Aware IM

Version 8.5

Programmers Reference

CONTENTS

ABOUT THIS DOCUMENT **4**

1 ADDING PROGRAMMING EXTENSIONS TO AWARE IM **4**

2 GENERAL GUIDELINES **5**

3 ADDING CUSTOM PROCESSES **5**

3.1 WRITING CODE FOR A CUSTOM PROCESS COMPONENT 6
3.1.1 execute (IExecutionEngine, Object []) *7*
3.1.2 resume (IExecutionEngine, Object []) *7*
3.1.3 cancel () *8*
3.1.4 Examples of custom process components *8*
3.1.5 IExecutionEngine interface *11*
3.1.6 IExecutionContext interface *23*
3.1.7 IEntity interface *25*
3.1.8 IEntityTemplate interface *27*
3.1.9 INotification interface *28*

4 ADDING CUSTOM CHANNELS **28**

4.1 IMPLEMENTING CHANNEL'S SOURCE AND SINK 30
4.1.1 Writing a Sink *31*
4.1.2 Writing a Source *34*
4.1.3 Handling replies to service requests *38*
4.1.4 MessageInterpreter class *39*
4.1.5 MessageBuilder class *41*
4.2 IMPLEMENTING CHANNEL TYPE COMPONENT 42
4.3 IMPLEMENTING CHANNEL SETTINGS EDITOR 46

5 ADDING FUNCTION LIBRARIES **49**

5.1 IMPLEMENTING FUNCTION LIBRARY 49
5.1.1 ISQLBuilderHelper interface *52*
5.1.2 Function library example *52*

6 ADDING CUSTOM DOCUMENT TYPES **55**

6.1 IMPLEMENTING CLIENT SIDE COMPONENT 56
6.2 IMPLEMENTING SERVER SIDE COMPONENT 59
6.3 DATAPRESENTATIONTEMPLATE CLASS 61
6.4 IDOCUMENTDATASOURCE INTERFACE 63
6.5 EXAMPLE 64

7 ADDING REPORT SCRIPTLETS **69**

7.1 IMPLEMENTING CODE FOR THE SCRIPTLET 70
7.1.1 IReportEnvironment interface *71*
7.2 EXAMPLE 73

8 WRITING CLIENT-SIDE PLUGINS **75**

8.1 ARCHITECTURE OF THE CLIENT-SIDE CODE 76
8.2 MODIFYING DEFAULT BEHAVIOR AND PRESENTATION OF QUERIES 77
8.3 MODIFYING DEFAULT BEHAVIOR AND PRESENTATION OF FORMS 79
8.4 MODIFYING DEFAULT BEHAVIOR AND PRESENTATION OF FORM SECTIONS 81
8.5 MODIFYING DEFAULT PRESENTATION OF INDIVIDUAL FIELDS ON FORMS 81
8.6 MODIFYING DEFAULT BEHAVIOR OF MENU IN VISUAL PERSPECTIVES 84
8.7 MODIFYING DEFAULT BEHAVIOR AND PRESENTATION OF CONTENT PANELS IN VISUAL PERSPECTIVES 84
8.8 AWAREAPP OBJECT 85
8.9 USING JAVASCRIPT TO INTEGRATE CUSTOM CORDOVA PLUGINS FOR NATIVE MOBILE APPLICATIONS 86
8.9.1 Creating a contact on the phone 87
8.9.2 Send email to the selected contact 89

APPENDIX A: CORRESPONDENCE OF ATTRIBUTE TYPES AND JAVA TYPES 90

Trademarks

About this document

This document describes the Application Programming Interface (API) to the ***Aware IM*** software system. The intended audience is Java and Javascript programmers who want to add programming extensions to the applications configured with ***Aware IM***.

1 Adding programming extensions to Aware IM

Aware IM is a powerful software system that allows users to configure the fully functional Internet enabled applications without involving any programming work in most cases. Thus ***Aware IM*** can be used not only by professional developers but also by business professionals and end users.

In certain cases, however, it may be necessary to add programming extensions to ***Aware IM*** in order to implement the application functionality that is otherwise very difficult or impossible to configure using the available configuration tools. This mostly applies to the calculation extensive functionality (for example, some mathematical algorithms) or interaction with a non-standard hardware that is not supported by ***Aware IM*** out-of-the-box.

It is important to stress, though, that in order to support such functionality programmers do not have to learn new technologies or understand at a very deep level the internals of the ***Aware IM*** software. All that is required is the knowledge of the Java programming language (other languages will be supported in the future) and generic understanding how the extensions fit into the ***Aware IM*** framework. In most cases programmers have to implement some standard interface that ***Aware IM*** requires in order for the new extensions to plug in naturally into the ***Aware IM*** software. The programmer's job is, therefore, to understand what different methods of the interface mean and write the code that implements these methods.

Aware IM allows programmers to write extensions (plug-ins) in the following areas (see "Aware IM User Guide" for the explanation of these areas):

- Custom processes implemented by software components rather than by rules.
- Custom channels for intelligent business objects.
- Custom function libraries.
- Custom document types.
- Report calculation components (scriptlets).

In addition ***Aware IM*** allows developers to use Javascript to affect the behaviour of user interface widgets on the client (browser).

This document aims to describe in detail what is involved in adding extensions to each of these areas. Before we start describing each area individually we will describe some common steps required to add the extensions to any of the above areas.

2 General guidelines

The following section describes the common steps required to add programming extensions to the ***Aware IM*** software system. It is assumed that the reader is familiar with the basics of programming in the Java programming language and the general steps involved in compiling Java source code and packaging jar files.

- All extensions (except scripts running on the client) have to be written in the Java programming language.
- As all extensions implement some standard Java interface exposed by ***Aware IM***, the `CLASSPATH` variable used when compiling the extensions must contain a reference to the correct location of the Aware IM jar filees (`awareim.jar`, `shared.jar`, `openAdaptor.jar`, `connection.jar`). For example, the command to compile the code implementing the extension might look something like this:

```
javac –classpath c:\AwareIM\Tomcat\lib\awareim.jar;
c:\AwareIM\Tomcat\lib\shared.jar;
c:\AwareIM\Tomcat\lib\connection.jar;
c:\AwareIM\Tomcat\lib\openAdaptor.jar myextension.java
```

- The name of the Java package where the extension will reside is not mandated and is up to a programmer.
- The resulting extensions may be packaged into one or several jar files. These jar files have to be placed into the AwareIM/CustomJars directory (if this directory does not exist, create it manually)
- If running Aware IM as a Windows service modify the files AwareIM/bin/awareim.conf and AwareIM/bin/wrapper.conf to add the jar files to the list of jar files used

3 Adding custom processes

The following section describes how to write extensions implementing custom processes and plug them into ***Aware IM***.

To implement custom processes and add them to the configuration of your application follow the steps below:

1. Write the code for the process, compile it and add it to the jar file with your custom extensions. Make sure that the jar file is placed in the AwareIM/CustomJars directory (see section 2)
2. Add the process to your application configuration using the Configuration Tool (see “Adding/Editing Processes” section in the “Aware IM User Guide”). When adding the process click on the “Implementation” property in the list of process properties and select the “Process is implemented by custom component” radio button. Then specify the fully qualified name of the class that implements the component, for example `com.myextensions.MyProcessExtension`

The rest of the section describes how to write the code implementing the custom process (step 1 above).

3.1 Writing code for a custom process component

When writing a process it is important to understand how processes fit into the architectural framework of ***Aware IM***. When a process is started ***Aware IM*** creates the *execution context*, which contains the information about the environment in which the process is running. This information contains the data about the user who started the process, the process start-up time, the information about the current database transaction as well as many other things. The execution context is available to the process as the `IExecutionContext` interface. Because a process in ***Aware IM*** runs in a "sandbox" there are certain limitations on what the process can do (see guidelines later in this section).

A process communicates with ***Aware IM*** through the call-back interface called `IExecutionEngine`. This interface has a variety of methods that a process can use to change attributes of objects, start other processes, call services of service providers etc.

Sometimes a particular process calls a service, which takes a long time to fulfil. In this case the call to the service may time out. If this happens the process is *suspended*. This means that ***Aware IM*** moves the process into the "standby" mode – it is no longer active and is waiting for the reply from the service provider. Whenever such reply arrives (the reply may arrive in a few seconds, hours, days or months or never arrive at all) ***Aware IM*** automatically *resumes* the process and the process continues execution from where it left off.

The custom process component must implement `IProcess` interface, which has two main methods – `execute` and `resume`. The `execute` method is called when the process is started and the `resume` method is called when the process is resumed after having been suspended. Both methods are described in detail later in the section.

The following guidelines should be used when writing a process:

- A process is not supposed to perform any operations with the database directly (such as create, open or modify database tables, manage transactions etc) – this is taken care of by the ***Aware IM*** framework.
- A process generally should not take long to execute as it is running within a context of a database transaction. If a process does take a long time to execute it should every now and again send a special signal to the ***Aware IM*** framework. Having received such a signal ***Aware IM*** would suspend the process, commit the current transaction and continue the process execution (resume the process). This signal can be sent if the process throws `SuspendProcessException` and sets its `resumeImmediately` flag to true, for example:

```
throw new SuspendProcessException (true).
```

Before doing so the process should obviously remember its current state so that it can correctly resume itself in the `resume` method.

- If a process calls a method of the IExecutionEngine interface, which throws ServerTimeOutException the process is supposed to catch this exception and throw SuspendProcessException passing the original ServerTimeOutException to the SuspendProcessException, for example:

```
....
try
{
     engine.updateEntity (this, entity, null, null);
}
catch (ServerTimeOutException se)
{
    throw new SuspendProcessException (se);
}
```

The methods of the IProcess interface that the custom process components must implement are described below:

3.1.1 execute (IExecutionEngine, Object [])

```
public Object execute (IExecutionEngine engine, Object [] parameters)
  throws SuspendProcessException, ExecutionException, AccessDeniedException;
```

This is the main method of the interface that is called when the process is started.

Parameters:

engine – call back interface that allows the process to call services of ***Aware IM***. See IExecutionEngine interface.

parameters – array of the process parameters. This array is null if process doesn't have any input declared in the configuration. Otherwise the array has as many members as there are input business objects in the declaration of the process. Each member of the array is an instance of IEntity interface.

Returns: should always return null.

Throws:

SuspendProcessException – if one of the methods of the IExecutionEngine threw SuspendProcessException or if the process wants to commit the current transaction.

ExecutionException – if something is wrong

AccessDeniedException – should not be thrown by the process component directly

3.1.2 resume (IExecutionEngine, Object [])

```
public Object resume (IExecutionEngine engine, Object reply)
  throws SuspendProcessException, ExecutionException, AccessDeniedException;
```


This method is called by the ***Aware IM*** framework when the process receives the reply from the service provider and gets resumed.

Parameters:

engine – the call back interface that allows the process to call services of ***Aware IM***. See IExecutionEngine interface.

reply – reply received from the service provider. This is the instance of the INotification object. The attributes of this notification can be used to check the specific values of the reply. For example, if a remote service of a service provider were called, the reply notification would have the attributes as defined by the service declaration exposed by the provider. If a process was resumed immediately after its suspension in order to commit the current transaction, the reply is `null`.

Returns: should always return `null`.

Throws:

SuspendProcessException if one of the methods of the IExecutionEngine threw `SuspendProcessException` or if a process wants to commit the current transaction.

ExecutionException if something is wrong

AccessDeniedException shouldn't be thrown by the process component directly

3.1.3 cancel ()

```
public boolean cancel ();
```

This method is called by the ***Aware IM*** framework when the operator sends the request to cancel the process.

Parameters: none

Returns: the method should return `true` if it allows the system to cancel the process. In this case it should set an internal flag indicating that the process has been cancelled. The `execute` and `resume` methods should periodically check this flag and if its value is `true` abort the execution of the process (see section 3.1.4). If a process is in a state that does not allow canceling, the method should return `false`.

3.1.4 Examples of custom process components

The following example shows the code for the simple process component that changes the attribute of the process input.

```
public class SimpleProcess implements IProcess
{
```



```
public SimpleProcess()
{
}

/**
 * @see com.bas.basserver.executionengine.IProcess#cancel()
 */
public boolean cancel()
{
	return true;
}

/**
* @see
com.bas.basserver.executionengine.IProcess#execute(com.bas.basserver.exe
cutionengine.IExecutionEngine, java.lang.Object[])
 */
public Object execute (IExecutionEngine engine, Object[] parameters)
	throws SuspendProcessException, ExecutionException,
AccessDeniedException
{
	// one entity is declared as process input
	if (parameters == null || parameters.length != 1 ||
		! (parameters [0] instanceof IEntity))
	{
		return null;
	}

	// get the parameter
	IEntity entity = (IEntity) parameters [0];
	Object oldValue = null;
	try
	{
		// change "attrName" attribute to "attrValue" in memory
		oldValue = entity.getAttributeValue ("attrName");
		entity.setAttributeValue ("attrName", "attrValue");
	}
	catch (Exception e)
	{}

	try
	{
		// update entity in persistence
		engine.updateEntity (this, entity,
			new String [] {"attrName"},
			new Object [] {oldValue} );
	}
	catch (ServerTimeOutException ste)
	{
		throw new SuspendProcessException (ste);
	}

	return null;
}

/**
```

```
	* @see
	com.bas.basserver.executionengine.IProcess#resume(com.bas.basserver.exec
	utionengine.IExecutionEngine, java.lang.Object)
	 */
	public Object resume (IExecutionEngine engine, Object reply)
		throws SuspendProcessException, ExecutionException,
			AccessDeniedException
	{
		return null;
	}
```

The next example shows the code of the process that performs some lengthy calculation and throws `SuspendProcessException` every now and again:

```
public class LongProcess implements IProcess
{
	private boolean m_cancelled = false;
	private long m_count;

	public LongProcess()
	{
	}

	/** @see IProcess#cancel () */
	public boolean cancel ()
	{
		// set m_cancelled flag. This flag will be checked by execute //
		and resume methods
		synchronized (this)
		{
			m_cancelled = true;
		}
		return true;
	}

	/**
	 * @see IProcess#execute(IExecutionEngine, Object [])
	 */
	public Object execute (IExecutionEngine engine, Object [] parameters)
		throws SuspendProcessException, ExecutionException
	{
		m_count = 0;
		execute (engine);

		return null;
	}

	private void execute (IExecutionEngine engine)
		throws SuspendProcessException
	{
		try
		{
			// get the context in which the process is running
			IExecutionContext myContext =
					engine.getExecutionContext(this);

			while (true)
```

```
                {
                        Thread.sleep (500);
                        // finish after 400 iterations
                        if (m_count++ >= 400)
                                break;

                        // every 4 iterations update our progress
                        if (m_count % 4 == 0)
                        {
                                int progress = myContext.getProgress ();
                                if (progress < 100)
                                        myContext.setProgress (progress + 1);
                        }

                        // check if we have been cancelled
                        synchronized (this)
                        {
                                if (m_cancelled)
                                        break;
                        }

                        // every 120 iterations suspend ourselves so that
                        // transaction can be committed (though this is
                        // only for illustrational purposes as we are not //
                        doing any transactional operations here such as //
                        changing values of attributes etc)
                        if (m_count % 120 == 0)
                        {
                                throw new SuspendProcessException (true);
                                // we will be called soon - see resume
                        }
                }
        }
        catch (InterruptedException ie)
        {}
}

/**
 * @see IProcess#resume(IExecutionEngine, Object)
 */
public Object resume (IExecutionEngine engine, Object reply)
        throws SuspendProcessException, ExecutionException
{
        // this is called every 120 iterations
        execute (engine);
        return null;
}
```

3.1.5 IExecutionEngine interface

The reference to this interface is passed to the `execute` and `resume` methods. A process can use the methods of this interface to communicate with the ***Aware IM*** system. The interface has the methods listed below. Note that most methods throw the following exceptions:

`ServerTimeOutException` – thrown if there was a call of the service exposed by a remote service provider, which timed out.

`ExecutionException` – thrown if there was a run-time error. All changes are discarded.

`AccessDeniedException` – thrown if there was violation of the access level. All changes are discarded.

The first parameter of each method is the process that called the method. When calling methods of `IExecutionEngine` interface the process component should always pass a reference to "this" as the first parameter to the method call. This parameter is omitted from the description below.

