass="class name"/>]
```

where `javaClass` indicates the name of a Java class that implements the `IlpClassLoaderService` interface.

The following example illustrates the customization of the class loader service through the deployment descriptor:

```
<?xml version="1.0"?>
<deployment xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation = "ilog/cpl/schema/deploy.xsd">
  <classLoader javaClass = "package.MyClassLoader" />
</deployment>
```

How to Initialize the Class Loader Service Through the API

```
IlpDefaultContext context = ...
  IlpClassLoaderService service = new MyClassLoaderService();
context.addService(IlpClassLoaderService.class, service);
```

Monitoring service

The Monitoring Service is defined by the interface `IlpMonitoringService`.

It provides a facility for developers to register and monitor time-consuming activities, so that all interested parties can be aware of and appropriately act upon these activities and their progress.

The Monitoring Service makes use of the IBM® ILOG® JViews Maps activity monitoring API by exposing an instance of `IlvThreadedActivityMonitor` that is associated with each `IlpContext` for a given `IlpGraphicComponent`. This `IlvThreadedActivityMonitor` instance can be used to register activities and track their progress.

For more details on the use of the `IlvThreadedActivityMonitor` and related API, see Thread monitoring.

To see how to use the Monitoring Service, look at the Editor sample for the the Network Component at **<*installdir*> /samples/network/editor/index.html** .

Note: All access to the underlying `IlvThreadedActivityMonitor` should be made through the Monitoring Service (`IlpMonitoringService.getActivityMonitor`). This means that the `IlvThreadedActivityMonitorProperty` is not supported.

How to configure the monitoring service through the deployment descriptor

The Monitoring Service is configured through the deployment descriptor. You can specify implementations of the `IlvThreadedActivityMonitor.ActivityListener` that will be added automatically to the underlying `IlvThreadedActivityMonitor` of the Monitoring Service, in the order in which they are specified in the deployment descriptor.

```
<monitoring>
    <activityListeners>
      <activityListener javaClass="ActivityListenerOne"/>
      <activityListener javaClass="ActivityListenerTwo"/>
    </activityListeners>
</monitoring>
```

`ActivityListenerOne` will be added to the `IlvThreadedActivityMonitor` before `ActivityListenerTwo`.

Check the XML schema file of the deployment descriptor at **file://data/ilog/cpl/schema/deploy.xsd** to know how to place the XML tag `<monitoring>` for the Monitoring Service in the deployment descriptor.

Index