1. getExecutionContext (IProcess)

```
public IExecutionContext getExecutionContext (IProcess process)
```

This method provides the execution context in which the process is running.

Parameters:

process – the process whose execution context is required

Returns: the execution context in which the process is running as an instance of the `IExecutionContext` interface.

2. createEntity (IProcess, String)

```
public IEntity createEntity (IProcess parent, String entityName)
  throws ServerTimeOutException, AccessDeniedException, ExecutionException;
```

This method creates new instance of the specified business object. If there are any rules associated with the business object they are executed.

Parameters:

entityName – the name of the business object to create

Returns: the created instance of the business object. The attributes are initialized with the values set by the initialization rules (if any), otherwise the initial values are `null`.

3. createEntityFromTemplate (IProcess, IEntityTemplate, ChangedReference [])

```
public IEntity createEntityFromTemplate (IProcess parent, IEntityTemplate
                                         template, ChangedReference []
                                         changedRefs)
  throws ServerTimeOutException, AccessDeniedException, ExecutionException;
```

This method creates the new instance of the specified business object from the specified template, which can be pre-initialized with some values. If there are any rules associated with the business object they are executed.

Parameters:

template – an instance of the IEntityTemplate interface. The blank instance of the template can be obtained calling the static method `DomainFactory.createEntityTemplate(IEntityDefinition)` where `IEntityDefinition` is the definition of the entity being created. This definition can be obtained from `IDomainVersion`, which in turn can be obtained from the execution context of the process (see API of the `DomainFactory`, `IEntityDefinition` and `IDomainVersion` interfaces provided in the JavaDoc).

changedRefs – should be null

Returns: the created instance of the business object. The attributes are initialized with the values supplied in the provided template and set by the initialization rules (if any). The initialization rules will overwrite any values set in the original template.

4. updateEntity (IProcess, IEntity, String [], Object [], ChangedReference [])

```
public void updateEntity (IProcess parent, IEntity entity, String []
                          changedAttributes, Object [] oldValues,
                          ChangedReference [] changedRefs)
  throws ServerTimeOutException, AccessDeniedException, ExecutionException;
```

This method updates the instance of the business object in the database. If there are any rules associated with the business object they are executed.

Parameters:

entity – an instance of the business object to write to the database

changedAttributes – an array of the attribute names of the business object that have been changed (`null` if unknown).

oldValues – an array of the old values of the changed attributes (`null` if unknown). Must be of the same size as the `changedAttributes` parameter

changedRefs – should be null

5. deleteEntities (IProcess, EntityIdAndName [])

```
public void deleteEntities (IProcess parent, EntityIdAndName [] entities)
  throws ServerTimeOutException, AccessDeniedException, ExecutionException;
```

This method deletes the specified instances of the business objects. If there are references to the deleted instances anywhere in the database, these references are deleted as well. If the deleted instance is a parent of some other object, the child object is deleted as well.

Parameters:

entities – the names and ID's of the instances to be deleted.

6. getEntity (IProcess, String, Long)

```
public IEntity getEntity (IProcess parent, String entityName, Long id)
  throws AccessDeniedException, ExecutionException;
```

This method retrieves the specified instance of the specified business object.
Note: any large binary attributes (of the `Binary`, `Document` and `Picture` types) are not retrieved by this method – the value of the corresponding attributes will be `null`. To retrieve the values of the binary attributes use the `getBinaryData` method. The `getEntity` method also retrieves only single references of the object. To retrieve all references of the object, use `getAllReferences` method.

Parameters:

entityName – the name of the business object to retrieve

id – the unique id of the instance to retrieve (the unique id is assigned to the instance automatically by the system when the instance is created)

Returns: the retrieved instance of the business object or `null` if the instance was not found.

7. getEntities (IProcess, EntityIdAndName [])

```
public IEntity [] getEntities (IProcess parent, EntityIdAndName []
                               entities)
  throws AccessDeniedException, ExecutionException;
```

This method retrieves the specified instances of the specified business objects.

Note: any large binary attributes (of the `Binary`, `Document` and `Picture` types) are not retrieved by this method – the value of the corresponding attributes will be `null`. To retrieve the values of binary attributes use the `getBinaryData` method. . The `getEntities` method also retrieves only single references of the object. To retrieve all references of the object, use `getAllReferences` method.

Parameters:

`entities` – the names and unique id's of the business object instances to retrieve

Returns: the retrieved instances of the business objects or `null` if the instances were not found.

8. getBinaryData (IProcess, String, Long, String)

```
public byte [] getBinaryData (IProcess parent, String entityName, Long id,
                              String attributeName)
  throws AccessDeniedException, ExecutionException;
```

This method retrieves the value of the specified binary attribute of the specified instance of the business object. Normally if an instance of a business object is retrieved using `getEntity` or `getEntities` methods the values of attributes of the `Binary`, `Document` or `Picture` types are not retrieved. The above method will retrieve the binary value of the specified attribute.

Parameters:

`entityName` – the name of the business object to retrieve

`id` – the unique id of the instance to retrieve (the unique id is assigned to the instance automatically by the system when the instance is created)

`attributeName` – the name of the attribute the value of which is retrieved (must be of the binary type). For attributes of the `Document` and `Picture` types the attribute name must be specified using the "dot" notation as attributes of these types are complex attributes consisting of the file name and file data. The file data part of the attribute is binary:
`attributeName.DOCDATA`
where `attributeName` is the name of the attribute of the `Document` or `Picture` type. The returned bytes are the bytes of the document or image.

Returns: the bytes representing the document, image or binary data for the attribute.

9. setBinaryData (IProcess, String, Long, String, byte[])

```
public void setBinaryData (IProcess parent, String entityName, Long id,
                           String attributeName, byte[] data)
  throws AccessDeniedException, ExecutionException;
```


This method sets the data of the specified binary attribute of the specified instance of the business object. This method allows clearing binary attribute (if "data" is null). Normally if binary data is provided as part of the entity null is interpreted as "no change" for binary data and therefore it is impossible to clear binary data as part of entity update. This method allows doing that.

Parameters:

entityName – the name of the business object to set the data of

id – the unique id of the instance to set the data of (the unique id is assigned to the instance automatically by the system when the instance is created)

attributeName – the name of the attribute the value of which is set (must be of the binary type). For attributes of the `Document` and `Picture` types the attribute name must be specified using the "dot" notation as attributes of these types are complex attributes consisting of the file name and file data. The file data part of the attribute is binary:
`attributeName.DOCDATA`
where `attributeName` is the name of the attribute of the `Document` or `Picture` type. The returned bytes are the bytes of the document or image.

data – the data to set. If "null" the binary data of the attribute is cleard.

10. getAllReferences (IProcess, IEntity, String)

```
public IEntity [] getAllReferences (IProcess parent, IEntity refOwner,
                                    String refOwnerAttr)
  throws AccessDeniedException, ExecutionException;
```

This method retrieves all references of the specified instance of the business object referenced by the specified attribute of the reference type. The method `getReferences` of the `IEntity` interface also returns the references of the business object, so it is possible to retrieve the object by calling `getEntity` and then call `getReferences` method. However, calling the `getEntity` or `getEntities` methods will only retrieve single references and will not retrieve multiple references. The `getAllReferences` method, on the other hand, guarantees that all references of the business object referenced through the specified attribute are retrieved.

Parameters:

refOwner – the instance of the business object the references of which are retrieved

refOwnerAttr – the name of the attribute of the business object () the references of which are retrieved (must be of the reference type)

Returns: the instances of the business objects referenced through the specified attribute or `null` if no instances are referenced through the specified attribute.

11. getObjectReferences (IProcess, IEntity, String)

```
public Vector getObjectReferences (IProcess parent, IEntity refOwner,
                                   String refOwnerAttr)
  throws AccessDeniedException, ExecutionException;
```

This method is very similar to the `getAllReferences` method. However, it returns references as a collection of `ObjectReference` objects rather than `IEntity` objects. `ObjectReference` is a structure containing an object name and `ID`. If only names and `ID`'s of the references are required calling this method is much more efficient than calling `getAllReferences`, which retrieves the values of all attributes.

Parameters:

refOwner – the instance of the business object the references of which are retrieved

refOwnerAttr – the name of the attribute of the business object the references of which are retrieved (must be of the reference type)

Returns: the names and ID's of business objects referenced through the specified attribute as a vector of the `ObjectReference` objects or `null` if no instances are referenced through the specified attribute.

12. addReferences (IProcess, String, Long, String, EntityIdAndName [])

```
public void addReferences (IProcess parent, String refOwnerName, Long
                           refOwnerId, String refOwnerAttr,
                           EntityIdAndName [] refsToAdd)
  throws ServerTimeOutException, AccessDeniedException, ExecutionException;
```

This method adds the specified references to the specified instance of the business object. There are two ways references can be added to a business object. The instance of the object can be obtained using the call to the `getEntity` or `getEntities` methods and then references can be set using the `setReferences` method of the `IEntity` interface. However, the `setReferences` method requires that *all* references are set into the business object and therefore all the existing references must be obtained before adding the new ones. This is, of course, very inefficient. The `addReferences` method does not require that the existing references of the business object are retrieved and allows adding the specified references to the existing ones.

Parameters:

refOwnerName – the instance of the business object that references are added to

refOwnerAttr – the name of the attribute of the business object the references of which are being modified (must be of the reference type)

refOwnerId – the unique ID of the instance of the business object that the references are added to

refsToAdd – the references to be added as an array of business object names and ID's

13. deleteReferences (IProcess, String, Long, String, EntityIdAndName [])

```
public void deleteReferences (IProcess parent, String refOwnerName, Long
                              refOwnerId, String refOwnerAttr,
                              EntityIdAndName [] refsToDelete)
  throws ServerTimeOutException, AccessDeniedException, ExecutionException;
```

The method is similar to the addReferences method. However, it deletes the specified references rather than adds them.

Parameters:

refOwnerName – the instance of the business object that references are deleted from

refOwnerAttr – the name of the attribute of the business object the references of which are being deleted (must be of the reference type).

refOwnerId – the unique ID of the instance of the business object that references are deleted from

refsToDelete – the references to be deleted as an array of business object names and ID's. If `null` all existing references of the business object's instance are deleted.

14. createNotification (IProcess, String)

```
public INotification createNotification (IProcess parent, String
                                         notificationName)
  throws ServerTimeOutException, AccessDeniedException, ExecutionException;
```

This method creates the specified notification. If there are any rules associated with the creation of the notification they are executed.

Parameters:

notificationName – the name of the notification to create (the notification with this name must be configured in the business space version)

Returns: the instance of the created notification

15. executeChildProcess (IProcess, String, IEntity [], Boolean)


```
public IProcess executeChildProcess (IProcess parent, String processName,
                                     IEntity [] parameters,
                                     Boolean asynchronous)
  throws ServerTimeOutException, AccessDeniedException, ExecutionException;
```

This method starts the specified process as a child of the current process. This method allows custom process components to start other processes.

Parameters:

processName – the name of the process to start (the business space version must have the process with this name configured)

parameters – the instances of the business objects representing the process input if it is declared or null if the process has no input

asynchronous – if `true` the new process is started in a separate thread.

Returns: the reference to the child process component.

16. executeEntityService (IProcess, String, EntityIdentifier, IEntity [], String, Boolean)

```
public INotification executeEntityService (IProcess parent,
                                           String serviceName,
                                           EntityIdentifier provider,
                                           IEntity [] parameters,
                                           String channelTypeName,
                                           Boolean asynchronous)
  throws ServerTimeOutException, AccessDeniedException, ExecutionException;
```

This method calls the specified service of the specified service provider.

Parameters:

serviceName – the name of the service to call (the service provider must expose the service)

provider – the name and unique ID of the intelligent business object which provides the service

parameters – the instances of the business objects representing the service input if it is required or null if the service requires no input

channelTypeName – the name of the channel through which the service is requested. If `null` the service is requested through the default channel of the provider.

asynchronous – if `true` the service is called in a separate thread.

Returns: the notification appropriate to the reply type of the called service if the service call was successful.

17. sendNotification (IProcess, IEntity, INotification, String)

```
public void sendNotification (IProcess parent, IEntity recipient,
                              INotification notification,
                              String channelTypeName)
  throws ServerTimeOutException, AccessDeniedException, ExecutionException;
```

Send the specified notification to the specified recipient.

Parameters:

recipient – the instance of the intelligent business object which receives the notification

notification – the instance of the notification to send to the recipient. This can be obtained by calling the createNotification method.

channelTypeName – the name of the channel supported by the recipient through which the notification is sent. If `null` the notification is sent through the default channel of the recipient.

18. executeNamedQuery (IProcess, String, Integer, Integer)

```
public QueryResult executeNamedQuery (IProcess parent, String queryName,
                                      Integer startRange, Integer span)
  throws AccessDeniedException, ExecutionException;
```

This method runs the query with the specified name and returns the instances of the business objects matching the conditions of the query. The query with this name must be configured in the business space version. The method may return only a subset of found instances as defined by the `startRange` and `span` parameters.

Parameters:

queryName – the name of the query to run

startRange – the number identifying the start of the range of the business object instances returned by the query (starting with 1). If all instances are to be returned this parameter must be `null`.

span – the number of instances of the business objects to return. If all instances are to be returned this parameter must be `null`.

Returns: the instances of the business objects that match the query conditions. These instances are wrapped in the `QueryResult` object, which also has the total number of instances in the system matching the query conditions. Note that this number can be greater than the number of business object instances returned by the method as the method may return only a subset of instances matching the conditions (if `startRange` and `span` are specified).

19. executeQuery (IProcess, Query, Integer, Integer)

```
public QueryResult executeQuery (IProcess parent, Query query,
                                 Integer startRange, Integer span)
  throws AccessDeniedException, ExecutionException;
```

This method is very similar to the `executeNamedQuery` method except that it runs the provided query instead of the configured query. The provided query does not have to exist in the business space version and may be created by the calling process.

Parameters:

query – the query to run

startRange – the number identifying the start of the range of the business object instances returned by the query (starting with 1). If all instances are to be returned this parameter must be `null`.

span – the number of instances of the business objects to return. If all instances are to be returned this parameter must be `null`.

Returns: the instances of the business objects that match the query conditions. These instances are wrapped in the `QueryResult` object, which also has the total number of instances in the system matching the query conditions. Note that this number can be greater than the number of business object instances returned by the method as the method may return only a subset of instances matching the conditions (if `startRange` and `span` are specified).

20. importData (IProcess, String, String, Integer, Boolean)

```
public Collection importData (IProcess parent, String entityName,
                              String importFileName, Integer batchSize,
                              Boolean validate)
  throws AccessDeniedException, ExecutionException;
```

This method imports the instances of the specified business object into the system from the specified file (see also “Export and Import” section in the “Aware IM User Guide”).

Parameters:

entityName – the name of the business object the instances of which are imported

importFileName – the full path to the file being imported

batchSize – the number that specifies how many records are imported into the system before a transaction is committed. The system imports records in batches. Once the batch is complete the transaction is committed. If something goes wrong with the import of the subsequent batches so that the import process is aborted, those records that have been already committed are not affected. The higher the number the faster the processing of records, however it is also more likely that the whole process fails without writing any records.

validate – if the value of this parameter is `true` any rules associated with the business object being imported are executed. Consequently if there are any records which trigger the error action (see "Aware IM Rule Language Reference") they are ignored and the corresponding business object is not created or modified. Setting this flag to `true` slows down the import process considerably, but on the other hand it guarantees that only valid instances of the business object are imported

Returns: the created or modified instances of the business object as a collection of `IEntity` objects if `validate` flag is `true`. If `validate` flag is `false` `null` is returned.

21. importRelationships (IProcess, String, String, Integer)

```
public void importRelationships (IProcess parent, String entityName,
                                 String importFileName, Integer batchSize)
  throws AccessDeniedException, ExecutionException;
```

This method imports the relationships of the specified business object into the system from the specified file (see also "Export and Import Relationships" section in the "Aware IM User Guide").

Parameters:

entityName – the name of the business object the relationships of which are imported

importFileName – the full path to the file being imported

batchSize – the number that specifies how many records are imported into the system before a transaction is committed. The system imports records in batches.
Once the batch is complete the transaction is committed. If something goes wrong with the import of the subsequent batches so that the import process is aborted, those records that have been already committed are not affected. The higher the number the faster the processing of records, however it is also more likely that the whole process fails without writing any records.

22. exportEntities (IProcess, IEntity [], String,

ExportOptions)

```
public void exportEntities (IProcess parent, IEntity [] entsToExport,
                            String exportFileName, ExportOptions options)
  throws AccessDeniedException, ExecutionException;
```

This method exports the specified instances of the specified business object into the specified file (see also "Export and Import" section in the "Aware IM User Guide").

Parameters:

entsToExport – the instances of the business objects to be exported

exportFileName – the full path to the file to export to

options – the export options (whether to export ID's, relationships, binary attributes etc).

3.1.6 IExecutionContext interface

There are quite a number of methods exposed by this interface. Most of the methods are of interest to the system only. The following section only describes those methods that are relevant to custom process components.

1. getProgress ()

```
public int getProgress ()
```

Parameters: none
Returns: the number identifying the current progress of the process in percents.

2. setProgress (int)

```
public void setProgress (int progress)
```

This method can be used to set the progress of the custom process component. The system will use this number when the operator inquires about the progress of the process.

Parameters:

progress – the number identifying the current progress of the process in percents.

3. getStartupTime ()

```
public Date getStartupTime ()
```

Parameters: none

Returns: the date and time when the process was started as `java.util.Date` object.

4. getSuspensionTime ()

```
public Date getSuspensionTime ()
```

Parameters: none
Returns: the date and time when the process was last suspended as `java.util.Date` object or `null` if the process was never suspended.

5. getDomainVersion ()

```
public IDomainVersion getDomainVersion ()
```

Parameters: none
Returns: the current business space version.

6. getAccessLevel ()

```
public AccessLevel getAccessLevel ()
```

Parameters: none
Returns: `the` access level of the user that started the process.

7. getLoginName ()

```
public String getLoginName ()
```

Parameters: none
Returns: the login name of the user that started the process.

8. getTimeout ()

```
public long getTimeout ()
```

Parameters: none
Returns: the number of milliseconds that the system will wait before timing out the process when it is suspended. If the value is –1 the process never times out (this is the default value).

9. setTimeout (long)

```
public void setTimeout (long timeout)
```


Set the number of milliseconds that the system will wait before timing out the process if it is suspended.

Parameters:

`timeout` – the timeout value in milliseconds. If the value is –1 the process never times out.

3.1.7 IEntity interface

`IEntity` interface extends `IObject` interface. `IObject` interface is the parent to both `IEntity` and `INotification`. It exposes the following methods:

1. getName ()

```
public String getName ()
```

Parameters: none
Returns: the name of the business object or notification as configured in the business space version.

2. getId ()

```
public long getId ()
```

Parameters: none
Returns: the unique id of the business object instance (always –1 for notifications). The unique id is automatically assigned to the business object instance by the system when the instance is created.

3. getAttributeValue (String)

```
public Object getAttributeValue (String attributeName)
      throws InvalidParameterException
```

This method returns the value of the specified attribute. Note: the attribute must not be a reference. Use the `getReferences` method to retrieve references.

Parameters:

`attributeName` – the name of the attribute of the business object or notification the value of which is returned

Returns: the value of the attribute as `Object`. The type of the object corresponds to the type of the attribute. See the Appendix A that provides the mapping between attribute types and Java types.

Throws: `InvalidParameterException` if the specified attribute name does not exist in the business object or notification

4. setAttributeValue (String, Object)

```
public void setAttributeValue (String attributeName, Object value)
      throws InvalidParameterException
```

This method sets the value of the specified attribute.
Note: the attribute must not be a reference. Use the `setReferences` method to set references.

Parameters:

attributeName – the name of the attribute of the business object or notification the value of which is set

value - the value of the attribute to set as `Object`. The type of the object corresponds to the type of the attribute. See Appendix A that provides the mapping between attribute types and Java types.

Throws: `InvalidParameterException` if the specified attribute name does not exist in the business object or notification

5. deepCopy ()

```
public IObject deepCopy ();
```

This method takes a "deep" copy of "this" object.

Parameters: none
Returns: copied business object or notification

6. getReferences (String)

```
public ObjectReference [] getReferences (String attributeName)
      throws InvalidParameterException
```

This method retrieves references of the business object or notification. If the business object was obtained in the first place by the calls to the `getEntity` or `getEntities` methods of the `IExecutionEngine` interface the `getReferences` method will always return `null` for multiple references (it will always return the only existing reference if the attribute is a single reference). Use the `getAllReferences` method in the `IExecutionEngine` interface to obtain all multiple references of the business object.

Parameters:

attributeName – the name of the attribute of the business object or notification the references of which are returned (must be of the reference type)

Returns: the references as `ObjectReference` objects or `null` if no objects are referenced by the specified attribute.

Throws: `InvalidParameterException` if the specified attribute name does not exist in the business object or notification or it is not a reference.

7. setReferences (String, ObjectReference [])

```
public void setReferences (String attributeName, ObjectReference [] refs)
      throws InvalidParameterException
```

This method sets references of the business object or notification. Note that if the `updateEntity` method of the `IExecutionEngine` interface is used to update the business object in the database after setting its references the system will delete any existing references of the business object and add whatever references are set by the `setReferences` method. If, however, the `setReferences` method sets the value to `null` and then the `updateEntity` method is called, the `null` value is interpreted by the system as "no change" so no existing references are deleted. To clear existing references use the `deleteReferences` method in the `IExecutionEngine` interface.

Parameters:

attributeName – the name of the attribute of the business object or notification the references of which are being set (must be of the reference type)

refs - the references being set.

Throws: `InvalidParameterException` if the specified attribute name does not exist in the business object or notification or it is not a reference.

There are a number of methods that are specific to the `IEntity` interface as opposed to the `IObject` interface, however, these are mostly used by the system are not relevant to custom process extensions.

3.1.8 IEntityTemplate interface

`IEntityTemplate` represents the instance of the business object before it has been added to the database. As such it does not have the unique ID and some other attributes assigned to the instance of the business object automatically by the system. The main usage of this interface is to set initial values of an instance when using the `createEntityFromTemplate` method in the `IExecutionEngine` interface.

The interface exposes `getName`, `getAttributeValue`, `setAttributeValue`, `getReferences` and `setReferences` methods. Their signatures and purpose is identical to the corresponding methods in the `IEntity` interface.

3.1.9 INotification interface

The `INotification` interface extends the `IObject` interface and therefore shares most methods with the `IEntity` interface. Some specific methods are described below (these methods are not relevant to process extensions, however, they may be helpful when writing custom channels – see section 4.1)

1. `getServiceStatus ()`

```
public int getServiceStatus ()
```

This method returns the status of service request. Valid values are:

```
SERVICE_STATUS_SUCCESS;
SERVICE_STATUS_WAITING;
SERVICE_STATUS_FAILURE;
```

Parameters: none
Returns: an integer indicating the service status.

2. `getMessage ()`

```
public String getMessage ()
```

Parameters: none
Returns: any message associated with the notification (for example, the error message if the service request failed) or `null` if no message is associated with the notification.

4 Adding custom channels

Communication between external entities, such as other software systems and hardware devices (which are modeled in ***Aware IM*** as intelligent business objects) and ***Aware IM*** is performed through *channels*. A channel represents a media through which messages are sent between external systems and hardware devices (collectively called here as “external parties”) and ***Aware IM***. A channel is responsible for converting messages into the format that can be understood by the ***Aware IM*** software on one end and the external party on the other end. There can be one-way or two-way channels. In a one-way channel messages are only sent in one direction – from ***Aware IM*** to the external party or from the external party to ***Aware IM***. In a two-way channel messages are sent in both directions.

From ***Aware IM*** point of view there are only two types of messages – *service requests* and *notifications*. Service request is a request for execution of a particular service. The external party

being an intelligent business object may expose a number of services that can be requested by ***Aware IM***. Similarly the current business space version in ***Aware IM*** may expose a number of services that can be requested by the external party. The service requestor may expect a particular reply to the service as declared in the definition of the service.

A notification, on the other hand, represents just a piece of information that is sent to the external party or to ***Aware IM*** (the business space version in ***Aware IM*** may contain rules which define how different notifications are handled). It is up to a receiver of the notification to decide how it will react to this information or whether it will react at all. Generally no reply to the notification is expected from the receiver.

A channel therefore must translate the service request or notification message that is sent by ***Aware IM*** to the external party into whatever format the external party understands. Similarly a channel has to translate a message sent by the external party into the service request or notification message that can be understood by ***Aware IM***.

Note that if an external party is sending service requests to ***Aware IM*** it must first log into ***Aware IM*** as a particular user (an intelligent business object representing the external party must be a member of the `SystemUsers` business object group), otherwise the service request will be sent on behalf of a "guest" user. Sending notifications to ***Aware IM,*** on the other hand, does not require logging in.

There are a number of predefined channels that ***Aware IM*** comes with, such as the `E-mail` channel, the `SOAP` channel and others (see the "Defining Intelligent Business Objects" section in the "Aware IM User Guide"). If your application connects to external software systems or hardware using protocols that are not supported by ***Aware IM*** out-of-the-box, you are likely to need to write your own custom channel that will connect your ***Aware IM*** system with the required software system or a hardware device.

The following section describes what is involved in adding a new custom channel to ***Aware IM***.

To add a new custom channel to ***Aware IM*** follow the steps below:

1. Write the code for the component that will handle communication between ***Aware IM*** and your software system or a hardware device (see section 4.1 for details)
2. Write the code for the component that describes the capabilities of the new channel – this component must implement the `IChannelType` interface (see section 4.1.2 for details).
3. If your channel has properties that need to be controlled via the user interface in the Configuration Tool, write the code for the component that implements this user interface (see section 4.3 for details).
4. Compile the source code of your components and package them in a jar file(s). Make sure that the jar file is referenced in the `CLASSPATH` (see section 2)
5. Add the definition of the new channel to the `BASServer.props` file located in the `BIN` directory of your ***Aware IM*** installation. You have to add a line that provides the fully qualified name of the component that you wrote in step 2. The name of the property must start with the `ChannelType` prefix and the value is the fully qualified name of the component. For example:


```
ChannelTypeMail=com.bas.basserver.channels.EmailChannelType
```

If everything has been done correctly the name of the new channel must appear in the list of channels available for intelligent business objects in the Intelligence tab of the Business Object Property Editor (see “Defining Intelligent Business Objects” section in the “Aware IM User Guide”)

4.1 Implementing channel’s Source and Sink

The first step that is required to add a custom channel to ***Aware IM*** is to write the component(s) that handle communication between ***Aware IM*** and the external software system or hardware device. In order to write the code for such components it is important to understand the architectural framework that ***Aware IM*** uses to communicate with an external software system or device (“external party”).

Aware IM incorporates the `openadaptor™` framework written by Dresdner Kleinwort Wasserstein to handle communication with external parties. This framework abstracts the process of sending messages between systems. The source of the message is abstracted in a component called *Source* and the destination of the message is abstracted in the component called *Sink*. Thus messages are generated by Source and travel to the corresponding Sink.

If communication with the external party requires that messages be sent from ***Aware IM*** to the external party the specific Sink needs to be implemented (***Aware IM*** framework will automatically provide the corresponding Source). If communication requires that messages be sent from the external party to ***Aware IM*** the specific Source needs to be implemented (***Aware IM*** framework will automatically provide the corresponding Sink).

Messages that float between ***Aware IM*** and external party represent arrays of `DataObject’s`. `DataObject` is the class that allows defining any number of attributes of different types and provides generic access to them - very much like the `IEntity` interface (in fact `IEntity` is a wrapper around `DataObject`). `DataObject` exposes the `getAttributeValue` and `setAttributeValue` methods that allow access to the values of its attributes.

Aware IM sends `DataObject’s` of some specific structure to the external party and it expects `DataObject’s` of this structure from the external party. As explained in the previous section this structure implies that there are only two forms of requests that can be sent between ***Aware IM*** and the external party – service requests and notifications.

Aware IM provides two utility classes `MessageBuilder` and `MessageInterpreter` that allow a programmer to easily create or read messages representing service requests and notifications without knowing too many details about the underlying structures of `DataObject’s` and its attributes.

To summarize a programmer has to do the following:

1. When writing a Sink the service request or notification from ***Aware IM*** has to be translated into the protocol of the external party and sent to this party.
2. When writing a Source the message in the external party's format has to be translated into the standard service request or notification and sent to ***Aware IM***.

The details of this process are described in the sections below.

4.1.1 Writing a Sink

To write a custom Sink follow the steps below:

1. Inherit the Sink from the `AbstractSimpleSink` class.
2. If necessary override `init` method to provide the Sink-specific initialization (if there's any) – this method is called once when the Sink is initialized. The Sink-specific properties would have been entered by the user in the Configuration Tool (see also section 4.3)
3. You must override the following method to provide your own handling of messages sent from ***Aware IM***:

```
public void processMessage (Message msg) throws PipelineException;
```

You can translate messages from ***Aware IM*** using the following techniques:

a) get the array of `DataObject's` from the message:
```
DataObject [] dobs = msg.peekDataObjects ();
```

b) use `MessageInterpreter` class to interpret the message (see also 4.1.3):
```
MessageInterpreter mi = new MessageInterpreter (dobs);
INotification notif = mi.getNotification (); or
String serviceName  = mi.getServiceName ();
```

As an example you can see the code snippet below that implements the Sink that sends outgoing e-mails from ***Aware IM***. Note that this code is for illustration purposes only and certain details such as e-mail construction and sending and error handling are omitted.

```
public class EmailSink extends AbstractSimpleSink
{
      private String m_mailHost = null;
      private Session m_session = null;
      private String m_fromAddress = null;

      /** Initialize the sink. Mail host and 'from address' properties are **
      specified in the user interface */
      public void init (String name, Properties props, String prefix,
                              Controller controller)
                  throws IbafException
      {
            try
            {
                  super.init(name, props, prefix, controller);

                  m_mailHost = props.getProperty (prefix+"."+"MAIL_HOST");
```

```
                if (m_mailHost == null)
                      throw new IbafException ("Mail host property for
                      outgoing e-mails is not specified.");

                Properties sessionProps = System.getProperties ();
                sessionProps.put ("mail.smtp.host", m_mailHost);

                m_fromAddress = props.getProperty (prefix+"."+
                                  "MAIL_FROM_ADDRESS");

                m_session=Session.getDefaultInstance(sessionProps,null);
          }
          catch (Exception e)
          {
                e.printStackTrace();
                throw new IbafException(e.toString());
          }
    }

    /** Message handling method - must be provided */
    public void processMessage (Message msg)
                throws PipelineException
    {
          String subject = null, toAddress = null;
          String sentDateStr = null;
          try
          {
                DataObject [] dobs = msg.peekDataObjects ();
                // get all message parameters from data objects
                MessageInterpreter mi = new MessageInterpreter (dobs);

                // get e-mail address from channel values
                DataObject chvDob = mis.getChannelValues ();
                if (chvDob == null)
                      return;  // ignore the message

                try
                {
                      toAddress = (String) chvDob.getAttributeValue
                                  (EMAIL_ADDRESS);
                }
                catch (InvalidParameterException ipe)
                {}
                if (toAddress == null)
                      return; // ignore the message

                INotification notif = mis.getNotification ();
                if (notif == null)
                      return;  // ignore the message

                // we expect notification of a certain structure - it
                // must have "BODY" and SUBJECT attributes defined
                String body = null;
                try
                {
                      body = (String) notif.getAttributeValue
                                        (EMAIL_BODY_ATTR);
                }
```

```
                catch (InvalidParameterException ipe)
                {}
                if (body == null)
                      return;  // ignore the message

                // subject is optional
                try
                {
                      subject = (String) notif.getAttributeValue
                                  (EMAIL_SUBJECT_ATTR);
                }
                catch (Exception e)
                {}

                if (subject == null)
                      subject = "";

                // create MAPI e-mail message
                MimeMessage email = createEmail (body, subject,
                                                 toAddress);

                if (email != null)
                {
                      // send e-mail using MAPI
                      sendEmail (email, toAddress, new Date ());
                }
          }
          catch (Exception e)
          {}
    }

    /** Send the e-mail using MAPI. A lot of details are omitted **/
    private boolean sendEmail (MimeMessage email, String toAddress,
                               String sentDate)
    {
          try
          {
                Transport.send (email);
          }
          catch (Exception e)
          {}
          return true;
    }

    /** Create e-mail message using MAPI. Details are omitted
     */
    private MimeMessage createEmail (String body, String subject, String
                                     toAddress, Date sentDate)
          throws Exception
    {
          MimeMessage mailMessage = new MimeMessage (m_session);

          if (m_fromAddress != null)
                mailMessage.setFrom (InternetAddress.parse

          mailMessage.setRecipients (javax.mail.Message.RecipientType.TO,
                                     InternetAddress.parse (toAddress,
                                     false));
```



```
        mailMessage.setSubject  (subject);
        mailMessage.setSentDate (sentDate);
        return mailMessage;
    }
```

4.1.2 Writing a Source

Before writing a Source you need to decide the type of Source that will receive messages from your software system or a hardware device. Three types of Sources are supported by the `openadaptor`™ framework:

1. *Polling* - the Source of this type will periodically poll a resource looking for data and messages that need to be processed or sent, e.g. a database, a file system or queue-based middleware.
2. *Listening* - the Source of this type will wait for a connection, establish a connection and listen, e.g. a socket.
3. *Callback* - the Source of this type will register for events and receive call-backs, e.g. publish/subscribe middleware.

You must decide which is the most appropriate type of Source for your situation. Depending on which implementation you choose, you will need to implement different methods.

After you decide which Source type you will use follow the steps below:

1. Inherit your class implementing the Source from the `AbstractSimpleSource` class.
2. Override the `init` method to set the Source type and/or initialise the Source with some specific properties. The Source-specific properties would have been entered by the user in the Configuration Tool (see also section 4.3) The type of the Source is set using the following method:

```
setSourceType(POLL_SOURCE); or
setSourceType(LISTEN_SOURCE); or
setSourceType(CALLBACK_SOURCE); or
```

3. Depending on the Source type override the following methods:

- For polling Source -

```
public DataObject[] sourcePoll() throws IbafException
```

  This method should return an array of `DataObject's` (or `null` if there is no data to process at the moment).
- For listening Source -

```
public void sourceListen() throws IbafException
```

  This method should call the following method when it has some `DataObject's` to process:

```
sourceProcess(DataObject[] dobs)
```

- For call-back Source –

You will need to implement the third party call-back method. This method should call the following method when it has some `DataObject's` to process:

```
sourceProcess(DataObject[] dobs)
```

In the methods above you need to construct the appropriate `DataObject's,` which will be delivered to ***Aware IM*** by the framework. To construct the `DataObject's` you can use the `MessageBuilder` class as shown in the example below.

4. You should implement the following method to perform any necessary "start up", such as establishing connections, registering call-backs, etc:
   ```
   public void sourceStartUp() throws IbafException
   ```
5. You should implement the following method to perform necessary "clean up", such as destroying connections, releasing resources, etc:
   ```
   public void sourceCleanUp() throws IbafException
   ```

As an example you can see the code snippet that implements the Source that handles e-mails coming to a mailbox and sends them as notifications to ***Aware IM***. Note that this code is for illustration purposes only and certain details are omitted.

```
public class EmailSource extends AbstractSimpleSource
{
	private String m_mailHost;
	private String m_userName;
	private String m_password;
	private Session m_session;
	private Store   m_store;
	private Folder  m_inbox;

	/** Initializes the source.
	 * Mandatory properties
	 * MAIL_HOST_PROP name of the mail host, e.g. mail.optus.com
	 * MAIL_USER_PROP name of the user
	 * MAIL_PASSWORD_PROP credentials of the user
	 * @exception IbafException If initialization fails.
	 */
	public void init( String name, Properties props, String prefix,
	                  Controller controller )
		throws IbafException
	{
		try
		{
			super.init(name, props, prefix, controller);

			// set the type of the source!
			setSourceType (CALLBACK_SOURCE);

			// get different settings that would have been entered// //
			by the user in the Configuration Tool
			m_mailHost = props.getProperty (prefix+"."+MAIL_HOST);
			if (m_mailHost == null)
				throw new IbafException ("Mail host property for
```

```
                        incoming e-mails is not specified.");

                m_userName = props.getProperty (prefix+"."+MAIL_USER);
                if (m_userName == null)
                        throw new IbafException ("User name property is not
                        specified");

                m_password = props.getProperty (prefix + "." +
                                                MAIL_PASSWORD);
                if (m_password == null)
                        throw new IbafException ("Password property is not
                                                specified");

                Properties settings = System.getProperties ();
                settings.put ("mail.smtp.host", m_mailHost);
                settings.put ("mail.store.protocol", "pop3");

                m_session = Session.getInstance (settings, null);

                m_store = m_session.getStore ();

        }
        catch (Exception e)
        {
                e.printStackTrace();
                throw new IbafException (e.getMessage ());
        }
}

public void sourceStartUp()
        throws IbafException
{
        super.sourceStartUp ();

        // try to connect to the remote host
        try
        {
                connect ();
        }
        catch (Exception me)
        {
        }
}

public void sourceCleanUp()
        throws IbafException
{
        super.sourceCleanUp ();

        // close everything
        if (m_inbox != null)
        {
                synchronized (m_inbox)
                {
                        try
                        {
                                m_inbox.close (false);
                        }
```



```
                    catch (Exception e)
                    {}
              }
        }

        if (m_store != null)
        {
              try
              {
                    m_store.close ();
              }
              catch (Exception e)
              {}
        }
  }

  /** connect to the remote host and register our callback function */
  private void connect () throws Exception
  {
        try
        {
              m_store.connect (m_mailHost, 110, m_userName,
                               m_password);
        }
        catch (Exception se)
        {}

        // Open a Folder
        try
        {
              m_inbox = m_store.getFolder ("INBOX");

        }
        catch (Exception e)
        {
              throw new IbafException (e.getMessage ());
        }

        // Register our call back method with the third party API!!
        synchronized (m_inbox)
        {
              m_inbox.addMessageCountListener(new MessageCountAdapter()
              {
                    public void messagesAdded (MessageCountEvent ev)
                    {
                          gotMessage (ev);
                    }
              });
        }
  }

  // This is a call-back method that creates data objects */
  private void gotMessage (Message m)
  {
        // create notification of a particular structure
        INotification notif = DomainFactory.createNotification
        (DomainFactory.createIncomingEmailNotification ());
```

```
        // set attribute values
        try
        {
              notif.setAttributeValue (INCOMING_EMAIL_MAIL_HOST,
                                       m_mailHost);
              notif.setAttributeValue (INCOMING_EMAIL_USER_NAME,
                                       m_userName);
              // etc…

              // using MessageBuilder to construct required data
              // objects
              MessageBuilder mb  = new MessageBuilder ();
              DataObject [] dobs = mb.addNotification (null, notif);

              // send the data objects to Aware IM!
              sourceProcess (dobs);
        }
        catch (Exception e)
        {
              // some error handling
        }
}
```

4.1.3 Handling replies to service requests

When a service of the external party or ***Aware IM*** has been requested the service provider may send a reply. The reply is sent as a special notification (called the *service reply notification*). This notification implements the `INotification` interface and exposes methods that allow checking the status of the service request and retrieve error message if there was any.

If ***Aware IM*** requests services of external parties, replies from the external parties need to be delivered back to ***Aware IM***. Rather than write the Source that would do this it is possible to use the special version of a Sink that would not only deliver service requests to the external party but also send the immediate replies back to ***Aware IM***.

Rather than extending `AbstractSimpleSink` class such a Sink must extend the `AttachedReplySink` class. `AttachedReplySink` has the Source, which ***Aware IM*** automatically attaches to the Sink when Sources and Sinks are constructed. The only extra functionality that the Sink extending `AttachedReplySink` has to provide is to feed the reply to the attached Source when the reply becomes available – this will guarantee that the reply will be delivered to ***Aware IM***. Feeding reply to the source is achieved by calling the `feedToSource` method of `AttachedReplySink` passing it the `DataObject's` as required by the Source.

The message fed to the source must contain the service reply notification. The service reply notification can be constructed calling the static method

```
INotification DomainFactory.createServiceReplyNotification
            (InotificationDefinition definition, int serviceStatus,
             String errorMessage);
```

where `definition` must correspond to the structure of the reply declared for the requested service (or `null` if a "standard" reply has been declared), `serviceStatus` is the constant described in the `getServiceStatus` method of the `INotification` interface and `errorMessage` is the message describing the error if service failed.

Once the notification has been created it can be converted to `DataObject's` using the `MessageBuilder` class and fed to the source. Here is the snippet of the code illustrating this:

```
public void processMessage (Message msg)
{
  // call the service here
  // …
  // construct service reply notification and feed it back to Aware IM
  INotification serviceReply = DomainFactory.createServiceReplyNotification
       (null, INotification.SERVICE_STATUS_SUCCESS, null);

  MessageBuilder mb = new MessageBuilder ();
  feedToSource (msg.peekDataObjects (),
                mb.addNotification (null, serviceReply));
}
```

4.1.4 MessageInterpreter class

The `MessageInterpreter` class can be used to obtain the notification or service name and parameters from a message sent by ***Aware IM*** to an external party. The following public methods can be used:

1. MessageInterpreter (DataObject [])

```
public MessageInterpreter (DataObject [] message)
```

Constructor. Construct MessageInterpreter from the array of `DataObject's` representing the message sent by ***Aware IM*** to an external party.

Parameters:

`message` – an array of DataObject's sent by ***Aware IM***. Usually obtained by calling `peekDataObjects` method of the `Message` class.

2. getNotification ()

```
public INotification getNotification ()
```

Retrieve the notification from the message.

Parameters: none
Returns: the notification contained in the message or `null` if the message represents a service request.

3. getServiceName ()

```
public String getServiceName ()
```

Retrieve the name of the requested service from the message.

Parameters: none
Returns: the service name of the requested service or `null` if the message represents a notification.

4. getServiceParameters ()

```
public Collection getServiceParameters ()
```

Retrieve parameters of the requested service from the message.

Parameters: none
Returns: parameters of the requested service as a collection of `IEntity` objects or `null` if the message represents a notification or if the service requires no parameters.

5. isServiceRequest ()

```
public boolean isServiceRequest ()
```

Parameters: none
Returns: `true` if the message represents a service request or `false` if the message represents a notification.

6. getChannelValues ()

```
public DataObject getChannelValues ()
```

Retrieve the data object containing values specific to the receiver of the message. It may happen that the same channel is used to send messages to different intelligent business objects (for example, an e-mail channel may be used to send e-mails to different recipients). If this is the case the message itself must contain the values, which uniquely identify the recipient (in the e-mail case it is the e-mail address of the recipient). The method `getChannelValues` allows retrieving values identifying the recipient from the message (see also section 4.1.2)

Parameters: none
Returns: DataObject containing the values specific to the recipient – the names of the attributes contained in the DataObject are channel-specific (see also section 4.1.2).

4.1.5 MessageBuilder class

The `MessageBuilder` class allows creating DataObject's in the format that ***Aware IM*** understands so that a message can be sent to ***Aware IM***. The following public methods can be used:

1. MessageBuilder ()

```
public MessageBuilder ()
```

Constructor.

Parameters: none

2. addServiceRequest (String, String, Object [])

```
public DataObject [] addServiceRequest (String componentName, String
                                        serviceName, Object [] parameters)
```

This method constructs a message representing a service request

Parameters:

componentName – this should always be `null`

serviceName – the name of the service to call (the service with this name must be exposed by the service provider). If a message is sent to ***Aware IM*** the name of the service is the name of one of the methods exposed by the `IExecutionEngineServices` interface (see JavaDoc API)

parameters – the parameters of the service as an array of `IEntity` objects or `null` if the service does not require any parameters. If a message is sent to ***Aware IM*** the parameters must correspond to the methods exposed by the `IExecutionEngineServices` interface (see JavaDoc API)

Returns: an array of `DataObject's` representing the message that can be sent to ***Aware IM***.

3. addNotification (EntityIdentifier, INotification)

```
public DataObject [] addNotification (EntityIdentifier receiver,
                                      INotification notification)
```

This method constructs a message representing a notification

Parameters:

receiver – this should always be `null`

notification – the notification to be sent

Returns: an array of `DataObject's` representing the message that can be sent to ***Aware IM***.

4.2 Implementing Channel Type component

The next step after writing the channel-specific Sources and Sinks is to write a class that implements the `IChannelType` interface. The implementation of the following methods must be provided:

1. getName ()

```
public String getName ()
```

This method must return the unique name of the channel within the ***Aware IM*** system. This is the name that can be used by the `executeEntityService` and `sendNotification` methods of the `IExecutionEngine` interface and the `REQUEST SERVICE` and `SEND` actions of the Rule Language (see "Aware IM Rule Language Reference"). This name is for system purposes only – it is not displayed in the list of available channels of the intelligent business object in the Configuration Tool.

2. getExternalName ()

```
public String getExternalName ()
```

This method must return the name by which the channel will be known to configurators. This name is displayed in the list of available channels of the intelligent business object. This name is not used by the system in any other way.

3. getOutputSinkClassName ()

```
public String getOutputSinkClassName ()
```

This method must return the fully qualified name of the Sink that handles communication with the external software system or device (see also section 4.1). If messages are never sent from ***Aware IM*** to the external software system or device (only received), `null` should be returned.

4. getInputSourceClassName ()

```
public String getInputSourceClassName ()
```

This method must return the fully qualified name of the Source that handles communication with the external software system or device (see also section 4.1). If messages from the external system or device are never received (only sent to), `null` should be returned.

5. getChannelSettingsEditor ()

```
public String getChannelSettingsEditor ()
```

This method must return the fully qualified name of the component that will provide the user interface to edit the channel-specific properties (see also section 4.3). If there are no channel-specific properties that need to be specified by the user, `null` should be returned.

6. getIntelligenceTypeApplicability ()

```
public int [] getIntelligenceTypeApplicability ()
```

This method must return those types of intelligent business object that the channel is applicable for. For example, the `SOAP` or `Sockets` channel is not applicable to business objects with the `Human` intelligence and the `E-mail` channel is not applicable to `Devices`. The constants specifying possible intelligence types are available in the `IEntityDefinition` interface. They are:

```
INTELLIGENCE_BUSINESS   - AWARE IM software system;
INTELLIGENCE_HUMAN      - human being;
INTELLIGENCE_DEVICE     - hardware device;
INTELLIGENCE_SYSTEM     - software system (non Aware IM)
```

At least one intelligence type must be returned.

7. multiUser ()

```
public boolean multiUser ()
```

This method must return `true` if communication with several intelligent business objects is possible through the channel. In this case a message sent through the channel must identify the recipient of the message. For example, the `E-mail` channel allows communicating with several recipients (a message contains an e-mail address of the recipient) whereas a dedicated channel handling the protocol specific to some software system is probably not multi-user.

If this method returns `true` intelligent business objects communicating through this channel must have a certain attribute defined (such as an e-mail address). When the message is sent to the recipient through the channel ***Aware IM*** automatically gets the current value of this attribute from the instance of the business object and writes it into the message. See also the `getChannelValues` method in the `MessageInterpreter` class and the `getReceiverEntityAttribute` method below.

8. getMessageValues ()

```
public DataObject getMessageValues (IchannelTypeCallback callback)
```

This method is called when a notification is sent to an intelligent business object. The method is supposed to return the values that are specific to the communication with this particular instance of the object – for example, an e-mail address of this instance. The method can use a callback interface to get the instance of the object and the instance of the SystemSettings object, like so:

```
IEntity intelligentEntity = callback.getEntity ();
IEntity systemSettings = callback.getSystemSettings ();
```

The method is supposed to return the data object containing specific values, or null if none are required. As an example, take a look at what the e-mail channel does:

```
      public DataObject getMessageValues (IChannelTypeCallback callback)
      {
            IEntity entity = callback.getEntity ();
            if (entity == null)
                  return null;

            try
            {
                  String emailAddr = (String) entity.getAttributeValue
(CommonConstants.EMAIL_ADDRESS_ATTR);

                  SDOType dobType = new SDOType ("EmailChannelEmailValues");
                  dobType.addAttribute (CommonConstants.EMAIL_ADDRESS_ATTR,
SDOType.STRING);
                  DataObject dob = new SimpleDataObject (dobType);

                  dob.setAttributeValue (CommonConstants.EMAIL_ADDRESS_ATTR,
emailAddr);
                  return dob;
            }
            catch (Exception e)
            {
                  System.err.println ("Unable to read e-mail address in object
" + entity.getName ());
            }

            return null;
      }
```

9. getDefaultChannelSettings ()

```
public Properties getDefaultChannelSettings ()
```

This method must return the properties that will be used by default for business objects communicating through the channel. The non-null value should generally be returned if there are settings for the channel that are editable by the user in the Configuration Tool. The returned

values will be used to initialize the channel settings in the user interface. The specific names of the properties are up to the channel to define – as long as the same names are used by the Channel Settings Editor and in the initialization of the corresponding Sources and Sinks – see sections 4.3 and 4.1). If the channel does not support any settings `null` should be returned. If the channel settings are supported but the initialization with blank values is required, the blank Properties object should be returned.

10. getSourceSettings ()

```
public Properties getSourceSettings (Properties channelSettings)
```

This method extracts the channel settings specific only to the Source supported by the channel from the combined settings that has properties of both Sink and Source. This may only be relevant if for whatever reason the Source cannot use the combined channel settings.

Parameters:

`channelSettings` – the channel settings prepared by the Channel Settings Editor that contain the settings for both Sink and Source

Returns: the properties that only have the settings specific to the Source. If this is not relevant the original settings may be returned.

11. getSinkSettings ()

```
public Properties getSinkSettings (Properties channelSettings)
```

This method extracts the channel settings specific only to the Sink supported by the channel from the combined settings that has properties of both Sink and Source. This may only be relevant if for whatever reason the Sink cannot use the combined channel settings.

Parameters:

`channelSettings` – the channel settings prepared by the Channel Settings Editor that contain the settings for both Sink and Source

Returns: the properties that only have the settings specific to the Sink. If this is not relevant the original settings may be returned.

12. supportsServiceDiscovery ()

```
public boolean supportsServiceDiscovery ()
```

Aware IM supports service discovery mechanism – it may send a special request through the channel to find out which services are supported by an intelligent business object. The

`supportsServiceDiscovery` method is supposed to return whether it supports the service discovery mechanism (for custom channels it should return `false`).

Parameters: none
Returns: `true` if services can be discovered through this channel

4.3 Implementing Channel Settings Editor

If a channel supports the specific settings that need to be specified by the user in the Configuration Tool a Channel Settings Editor needs to be written (the fully qualified name of the editor must be returned by the class implementing `IChannelType` interface – see section 4.1.2). The Channel Settings Editor is the class implementing the `IChannelSettingsEditor` interface.

The following methods must be implemented:

1. startEditing (int, java.awt.Component, Properties, boolean)

```
public Properties startEditing (int intelligenceType, java.awt.Component
                                parent, Properties initialSettings, boolean
                                viewOnly) throws Exception
```

This method is called when the user presses the `Settings` button to start editing the settings specific to the channel. The method is supposed to invoke the user interface (for example, a dialog) that would show the initial values of the channel settings and let the user edit these settings.

Parameters:

intelligenceType – the intelligence type of the business object for which the channel is configured. Possible values are specified in the `getIntelligenceTypeApplicability` method of the `IChannelType` interface.

parent – GUI component that should be the parent of the dialog invoked by this method.

initialSettings – the current values of the channel settings. When the Configuration Tool is started they are initialized with the values returned by the `getDefaultChannelSettings` method of the `IChannelType` interface and then they are set to the return of this method in each editing session.

viewOnly – if this flag is true the editor is not supposed to let the user modify any of the settings – all values must be read-only.

Returns: channel settings after they have been edited by the editor.

2. validateSettings (Properties, int)

```
public void validateSettings (Properties settings, int intelligenceType)
    throws Exception
```

This method is called when changes to the business object (which include changes to the channels) are committed. The editor is supposed to validate the settings and throw the exception with the appropriate message if the settings are invalid. The validation should happen during editing as well, however, this method may be useful when the editor was never invoked by the user in the first place so that the default settings are committed. The default settings for the channel, however, may not be complete and need validation.

Parameters:

intelligenceType – the intelligence type of the business object for which the channel is configured. Possible values are specified in the `getIntelligenceTypeApplicability` method of the `IChannelType` interface.

settings – the current values of the channel settings to be validated.

Throws: the exception if channel settings are invalid.

Note: If you want ***Aware IM*** to build the channel with the intelligent business object when the ***Aware IM*** server starts up so that ***Aware IM*** receives notifications from the business object, set the value of `CommonConstants.CHANNEL_SETTINGS_BUILD_NOTIF_CHANNEL` to be `"true"` in the channel settings.

Below is an example of the Channel Settings Editor that edits the settings of the `E-mail` channel:

```
public class EmailSettingsEditor implements IChannelSettingsEditor
{
      public EmailSettingsEditor()
      {}

      public Properties startEditing(
            int intelligenceType,
            Component parent,
            Properties initialSettings,
            boolean viewOnly)
            throws Exception
      {
            String hostServer  = null;
            String fromAddress = null;
            String userName = null, password = null;
            if (initialSettings != null)
            {
                  hostServer  = initialSettings.getProperty (MAIL_HOST);
                  fromAddress = initialSettings.getProperty
                                    (MAIL_FROM_ADDRESS);
```

```
            userName    = initialSettings.getProperty
                              (CHANNEL_SETTINGS_USER_NAME);
            password    = initialSettings.getProperty
                              (CHANNEL_SETTINGS_PASSWORD);
      }

      EmailSettingsDlg dlg = null;
      if (parent instanceof Frame)
      {
            dlg = new EmailSettingsDlg ((Frame) parent, hostServer,
                        fromAddress, userName,
                        password, viewOnly);
      }
      else if (parent instanceof Dialog)
      {
            dlg = new EmailSettingsDlg ((Dialog) parent, hostServer,
                              fromAddress, userName,
                              password, viewOnly);
      }

      dlg.show ();
      if (dlg.getReturnStatus () != EmailSettingsDlg.RET_OK)
            return null;

      Properties props = new Properties ();
      props.setProperty (MAIL_HOST, dlg.getHostName ());
      props.setProperty (MAIL_FROM_ADDRESS, dlg.getFromAddress ());
      props.setProperty (CHANNEL_SETTINGS_USER_NAME,
                         dlg.getUserName());

      props.setProperty (CHANNEL_SETTINGS_PASSWORD,
                         dlg.getPassword());
      return props;
}

public void validateSettings (Properties settings, int
                              intelligenceType)
      throws Exception
{
      String hostServer  = null;
      String fromAddress = null;
      if (settings != null)
      {
            hostServer  = settings.getProperty (MAIL_HOST);
            fromAddress = settings.getProperty (MAIL_FROM_ADDRESS);
      }

      if (hostServer == null || hostServer.length () == 0)
            throw new Exception ("Mail host name for e-mail channel
                  is not specified");

      if (fromAddress == null || fromAddress.length () == 0)
            throw new Exception ("Mail 'from' address for e-mail
                  channel is not specified");
}
```


5 Adding function libraries

This section describes what programmers have to do to add new function libraries to ***Aware IM***.

Function libraries contain functions that can be used in the Rule Language (see "Aware IM Rule Language Reference"). ***Aware IM*** includes some standard function libraries, such as the Date Function Library and Maths Function Library, which contain functions that work with date/time as well as some standard mathematical functions. You can find the complete list of functions supported by ***Aware IM*** in the "Aware IM Rule Language Reference". These functions should be sufficient for most standard needs. However in certain cases it may be necessary to provide new functions – for example, functions that perform calculations in different mathematical areas, some non-standard processing of strings etc.

Note that unlike processes functions are not supposed to communicate with the rest of the ***Aware IM*** system – they are not supposed to change attributes of business objects, update objects in the database etc. Functions are only supposed to perform calculations and return the result.

Functions are organized in libraries. The grouping of functions in a library is up to the designer of the library. The name of the function must be unique across all libraries, so it is generally a good idea to start the name of the function with a prefix uniquely identifying the library that the function belongs to.

To add a new function library to ***Aware IM*** follow the steps below:

1. Write the code for the component that will implement functions of the library (see section 5.1 for details)
2. Compile the source code of your function library component and package it in a jar file(s). Make sure that the jar file resides in the AwareIM/CustomJars directory (see section 2)
3. Add a definition of the new function library to the `BASServer.props` file located in the `BIN` directory of your ***Aware IM*** installation. You have to add a line that provides the fully qualified name of the function library component. The name of the property must start with the `FunctionLibrary` prefix and the value must be the fully qualified name of the component. For example:

```
FunctionLibraryStatistics=com.mypackage.StatisticsLibrary
```

If everything has been done correctly the name of the new functions must appear in the list of functions displayed by the Rule Language Context Assistant (see "Context Assistant" section in the "Aware IM User Guide")

5.1 Implementing function library

To implement a function library you need to write a class that implements the `IFunctionLibrary` interface. The implementation of the following methods must be provided:

1. getName ()

```
public String getName ()
```

This method must return the name of the library. The name must be unique within the ***Aware IM*** system.

Parameters: none
Returns: the name of the library.

2. hasFunction (String, int)

```
public boolean hasFunction (String functionName, int paramNmb)
```

This method must return `true` if the library contains the function with the specified name and number of parameters. ***Aware IM*** will call this method to find the library responsible for the calculation of a particular function referenced in a rule.

Parameters:

functionName – the name of the function

paramNmb – the number of function parameters

Returns: `true` if the function with the specified name and number of parameters is supported by the library, `false` otherwise.

3. getAvailableFunctions ()

```
public FunctionDescription [] getAvailableFunctions ()
```

This method must return all functions supported by the library as an array of structures containing the function name and the short description of what the function does.

Parameters: none
Returns: an array of structures containing the function name and the short description of what the function does. At least one function description must be returned.

4. calculate (String, Object [], INodeHelper)

```
public Object calculate (String functionName, Object [] parameters,
                         INodeHelper helper)
```

This method is what the function is all about – it performs the required calculation and returns the result.

Parameters:

functionName – the name of the function in the library that needs to be calculated

parameters – the values of function parameters as specified in a rule where the function is used. The given array contains Java objects corresponding to different attribute types (`String`, `Integer`, `DateHolder` etc – see a complete list in the Appendix A)

helper – a call-back interface. Should not be used by custom function libraries.

Returns: the result of the calculation as a Java object of the type corresponding to the attribute types – see Appendix A. If the function value cannot be calculated for whatever reason (for example, the number of parameters or their types are wrong), `null` should be returned.

5. toSQL (String, Collection, ISQLBuilderHelper)

```
public String toSQL (String functionName, Collection parameters,
                     ISQLBuilderHelper helper) throws Exception
```

This method must return the `SQL` representation of the function if it supports one. The `SQL` representation is essential if a function is to be used in queries. If the `SQL` representation is not provided it will not be possible to use the function in queries. See also an example later in this section.

Parameters:

functionName – the name of the function in the library that needs to be represented in the `SQL` form.

parameters – parameters of the function specified as a collection of the `IArithmeticNode` objects. To get an `SQL` representation of a parameter use the `toSQL` method of the `IArithmeticNode` interface.

helper – a call-back interface that contains certain methods that may help define the `SQL` representation. See the `ISQLBuilderHelper` interface.

Returns: a string representing the `SQL` expression that calculates the function value.
Throws: `UnsupportedSQLElementException` if the `SQL` representation for the function is not supported.

6. getTypeClass (String, Collection)

```
public Class getTypeClass (String functionName, Collection parameters)
```

This method must return the class of the object that is returned when the function is calculated – see `calculate` method.

Parameters:

functionName – the name of the function in the library the return type of which is required.

parameters – the parameters of the function specified as a collection of the IArithmeticNode objects.

Returns: the class of the object that is returned by the calculate method for the specified function.

7. getRequiredFactPatterns (String, Collection)

```
public HashSet getRequiredFactPatterns (String functionName, Collection parameters)
```

functionName – the name of the function in the library the return type of which is required.

parameters – the parameters of the function specified as a collection of the IArithmeticNode objects.

Returns: this method should have no implementation and return null.

5.1.1 ISQLBuilderHelper interface

This interface has quite a number of methods. Most of the methods, however, are used by ***Aware IM*** only and are of no interest to a function library developer. The only method that may be useful to developers of function libraries is the method that returns the specific database interface used by ***Aware IM***. The method has the following syntax:

1. getDatabaseInterface ()

```
public IDatabaseClientInterface getDatabaseInterface ()
```

This method returns the specific database currently used by ***Aware IM*** (see JavaDoc API for details).

Parameters: none
Returns: the database interface.

5.1.2 Function library example

See the code snippet below for an example of a function library. For illustration purposes this code snippet contains only a handful of functions supported by the standard Date Function Library.

```
public class DateFunctionLibrary implements IFunctionLibrary
{
	// supported date functions
	public static final String DAY_OF_WEEK_FUNCTION = "DAY_OF_WEEK";
	public static final String CURRENT_YEAR_FUNCTION = "CURRENT_YEAR";
```



```
public DateFunctionLibrary ()
{
}

/**
 * @see IFunctionLibrary#getName()
 */
public String getName ()
{
      return "DateFunctionLibrary";
}

/**
 * @see IFunctionLibrary#hasFunction(String, int)
 */
public boolean hasFunction(String functionName, int paramNmb)
{
      if (paramNmb == 0)
      {
            return functionName.equalsIgnoreCase
                        (CURRENT_YEAR_FUNCTION);
      }
      else if (paramNmb == 1)
      {
            return functionName.equalsIgnoreCase
                        (DAY_OF_WEEK_FUNCTION);
      return false;
}

/**
 * @see IFunctionLibrary#getAvailableFunctions()
 */
public FunctionDescription [] getAvailableFunctions ()
{
      FunctionDescription [] results = new FunctionDescription [2];

      results [0] = new FunctionDescription (CURRENT_YEAR_FUNCTION,
                                    "Parameters: none\nReturn:
                                     current year as 4 digit
                                     Number");
      results [1] = new FunctionDescription (DAY_OF_WEEK_FUNCTION,
                                    "Parameters: attribute of Date
                                     type or expression producing
                                     Date" +
                                     "\nReturn:     day of week of
                                     the provided date as Plain Text
                                    ('Monday', 'Tuesday' etc)");

      return results;
}

/**
 * @see IFunctionLibrary#calculate(String, Object[], INodeHelper)
 */
public Object calculate (String functionName, Object[] params,
                         INodeHelper helper)
{
```

```
        if (params == null || params.length == 0)
        {
              if (functionName.equalsIgnoreCase(CURRENT_YEAR_FUNCTION))
              {
                    Calendar cal = Calendar.getInstance ();
                    cal.setTime (new Date ());
                    return new Integer (cal.get (Calendar.YEAR));
              }
        }
        else if (params != null && params.length == 1)
        {
              // a single parameter must be of DateHolder type
              if (functionName.equalsIgnoreCase (DAY_OF_WEEK_FUNCTION))
              {
                    if (! (params [0] instanceof DateHolder))
                          return null;

                    Date date=TypeUtils.toDate((DateHolder)params[0]);

                    Calendar cal = Calendar.getInstance ();
                    cal.setTime (date);
                    SimpleDateFormat sdf = new SimpleDateFormat ("E");
                    return sdf.format (date);
              }
        }
        return null;
  }

  /**
   * @see IFunctionLibrary#toSQL(String, Collection, ISQLBuilderHelper)
   */
  public String toSQL (String functionName, Collection parameters,
                              ISQLBuilderHelper sqlHelper)
        throws Exception
  {
        IDatabaseClientInterface dbi = sqlHelper.getDatabaseInterface ();
        if (parameters == null || parameters.size () == 0)
        {
              if (dbi instanceof MySQLClientInterface)
              {
                    if (functionName.equalsIgnoreCase
                          (CURRENT_YEAR_FUNCTION))
                    {
                          return "YEAR(CURDATE())";
                    }
              }
              else if (dbi instanceof MSSQLServerClientInterface)
              {
                    if (functionName.equalsIgnoreCase
                          (CURRENT_YEAR_FUNCTION))
                    {
                          return "DATEPART (yy, GETDATE())";
                    }
              }
        }
        else if (parameters != null && parameters.size () == 1)
        {
              Iterator iter = parameters.iterator ();
```

```
                  IArithmeticNode arNode = (IArithmeticNode) iter.next ();

                  // representation of the argument
                  String dateArg = arNode.toSQL (sqlHelper);
                  if (dateArg == null || dateArg.length () == 0)
                        return "";  // the expression won't be calculated

                  if (functionName.equalsIgnoreCase (DAY_OF_WEEK_FUNCTION))
                  {
                        if (sqlFlavour == CommonConstants.SQL_FLAVOUR_TSQL)
                              return "DATENAME(dw," + dateArg + ")";

                        else if (sqlFlavour ==
CommonConstants.SQL_FLAVOUR_MYSQL)
                              return "DAYNAME(" + dateArg + ")";
                  }
            }
            throw new UnsupportedSQLElementException (functionName);
      }

      /**@see IFunctionLibrary#getTypeClass(String, Collection) */
      public Class getTypeClass (String functionName, Collection params)
      {
            return Integer.class;
      }

      /**@see IFunctionLibrary#getRequiredFactPatterns(String, Collection) */
      public HashSet getRequiredFactPatterns (String functionName, Collection
params)
      {
            return null;
      }
}
```

6 Adding custom document types

Aware IM comes with a number of predefined document types: - `Report`, `MS Word`, `Text`, `MS Excel`, `HTML`. These documents types should be sufficient for most applications. If, however, an application uses other document formats new document types can be added to ***Aware IM***.

Note that adding new document types is only necessary if you want to include tags in documents and you want ***Aware IM*** to replace tags with specific attribute values at run time. If you want to use documents without modifying them you can just attach a document in any format to a form of a business object that has an attribute of the `Document` type defined and ***Aware IM*** will store the document in the database. In this case ***Aware IM*** does not need to know about the internal structure of the document.

To add a new document type to ***Aware IM*** follow the steps below:

1. Write the code for the component that will work with documents of the new type on the client side (see section 6.1 for details).

2. Write the code for the component that will work with documents of the new type on the server side (see section 6.2 for details).
3. Compile the source code of your components and package them in a jar file(s). Make sure that the jar file resides in the AwareIM/CustomJars directory (see section 2).
4. Add the definition of a new document type to the `BASServer.props` file located in the `BIN` directory of your ***Aware IM*** installation. You have to modify the line that sets the `DocumentEngines` property to add the fully qualified name of your server side component. For example (note text in bold):

```
DocumentEngines=com.bas.basserver.documents.JasperReportsEngine,com.bas.basser
ver.documents.MSWordDocumentEngine,com.bas.basserver.documents.TextDocumentEng
ine,com.bas.basserver.documents.HtmlDocumentEngine,com.bas.basserver.documents
.MSExcelDocumentEngine,com.mypackage.MyServerDocEngine
```

Note that you have to make sure that the jar file with your server side component as well as the modifications to `BASServer.props` file are available to the Aware server that hosts your application (see "Aware IM Installation Guide" for details).

If everything has been done correctly the new document type must appear in the list of document types displayed by the Document Template Property Editor (see "Adding/Editing Document Templates" section in the "Aware IM User Guide").

6.1 Implementing client side component

The client side component of the new document type runs inside the Configuration Tool (as opposed to the server side component which runs inside the ***Aware IM*** server). The client side component is only responsible for the initialization and checking of the document template, while the server side component is responsible for generation of documents from document templates (replacing tags in templates with specific values).

The client side component is a class implementing the `IDocEngineClient` interface. The following methods must be implemented by the class:

1. getTemplateType ()

```
public String getTemplateType ()
```

This method must return a string that uniquely identifies the new document type in the ***Aware IM*** system. This is the string that is also displayed in the list of available document types in the Document Template Property Editor of the Configuration Tool.

Parameters: none
Returns: a string uniquely identifying the new document type.

2. getTemplateFileType ()

```
public String getTemplateFileType ()
```

This method must return the file extension specific to the new document type (without a dot) – for example, "txt" for text documents.

Parameters: none
Returns: a string identifying the document type file extension.

3. getEditCommand ()

```
public FileToEdit getEditCommand (String filePath)
```

This method must return the operating system specific command that should be invoked to edit a document template of the new type. This command is invoked when the user presses the `Edit` button in the Document Template Property Editor of the Configuration Tool.

Parameters:

`filePath` – the full path of the document template to be edited.

Returns: a structure that contains the command line and file identifier. The resulting command is formed by concatenating the command line with the file identifier (separated by the space symbol). If method returns `null`, the document template is not editable. If the command line returned as part of the `FileToEdit` structure is empty, the document template is editable but it is up to the user to select the editor for the document template (the Configuration Tool in this case will offer a dialog to select the program to edit the document template).

4. canHaveExternalResources ()

```
public boolean canHaveExternalResources ()
```

This method must return `true` if documents of the new type can have resources that are stored outside of the document file itself – for example, `HTML` documents can have image files stored separately whereas `MS Word` documents are self-contained.

Parameters: none
Returns: `true` if a document can have resources stored outside of the document file or `false` otherwise.

5. initialiseTemplate (DataPresentationTemplate, IDomainVersion)

```
public void initialiseTemplate (DataPresentationTemplate template,
                                IDomainVersion domainVersion)
      throws Exception
```

This method is called by ***Aware IM*** when a new document template is being created in the Configuration Tool. The method is supposed to find the attribute paths referenced by tags contained in the document template and set the information about these paths into the provided `DataPresentationTemplate` object using its `setAttributePaths` method.

Parameters:

`template` – the document template to set the attribute paths into

`domainVersion` – the business space version where the document template is defined

Throws: an exception if an error occurs or if the document template structure is invalid.

6. elementsRenamed (DataPresentationTemplate, RenamedElements, INodeChangeHelper, IDomainVersion)

```
public DataPresentationTemplate elementsRenamed (
                  DataPresentationTemplate oldTemplate,
                  RenamedElements renamedElements,
                  InodeChangeHelper helper,
                  IDomainVersion domainVersion)
```

This method is called by ***Aware IM*** when the user of the Configuration Tool renames some elements in the business space version. The method is supposed to check whether the renamed elements are referenced by tags contained in the document template and change the contents of the tags accordingly so that they reference the new names.

Parameters:

`oldTemplate` – the document template to be checked if it references the renamed elements

`renamedElements` – a structure that describes the elements that have been renamed by the user. The elements are grouped by categories corresponding to the different concepts supported by the business space version (business objects, processes etc). Each element in each category contains the old name of the element and the new name.

`domainVersion` – the business space version where the document template is defined

`helper` – a call-back interface containing the useful method `checkAttributePath`. This method can be used to check whether a particular attribute path referenced by the document template needs to be modified as a result of renaming. If the method returns `null` the attribute path need not be changed, otherwise the modified path is returned.

Returns: if the document template needs to be modified as a result of renaming the modified template must be returned otherwise `null` must be returned. Note that the method is not allowed to modify the original template directly – it must copy the original template first using the `deepCopy` method and then modify the copy.

7. createDynamicTemplate (String, String, byte [], String)

```
public DataPresentationTemplate createDynamicTemplate (
                  String templateName,
                  String fileName,
                  byte [] fileData,
                  String dirPath)
```

Implementers of this method are supposed to create the DataPresentationTemplate on the fly from the given parameters. They should also save the template file and all resources (if any) in the provided directory.

Parameters:

tempalteName – name of the template to create

fileName – name of the original file containing the template and possibly resources (if file represents an archive, for example).

fileData – data of the original file (it can represent the template itself or an archive containing the template and its resources)

dirPath – name of the directory where the template file and all its resources (if any) should be written.

Returns: newly create template.

6.2 Implementing server side component

The server side component of the new document type runs inside the ***Aware IM*** server (as opposed to the client side component which runs inside the Configuration Tool). The server side component is mainly responsible for generation of documents from document templates – it goes through document template and generates documents by replacing tags encountered in the document template with the appropriate values of expressions contained inside the tags.

The server side component is a class implementing the `IDocumentEngine` interface. The following methods must be implemented by the class:

1. getClientEngine ()

```
public IDocEngineClient getClientEngine ()
```

This method must return the component that implements the document type on the client side – see section 6.1.

Parameters: none
Returns: the instance of the client side component (must not be `null`)

2. fillWithData (String, Map, IDocumentDataSource, DataPresentationTemplate, IDocEngineHelper, IDomainVersion, int, Boolean, int)

```
public ReportResult fillWithData (
      String dirPath,
      Map params,
      IDocumentDataSource dataSource,
      DataPresentationTemplate template,
      IDocEngineHelper helper,
      IDomainVersion domainVersion,
      int outputFormat,
      boolean printResults,
      int numCopies) throws DocumentException
```

This method is called by ***Aware IM*** at run time when a particular document template is processed. The method is supposed to replace tags used in the document template with the appropriate attribute values. The values must be obtained from the provided data source – see the `IDocumentDataSource` interface. The overall structure of the method must be as follows:

1. Find the contents of the tags used in the document template.
2. Call `reset` method of the provided data source to position it on the first element.
3. Create a document (`BinaryResource`) from the original template that will have tags replaced with their values
4. Go through all the tags and call the `getExpressionValue` method of the provided data source to calculate the value of the expression used inside the tag
5. Replace each tag in the document with its calculated value
6. Call the `next` method of the data source. While there are elements in the data source repeat steps 3 to 5.

See also the example.

Parameters:

dirPath – the name of the directory where the template file is stored

params – can be ignored

dataSource – the source of data for the values of the tags. See the IDocumentDataSource interface.

template – the document template which is being filled with data

helper – can be ignored

domainVersion – the business space version where the document template is defined

outputFormat – can be ignored

printResults – if true the resulting document(s) should also be sent to the printer

numCopies – number of copies to print (only if "printResults" is true)

Returns: A structure represented by the ReportResult object. The first document created at the step 3 must be set into this structure as the BinaryResource. All other documents must be stored on disk as files. Full path names of the files must be set into the ReportResult structure.

Throws: the DocumentException if error occurs.

3. printFile ()

```
public void printFile (File fileToPrint) throws DocumentException
```

This method is supposed to print the specified file. It is assumed that the provided file is of the format supported by the document type (presumably obtained by calling the fillWithData method).

Parameters:

fileToPrint – document to print

Throws: the DocumentException if an error occurs.

6.3 DataPresentationTemplate class

This class represents the document template that most methods of the document type components work with. Below are the descriptions of some of the methods of this class that can be useful to the document type components (for the full list of methods see the JavaDoc API).

1. getLoadedTemplate ()

```
public BinaryResource getLoadedTemplate ()
```

This method returns the data for the document template as a BinaryResource. BinaryResource is a structure that contains the document file name and the binary data of

the document as an array of bytes. This method should be used by `fillWithData` method to retrieve the document template so that it can find tags and replace them with values.

Parameters: none
Returns: the document template as a `BinaryResource` structure.

2. getLoadedResources ()

```
public BinaryResource [] getLoadedResources ()
```

This method retrieves any resources that the document may have that are stored outside of the document template file (for example, images, fonts etc).

Parameters: none
Returns: the external resources as the `BinaryResource` structures or `null` if the document template has no external resources.

3. getAttributePaths ()

```
public String [] getAttributePaths ()
```

This method returns the attribute paths used inside tags in the document template. The attribute paths are represented in the form `objectName.attributeName`. Note that the paths returned are not necessarily the complete expressions used inside the tags. A tag expression may contain not only the attribute paths, but also arithmetic expressions with these paths. The paths used by the template would have been set by the `setAttributePaths` method.

Parameters: none
Returns: the attribute paths used in the document template or `null` if none are used.

4. setAttributePaths (String [] attributePaths)

```
public void setAttributePaths (String [] attributePaths)
```

This method sets attribute paths used inside tags in the document template. The attribute paths are represented in the form `objectName.attributeName`. Note that the paths returned are not necessarily the complete expressions used inside the tags. A tag expression may contain not only the attribute paths, but also arithmetic expressions with these paths. This method is usually called from the `initialiseTemplate` method of the `IDocEngineClient` interface.

Parameters:

attributePaths – the attribute paths used in the document template

5. deepCopy ()

```
public IConfigurationElement deepCopy ()
```

This method creates a "deep copy" of "this"

Parameters: none
Returns: copy of "this" as `IConfigurationElement` (which can be cast back to the `DataPresentationTemplate`).

6.4 IDocumentDataSource interface

This interface is used in the `fillWithData` function of the `IDocumentEngine` interface. The component implementing this interface provides access to the current values of object attributes in the Context and performs calculations with these values. There may be several instances of the business object in the Context and `IDocumentDataSource` allows iterating over all such instances. See also "Context of Rule Execution" section in the "Aware IM User Guide"

The methods of this interface that may be useful to document type components are described below:

1. reset ()

```
public void reset ()
```

This method positions the data source before the start of the first element. The subsequent call to the `next` method will position the data source on the first element. This method must be called at the start of iterations over the elements of the data source.

2. next ()

```
public boolean next ()
```

This method positions the data source on the next element if there is any. When called immediately after `reset` positions the data source on the first element.

Parameters: none
Returns: `true` if the method found next element to position itself on or `false` if there is no more data in the data source.

3. getExpressionValue (String)

```
public Object getExpressionValue (String expression) throws Exception
```

This method calculates the value of an expression inside a tag. The expression may be a reference to the object's attribute (attribute path) or a more complicated arithmetic expression. If the

expression contains the formatting expression, the result is converted to the specified format; otherwise it is converted to the format specified in the attribute (if a single attribute is used). The `fillWithData` method can use the `Object.toString ()` method to convert the result to `String` and put it in the output document instead of the tag.

Parameters:

`expression` – the text of the expression. This is the text that is contained inside a tag (not including the tag symbols)

Returns: the value of the expression - possible types of the Java object are presented in the table of the correspondence between attribute and Java types in the Appendix A. `Null` can be returned if the value is undefined.

Throws: an exception if the value cannot be calculated for whatever reason.

4. getExpressionValueForCalcs (String)

```
public Object getExpressionValueForCalcs (String expression) throws Exception
```

This method is very similar to the `getExpressionValue` method. The only difference is that the resulting value is not converted into any presentation format.

Parameters:

`expression` – the text of the expression. This is the text that is contained inside a tag (not including tag symbols)

Returns: the value of the expression - possible types of the Java object are presented in the table of the correspondence between attribute and Java types in the Appendix A.

6.5 Example

The code snippet below represents essentially the full text of the document engine that handles the text document type in ***Aware IM***:

```
// Client side component
public class TextEngineClient implements IDocEngineClient
{
	public TextEngineClient()
	{
	}

	/**
	 * @see IDocEngineClient#getTemplateType()
	 */
	public String getTemplateType ()
	{
		return CommonConstants.TEMPLATE_TEXT;
	}
```

```
	/**
	 * @see IDocEngineClient#getEditCommand(String)
	 */
	public FileToEdit getEditCommand (String templateFilePath)
	{
		return Utils.isWindows () ?
			new FileToEdit ("cmd.exe /k", "\"" + templateFilePath
					+ "\"") :
			new FileToEdit ("", templateFilePath);
	}

	/**
	 * @see IDocEngineClient#getTemplateFileType()
	 */
	public String getTemplateFileType()
	{
		return "txt";
	}

	/**
	 * @see IDocEngineClient#canHaveExternalResources()
	 */
	public boolean canHaveExternalResources ()
	{
		return false;
	}

	/**@see IDocEngineClient#initialiseTemplate (DataPresentationTemplate,
IDomainVersion) */
	public void initialiseTemplate (DataPresentationTemplate tdef,
IDomainVersion domainVersion)
		throws Exception
	{
		// determine attribute paths used
		String text = getTemplateText (tdef);
		if (text != null)
		{
			// find if there are any tags
			Vector taggedParams = Utils.extractParameters (text,
						"<<", ">>");
			if (taggedParams != null && taggedParams.size () > 0)
				DocumentEngineUtils.setAttributePaths
					(taggedParams, tdef);
		}
	}

	/**
	 * @see
com.bas.shared.docengines.IDocEngineClient#elementsRenamed(com.bas.shared.doma
in.configuration.elements.DataPresentationTemplate,
com.bas.shared.domain.configuration.misc.RenamedElements,
com.bas.shared.ruleparser.INodeChangeHelper,IDomainVersion)
	 */
	public DataPresentationTemplate elementsRenamed (
		DataPresentationTemplate oldTemplate,
		RenamedElements renamedElements,
		INodeChangeHelper helper,
		IDomainVersion domainVersion)
```

```
{
      String text = getTemplateText (oldTemplate);
      if (text == null)
            return null;

      // extract all parameters, convert them to ASTTagStatement
      // expressions, apply changes
      // and then replace parameters in the document
      Vector taggedParams = Utils.extractParameters(text,"<<",">>");
      if (taggedParams == null || taggedParams.size () == 0)
            return null;

      Vector modified = DocumentEngineUtils.getModifiedParameters (
                        taggedParams, renamedElements, helper);
      Vector withTags = DocumentEngineUtils.appendTags (modified,
                                    "<<", ">>");

      String newText = Utils.replaceParameters (text,withTags, "<<",
                                          ">>");
      if (newText.equals (text))
            return null;

      DataPresentationTemplate newTemplate =
            (DataPresentationTemplate) oldTemplate.deepCopy ();
      try
      {
            DocumentEngineUtils.setAttributePaths (modified,
                                          newTemplate);

            BinaryResource br = new BinaryResource
                        (oldTemplate.getTemplate ().getName (),
                  newText.getBytes ());
            newTemplate.setTemplate (br);
      }
      catch (Exception e)
      {
            return null;
      }  // shouldn't happen

      return newTemplate;
}

protected String getTemplateText (DataPresentationTemplate template)
{
      try
      {
            BinaryResource templData = template.getTemplate ();
            if (templData == null)
                  return null;

            byte [] data = templData.getData ();
            if (data == null)
                  return null;

            return new String (data);
      }
      catch (ResourceUnavailableException re)
      {
```

```
                return null;
            }
        }
}
```

// Server side component

```
public class TextDocumentEngine implements IDocumentEngine
{
      public TextDocumentEngine ()
      {
      }

      /**
       * @see IDocumentEngine#getClientEngine()
       */
      public IDocEngineClient getClientEngine ()
      {
            return new TextEngineClient ();
      }

      /**
       * @see IDocumentEngine#fillWithData(String, Map, IDocumentDataSource,
DataPresentationTemplate, IDocEngineHelper, IDomainVersion, int, boolean)
       */
      public BinaryResource [] fillWithData (String path, Map params,


      IDocumentDataSource dataSource,
      DataPresentationTemplate tdef,
      IDocEngineHelper helper,
      DomainVersion domainVersion,
      int outputFormat,
      boolean printResults)
            throws DocumentException
      {
            BinaryResource template = tdef.getLoadedTemplate ();
            if (template == null)
                  throw new DocumentException ("No data in the presentation
                                                template");

            String fileName = path + "/" + template.getName ();
            try
            {
                  File inputFile = new File (fileName);

                  String text = FileUtils.readTextFile (fileName);
                  // find if there are any tags
                  Vector taggedParams = Utils.extractParameters (text);

                  if (taggedParams == null || taggedParams.size () == 0 ||
                      dataSource == null)
                  {
                        // just return the original template
                        BinaryResource [] results = new BinaryResource [1];
                        results [0] = new BinaryResource ();
                        results [0].setName (inputFile.getName ());
                        results [0].setData (FileUtils.readFile
```

```
                                        (fileName));
                return results;
            }

            if (dataSource == null)
                throw new DocumentException ("Document template
                    requires data, but the data source is not
                    provided");

            // prepare the data source
            dataSource.reset ();

            // for each group of records create resulting document
            // and resolve tagged fields inside the text
            Vector resources = new Vector ();
            while (dataSource.next ())
            {
                Vector tagValues = new Vector
                    (taggedParams.size());
                for (Iterator iter = taggedParams.iterator ();
                    iter.hasNext (); )
                {
                    String tagContents = (String) iter.next ();
                    Object o = dataSource.getExpressionValue
                                    (tagContents);
                    tagValues.add (o==null? null : o.toString());
                }

                String newText = Utils.replaceParameters (text,
                                                    tagValues);
                if (printResults)
                    printText (newText);

                BinaryResource br = new BinaryResource ();
                br.setData (newText.getBytes ());
                br.setName (inputFile.getName ());
                resources.add (br);
            }

            if (resources.size () == 0)
                return null;

            BinaryResource [] results = new BinaryResource
                                            [resources.size ()];
            int i = 0;
            for(Iterator iter=resources.iterator();iter.hasNext();
                ++i)
                results [i] = (BinaryResource) iter.next ();

            return results;
        }
    catch (DocumentException de)
    {
        throw de;
    }

    /**
     * @see IDocumentEngine#printFile()
```

```
     */
    public void printFile (File fileToPrint)
                throws DocumentException
    {
          try
          {
                String text =
                      FileUtils.readTextFile(fileToPrint.getPath());
                printText (text);
          }
          catch (Exception e)
          {
                throw new DocumentException (e.getMessage ());
          }
    }

    protected void printText (String text)
          throws DocumentException
    {
          try
          {
                PlainDocument doc = new PlainDocument ();
                doc.insertString (0, text, null);

                TextDocumentPrintRenderer printer = new
TextDocumentPrintRenderer ();
                printer.print (doc, false);
          }
          catch (Exception e)
          {
                throw new DocumentException (e.getMessage ());
          }
    }
}
```

7 Adding report scriptlets

Aware IM offers a powerful tool called the Report Designer that allows users to configure their own reports and presentations of business objects and groups. Most reports and presentations can be configured without any programming as the Report Designer offers quite a rich set of versatile tools. Sometimes though it may be necessary to add the programming components to a particular report to perform some non-standard tasks – for example, calculate an image representing a chart of the report data.

Aware IM allows plugging in such programming components (scriptlets) into a particular report or presentation. To add a scriptlet to a report follow the steps below:

1. Write the code for the scriptlet (see section 7.1 for details)
2. Compile the source code of your scriptlet and package it in a jar file(s). Make sure that the jar file resides in the AwareIM/CustomJars directory (see section 2)
3. Specify the fully qualified name of the scriptlet in the Report Designer in the Configuration Tool. A scriptlet can either be attached to a text element of the report or to an image element.

Select the element that will be implemented by the scriptlet and in the Properties dialog for this element tick "Calculated by System" box. Press the Settings button and specify the fully qualified name of the component – see also the "Working with Report/Presentation Designer" section in the "Aware IM User Guide".

7.1 Implementing code for the scriptlet

The scriptlet's code is called by ***Aware IM*** at run time when it calculates the report in which the scriptlet is present. The purpose of the scriptlet component is to calculate and set the value of the element that it is attached to. When the scriptlet is called it is given a call-back interface IReportEnvironment representing the environment in which the scriptlet is running. This interface exposes the method setElementValue, which has to be called by the scriptlet when it finishes the necessary calculations.

The scriptlet is a class that must implement IReportScriptlet interface. Methods of this interface are called at various times during the report calculation. The scriptlet designer must provide a non-empty implementation of those methods that are called at the times that the scriptlet will be calculating its values. The methods exposed by the IReportScriptlet interface are described below. As mentioned before all methods are given a reference to the IReportEnvironment interface. Also all methods are supposed to throw an exception if error occurs.

1. beforeReportInit (IReportEnvironment)

```
public void beforeReportInit (IReportEnvironment environment)
      throws Exception
```

This method is called just before the report is initialized.

2. afterReportInit (IReportEnvironment)

```
public void afterReportInit (IReportEnvironment environment)
      throws Exception
```

This method is called immediately after the report has been initialized.

3. beforePageInit (IReportEnvironment)

```
public void beforePageInit (IReportEnvironment environment)
      throws Exception
```

This method is called just before each page is initialized.

4. afterPageInit (IReportEnvironment)

```
public void afterPageInit (IReportEnvironment environment)
      throws Exception
```

This method is called after the initialization of each page.

5. beforeColumnInit (IReportEnvironment)

```
public void beforeColumnInit (IReportEnvironment environment)
      throws Exception
```

This method is called before each column is initialized.

6. afterColumnInit (IReportEnvironment)

```
public void afterColumnInit (IReportEnvironment environment)
      throws Exception
```

This method is called after the initialization of each column.

7. beforeGroupInit (String, IReportEnvironment)

```
public void beforeGroupInit (String groupName, IReportEnvironment
                             environment) throws Exception
```

This method is called just before the specified group is initialized.

8. afterGroupInit (String, IReportEnvironment)

```
public void afterGroupInit (String groupName, IReportEnvironment
                            environment) throws Exception
```

This method is called after the specified group has been initialized.

9. beforeDetailEval (IReportEnvironment)

```
public void beforeDetailEval (IReportEnvironment environment)
      throws Exception
```

This method is called before evaluation of the details section of the report.

10. afterDetailEval (IReportEnvironment)

```
public void afterDetailEval (IReportEnvironment environment)
      throws Exception
```

This method is called after evaluation of the details section of the report.

7.1.1 IReportEnvironment interface

Descriptions of the methods that can be used by a scriptlet are described below:

1. getElementValue (String)

```
public Object getElementValue (String elementName)
      throws Exception
```

This method returns the current value of the report element with the specified name. The element must be either an element with a scriptlet attached (text or image) or a tag element, in which case the current value of the tag expression is retrieved. Each element in the report may be given a name in its Properties dialog. If the element's name is not specified ***Aware IM*** generates its own name. If a scriptlet needs to access certain elements in the report it is a good idea to give such elements some meaningful names.

Parameters:

`elementName` – the name of the element the value of which is retrieved

Returns: the value of the element – either `String` representing the text if the element is a text element or an Object representing the value of the tag expression of any type listed in the table in Appendix A if the element is a tag element or `java.awt.Image` if the element is an image.

Throws: an exception if the element with the specified name was not found or there was some other error.

2. setElementValue (String, Object)

```
public void setElementValue (String elementName, Object value)
      throws Exception
```

This method sets the current value of the report element with the specified name. The element must be an element with a scriptlet attached (text or image). Each element in the report may be given a name in its Properties dialog. If the element's name is not specified ***Aware IM*** generates its own name. If a scriptlet needs to access certain elements in the report it is a good idea to give such elements some meaningful names.

Parameters:

`elementName` – the name of the element the value of which is set

`value` – the value to set – must be either a `String` if the element is a text element or `java.awt.Image` if the element is an image.

Throws: an exception if the element with the specified name was not found or there was some other error.

7.2 Example

The following example shows the code for a simple scriptlet that accumulates names encountered in the report and provides them in a single comma separated string. It also displays a static image. It is assumed that the report in which the scriptlet executes has the following elements:

- Tag element with the name “FirstName”
- Tag element with the name “Last Name”
- Text element with the name “NamesList” (the scriptlet is attached to this element)
- Image element with the name “Image” (the scriptlet is also attached to this element)

```
public class TestScriptlet implements IReportScriptlet
{
	private boolean m_firstTime = true;

	public TestScriptlet()
	{
	}

	/*
	 * @see
com.bas.basserver.documents.IReportScriptlet#beforeReportInit(com.bas.basserve
r.documents.IReportEnvironment)
	 */
	public void beforeReportInit(IReportEnvironment environment)
		throws Exception
	{
	}

	/*
	 * @see
com.bas.basserver.documents.IReportScriptlet#afterReportInit(com.bas.basserver
.documents.IReportEnvironment)
	 */
	public void afterReportInit(IReportEnvironment environment)
		throws Exception
	{
	}

	/*
	 * @see
com.bas.basserver.documents.IReportScriptlet#beforePageInit(com.bas.basserver.
documents.IReportEnvironment)
	 */
	public void beforePageInit(IReportEnvironment environment)
		throws Exception
	{
	}

	/* (non-Javadoc)
	 * @see
com.bas.basserver.documents.IReportScriptlet#afterPageInit(com.bas.basserver.d
ocuments.IReportEnvironment)
	 */
	public void afterPageInit(IReportEnvironment environment)
		throws Exception
```

```
	{
	}

	/*
	 * @see
com.bas.basserver.documents.IReportScriptlet#beforeColumnInit(com.bas.basserve
r.documents.IReportEnvironment)
	 */
	public void beforeColumnInit(IReportEnvironment environment)
		throws Exception
	{
	}

	/*
	 * @see
com.bas.basserver.documents.IReportScriptlet#afterColumnInit(com.bas.basserver
.documents.IReportEnvironment)
	 */
	public void afterColumnInit(IReportEnvironment environment)
		throws Exception
	{
	}

	/*
	 * @see
com.bas.basserver.documents.IReportScriptlet#beforeGroupInit(java.lang.String,
com.bas.basserver.documents.IReportEnvironment)
	 */
	public void beforeGroupInit(
		String groupName,
		IReportEnvironment environment)
		throws Exception
	{
	}

	/*
	 * @see
com.bas.basserver.documents.IReportScriptlet#afterGroupInit(java.lang.String,
com.bas.basserver.documents.IReportEnvironment)
	 */
	public void afterGroupInit(
		String groupName,
		IReportEnvironment environment)
		throws Exception
	{
	}

	/*
	 * @see
com.bas.basserver.documents.IReportScriptlet#beforeDetailEval(com.bas.basserve
r.documents.IReportEnvironment)
	 */
	public void beforeDetailEval(IReportEnvironment environment)
		throws Exception
	{
	}

	/*
```

```
       * @see
com.bas.basserver.documents.IReportScriptlet#afterDetailEval(com.bas.basserver
.documents.IReportEnvironment)
       */
      public void afterDetailEval(IReportEnvironment environment)
            throws Exception
      {

            // here all report elements have been calculated already

            String firstName = (String) environment.getElementValue
                                          ("FirstName");
            String lastName  = (String) environment.getElementValue
                                          ("LastName");
            String curList   = (String) environment.getElementValue
                                          ("NamesList");
            if (curList == null)
                  curList = "";

            // create the string with names
            if (m_firstTime)
                  m_firstTime = false;
            else
                  curList = curList + ",";

            // set the value for our element
            environment.setElementValue ("NamesList", curList + firstName +
                        " " + lastName);

            // set the value for another element
            ImageIcon image = new ImageIcon
(getClass().getResource("/com/bas/uiconfiguration/images/arrowLeft.gif"));
            environment.setElementValue ("Image", image.getImage ());
      }

}
```

8 Writing client-side plugins

In ***Aware IM*** you can not only add plugins for the server (such as custom processes, channels or functions), but you can also add plugins that execute on the client within a web browser. Most of the time you would write these plugins in order to add your custom user interface functionality, or modify the default ***Aware IM*** user interface behaviour.

All client-side plugins must be written in Javascript and in most cases you need the knowledge of the Kendo UI Javascript library from Telerik (www.telerik.com/**kendo-ui**) and a popular open source Javascript library called jQuery (https://**jquery**.com/) . The description that follows assumes that the reader is familiar with Javascript, Kendo UI library and jQuery.

There are several types of the client-side plugins you can add in ***Aware IM***:

- Modify the default behaviour and presentation of forms
- Modify the default behaviour and presentation of form sections in forms
- Modify the default presentation of individual fields within forms

- Modify the default presentation and behaviour of queries
- Modify the default presentation and behaviour of content panels inside visual perspectives

We will look at each of these client-side plugins separately

8.1 Architecture of the client-side code

Before we explain how to write scripts for different components it is useful if a developer understands roughly the general architecture of the client-side code.

This is what happens behind the scenes when a screen is displayed in the browser by ***Aware IM***. A screen (usually represented by visual perspectives) consists of multiple queries, forms, content panels and so on (we will call them "components").

Each component, such as a form or a query is handled by the appropriate "controller" (also called "parser"). The fist thing the controller does is ask the server to provide the definition of the layout of the component. Then the controller parses the XML returned by the server and prepares two Javascript objects:

- jQuery object containing the HTML of the component (form or query). This object is called "markup".
- An array of "widget configurations". Each member of the array represents configuration for some widget of the Kendo UI library that the component includes. For example a query usually just includes a single widget that implements a query, such as the grid widget, for example. But a form may have a number of widgets – almost one per each attribute displayed by the form

Then the screen to be displayed is assembled from HTML markups of different components that the screen contains and the final HTML for the screen is built. This HTML is then given to the Kendo UI library, which creates all widgets of the screen based on the screen HTML and configurations of the widgets prepared by the controllers. Kendo UI library modifies this HTML to add its own classes and performs other steps to ensure that its widgets are displayed correctly. Finally the resulting HTML document is given to the browser which draws it on the screen.

To summarize:

1. A screen consists of components. Each component is represented by its own controller
2. The process starts by each controller asking the server for the definition of components
3. The controller then prepares HTML markup of the component and configurations of containing widgets
4. The HTML of the screen is created from the markups of components returned by controllers
5. The HTML and widget configurations are then given to the Kendo UI library to prepare its widgets
6. Final HTML of the screen is produced and is drawn by the browser.

So where in this process do the client scripts come in? For most components there are two types of scripts – "initialization" script and "render" script. As a developer you can define one or both – depending on what the script needs to do. The initialization script if defined is run by ***Aware IM*** just before the controller of the component returns the markup of the controller and the array of

widget configurations (so immediately after step 3 above). The script, therefore, has a chance to modify the markup returned by the controller or the configuration of any of the widgets of the component.

The markup can be modified using jQuery functions that manipulate HTML. The script can only modify the markup for the component, but not the entire screen, because the entire screen hasn't been built yet.

Widget configurations represent Javascript objects with properties described by Kendo UI API Reference. For example, to see the API Reference of the Kendo UI grid widget that implements ***Aware IM*** queries in the standard form go to http://docs.telerik.com/kendo-ui/api/javascript/ui/grid and look up the Configuration section at the top. Note that you can only modify configuration of the widget (which also includes Events), but you cannot use Methods of the widget in the initialization script.

The render script, though, runs after everything has been drawn on the screen – i.e. after step 6 above. By this time all Kendo UI widgets will have been already created, so the script can access the widget and call its methods (see the Methods section in the Kendo UI API Reference for each widget). Configuration objects cannot be used at this stage.

The render script can also access the final browser document and manipulate it if need be using jQuery functions. The following sections describe how this can be done in more detail.

8.2 Modifying default behavior and presentation of queries

To modify the default behavior and presentation of queries you need to go to a particular query that you want to modify and click on the "Scripts" property in the list of properties of the query. You can define "initialization" script or "render" script or both (see "Architecture of the Client-side Code")

The following objects are exposed to the initialization script:

1. "config" object - this object represents Kendo UI configuration of the main widget implementing the query (see the table below)
2. "markup" object – HTML markup prepared by the controller
3. "parser" object – the controller itself
4. "widgets" – an array of all widget configurations for the query

Query presentation type	Kendo UI widget	Kendo UI reference
Standard	Grid	http://docs.telerik.com/kendo-ui/api/javascript/ui/grid
Custom (Custom Data Template, scroll view unticked)	List View	http://docs.telerik.com/kendo-ui/api/javascript/ui/listview
Custom	Scroll View	http://docs.telerik.com/kendo-ui/api/javascript/mobile/ui/scrollview

(Custom Data Template, scroll view ticked)		
Custom (Mobile Data Template)	Mobile List View	http://docs.telerik.com/kendo-ui/api/javascript/mobile/ui/listview
Calendar/Scheduler	Scheduler	http://docs.telerik.com/kendo-ui/api/javascript/ui/scheduler
Chart	Chart	http://docs.telerik.com/kendo-ui/api/javascript/dataviz/ui/chart
Gantt	Gantt	http://docs.telerik.com/kendo-ui/api/javascript/ui/gantt

It is usually unnecessary to modify the markup, but you are welcome to modify the configuration of the widget.

For example, the following script will change the alignment of the column that corresponds to the "Status" attribue:

```
for (var i = 0; i < config.columns.length; ++ i)
{
     if (config.columns[i].field == "Status")
     {
          config.columns[i].attributes = { alignment: "right" }
     }
}
```

The following script will make the grid "groupable", i.e allow dragging the columns to a special area in order to group the grid by this column:

```
config.groupable = true;
```

The "parser" object represents the controller and allows you to access certain properties of the system that you may need. The type of this object depends on the type of the query representation and is provided in the table below:

Query presentation type	Aware IM Javascript object type	Source code
Standard	AwareApp_QueryLayoutParser	AwareIM/Tomcat/webapps/AwareIM/aware_ext/parsers/queryLayoutParser.js
Custom (without scroll view)	AwareApp_CustomLayoutParser	AwareIM/Tomcat/webapps/AwareIM/aware_ext/parsers/customLayoutParser.js
Custom (with scroll view)	AwareApp_ScrollViewParser	AwareIM/Tomcat/webapps/AwareIM/aware_ext/parsers/scrollViewParser.js
Custom (with mobile template)	AwareApp_MobileListViewParser	AwareIM/Tomcat/webapps/AwareIM/aware_ext/parsers/customMobileParser.js

Chart	AwareApp_ChartParser	AwareIM/Tomcat/webapps/AwareIM/aware_ext/parsers/chartParser.js
Calendar/Scheduler	AwareApp_CalendarParser	AwareIM/Tomcat/webapps/AwareIM/aware_ext/parsers/calendarParser.js
Gantt	AwareApp_GanttParser	AwareIM/Tomcat/webapps/AwareIM/aware_ext/parsers/ganttParser.js

The "widgets" object representing an array of all widgets that the query has can be used to modify other widgets – for example, toolbars generated to represent query operations (if these are defined for the query).

Each member of the "widgets" array has the following properties:

- type (the type of the Kendo UI widget)
- id (the unique id in the markup of the query that the widget uses)
- config (the Kendo UI configuration of the widget)

So to modify a toolbar, for example (http://docs.telerik.com/kendo-ui/api/javascript/ui/toolbar) and stop it being resizable you would find the toolbar in the array and modify its "config" property like this:

```
for (var i = 0; i < widgets.length; ++ i)
{
     if (widgets[i].type == "toolbar")
          widgets[i].config.resizable = false;
}
```

The second script ("render" script) that you can define in the Scripts dialog allows you to modify the widget representing the query after it has been drawn. Here you would use methods of the corresponding Kendo UI object, rather than configuration options. Objects available for your Javascript here are:

- "widget" object
- "parser" object

The "widget" object represents the widget that has been drawn.

The "parser" object is the controller object described above.

8.3 Modifying default behavior and presentation of forms

To modify the default behavior and presentation of forms you need to go to a particular object form that you want to modify and click on the Scripts property under the "Advanced" category in the list of properties of the form. You can define "initialization" script or "render" script or both (see "Architecture of the Client-side Code").

The Javascript objects that are exposed to the initialization script for forms are the same as for queries:

- “config” object
- “markup” object
- “parser” object
- “widgets” object

Just like with queries, the parser object represents the form controller and the “markup” object represents the HTML markup of the form. However, there is a signifant difference in the “config” object exposed by forms. Whereas for queries the “config” object represents some widget of the Kendo UI library, that is mainly responsible for the implementation of the query, there is no such widget for forms. A form is just an HTML code that consists of rows and columns of the Bootstrap grid system (get**bootstrap**.com/css/). Each row and column contains attributes of the form – depending on the type of the attribute they are either implemented as plain HTML or they may also include a configuration for a Kendo UI widget.

This HTML is also wrapped in a “panel” that includes HTML of toolbars around the form (if they are defined) and also the implementation of the default or custom panel header. Note that the HTML that includes toolbars and panel header is also included as part of the query markup.

So the “config” object for forms represents the configuration of a special Aware IM object called “panel”. You can find the code of this object in `AwareIM/Tomcat/webapps/AwareIM/aware_kendo/panel.js` file. At the beginning of the file there is a description of all configturation parameters supported by this object. You can change these parameters by your script..

For example, the following script will turn off the display of the header for the form, no matter what is specified in the ***Aware IM*** properties of the form:

```
config.preventHeader = true;
```

However, you are unlikely to need it because most of these parameters can be customized in ***Aware IM*** without having to use a script.

You may, however, want to modify the markup of the form – for example, you may want to modify the generated layout or styling of some attributes. You can use it through the “markup” object (or in the “render” script after the form has been drawn). The description of the HTML markup is beyond the scope of this document. If you want to study it you can just generate a form and use the browser inspector to display the HTML of the form. You can then use your scripts to modify this markup.

You can, however, modify the widgets used by the form using the “widgets” object – you will probably modify widgets that represent attributes in their own Advanced scripts, but you can also modify other widgets used by a form, such as toolbars, for example – this has been already explained in the previous section.

The “parser” object allows you to access certain properties of the system that you may need (especially if you are modifying the markup of the form). The type of this object is

`AwareApp_FormParser`. You can look up full methods of this object in the file `AwareIM/Tomcat/webapps/AwareIM/aware_kendo/parsers/formParser.js`

Some useful methods of the parser object that you can use here are:

- `getField (attributeName, sectionName)` – get the field of the form for the specified attribute and form section
- `getReferenceParser (refAttrName)` – get the "parser" of the reference attribute of the form, responsible for displaying a table, calendar etc – see 8.1

The second script ("render" script) that you can define in the Scripts dialog allows you to modify the form after it has been drawn. Here you can only use jQuery methods to modify the resulting document markup. The only object available for you here is the "parser" object representing the form controller.

8.4 Modifying default behavior and presentation of form sections

Modifying the default behavior of a form section only makes sense if a form has more than one section defined. If a form has one section than you should modify the behavior of the form, not section, as described in the previous section. To modify the default behavior and presentation of a form section you need to go to a particular form section of the form and click on the "Scripts" property under the "Advanced" category in the list of properties of the form section.

The rules for defining the scripts are very similar to defining the scripts for forms, the only difference being the "parser" object. The type of this object is `AwareApp_FormSectionParser` rather than `AwareApp_FormParser` and the source code of the object is available in this file: `AwareIM/Tomcat/webapps/AwareIM/aware_kendo/parsers/formSectionParser.js`

8.5 Modifying default presentation of individual fields on forms

To modify the default presentation of an individual field on an object form you need to go to the presentation properties of the corresponding attribute and click on the "Advanced" property. There is only one script available for you here.

As explained in the "Architecture of the client-side code" the controller of the form prepares the HTML markup of the form as well as the list of Kendo UI widgets that the form includes. Apart from other things the markup of the form contains markups of individual fields present on the form. And the collection of widgets for the form includes widgets used by individual fields (note that not all fields use widgets, some use markup only). Each type of field on the form is represented by its own ***Aware IM*** Javascript object (see the table below).

The form controller asks every individual field on the form to prepare its markup and the collection of widgets. Then it assembles the result into the final markup and widget collection of

the form. The script for each indivisual field is executed just before it is given to the form controller, so that the script has a chance to modify the markup or widget configuration.

There are three objects exposed to the script:

- "field" – this is Aware IM object representing the field (see the table below)
- "markup" – this is the HTML markup of the field (jQuery object)
- "config" – this is the object that represents a widget configuration of the field or null if the field does not use a widget. The object has the following properties:
  - "type" – type of the widget
  - "id" – the id of the element in the markup used by the widget
  - "config" – the Kendo UI configuration of the widget

You can modify modify any of these object. For example, if you want to hide the field you can write the following script:

```
markup.css ("display, "none");
```

Or if you want to change which tools are available for an HTML editor field (see http://docs.telerik.com/kendo-ui/api/javascript/ui/editor#configuration-tools), you could write the following script:

```
config.config.tools = ["bold", "italic", "underline"]
```

There are some useful methods of the "field" object that you can use in your script (the code of all all the objects representing different fields is in `AwareIM/Tomcat/webapps/AwareIM/aware_kendo/field/fields.js` file.

- `field.getAttributeName ()` – retrieve the name of the object attribute
- `field.getObjectName ()` – retrieve the name of the object
- `field.getObjectId ()` – retrieve the id of the object

and so on.

Note that if you want to access the field after it has already been drawn you need to find the field on the form and so you need to modify the "render" script of the form, like so, for example:

```
var field = parser.getField ("Account", "Main");
var value = field.getValue ();
```

The following table lists all different field types and the corresponding Kendo UI widgets.

Aware IM attribute type	Kendo UI widget	Aware IM field object
Plain Text (no choices, 1 line)	None	AwareApp_TextField
Plain Text (no choices, several lines)	None	AwareApp_TextAreaField
Plain Text, Number, Date with radio or checkbox choices	NumericText Box (http://docs.telerik.com/kendo-ui/api/javascript/ui/numerictextbox)	AwareApp_NumberField

Plain Text, Number, Date with text choices, choices not editable	DropDownList (http://docs.telerik.com/kendo-ui/api/javascript/ui/dropdownlist)	AwareApp_ComboField
Plain Text, Number, Date with text choices, choices editable	ComboBox (http://docs.telerik.com/kendo-ui/api/javascript/ui/combobox)	AwareApp_ComboField
Date without choices	DatePicker (http://docs.telerik.com/kendo-ui/api/javascript/ui/datepicker)	AwareApp_DateField
Timestamp	DateTimePicker http://docs.telerik.com/kendo-ui/api/javascript/ui/datetimepicker	AwareApp_DateTimeField
Yes/No (displayed as checkbox)	None	AwareApp_CheckboxField
Yes/No (displayed as a switch)	Switch http://docs.telerik.com/kendo-ui/api/javascript/mobile/ui/switch	AwareApp_SwitchField
Plain Text with choices represented as checkboxes	None	AwareApp_CheckboxGroupField
Plain Text with choices represented as radio buttons	None	AwareApp_RadioButtonGroupField
PlainText with multi-selector	MultiSelect http://docs.telerik.com/kendo-ui/api/javascript/ui/multiselect	AwareApp_TagField
PlainText with HTML format	Editor http://docs.telerik.com/kendo-ui/api/javascript/ui/editor	AwareApp_HtmlEditorField
Document	Upload http://docs.telerik.com/kendo-ui/api/javascript/ui/upload	AwareApp_DocumentField
Picture (not represented as signature)	Upload http://docs.telerik.com/kendo-ui/api/javascript/ui/upload	AwareApp_PictureField
Picture (represented as signature)	None	AwareApp_SignatureField
Shortcut	None	AwareApp_ShortcutField
Reference represented by a drop down	DropDownList (http://docs.telerik.com/kendo-ui/api/javascript/ui/dropdownlist)	AwareApp_SelectReferenceField
Reference represented by a multi-selector	MultiSelect http://docs.telerik.com/kendo-ui/api/javascript/ui/multiselect	AwareApp_TagReferenceField
Reference represented by a “swap select”	None	AwareApp_SwapSelectField
HTML cell	None	AwareApp_HtmlFiield
Gauge cell (linear)	LinearGauge http://docs.telerik.com/kendo-ui/api/javascript/dataviz/ui/lineargauge	AwareApp_GaugeField
Gauge cell (radial)	RadialGauge http://docs.telerik.com/kendo-ui/api/javascript/dataviz/ui/radialgauge	AwareApp_GaugeField
Google Map cell	None	AwareApp_GoogleMapField
Number displayed as slider	Slider http://docs.telerik.com/kendo-ui/api/javascript/ui/slider	AwareApp_SliderField

8.6 Modifying default behavior of menu in visual perspectives

The idea here is very similar. You have two scripts available – initialization and render scripts. The initialization script has a chance to modify the configuration of the menu widgets (almost all menu types except Plain List are implemented by their own Kendo UI widget (see the table below). The render script can call the methods of the widget once it has been drawn.

The following objects are exposed to the initialization script:

- "config" – this object represents Kendo UI configuration of the menu widget
- "parser" – the controller object (AwareApp_VPParser) – see the code in the file `AwareIM/Tomcat/webapps/AwareIM/aware_kendo/parsers/vpParser.js`

For example to add some custom menu item to a toolbar menu you could write the following script:

```
config.items.push ({
    type: "button",
    spriteCssClass: "fa fa-edit",
    text:  "My Menu Item",
    click: function () {
        alert ("this is my menu item");
   }
});
```

Menu type	Kendo UI widget	Kendo UI reference
Toolbar	ToolBar	http://docs.telerik.com/kendo-ui/api/javascript/ui/toolbar
Standard Menu	Menu	http://docs.telerik.com/kendo-ui/api/javascript/ui/menu
Panel Bar	PanelBar	http://docs.telerik.com/kendo-ui/api/javascript/ui/panelbar
Tree	TreeView	http://docs.telerik.com/kendo-ui/api/javascript/ui/treeview

8.7 Modifying default behavior and presentation of content panels in visual perspectives

To modify the default behavior and presentation of content panels in visual perspectives you need to go to a particular visual perspective that you want to modify, select the content panel and then click on the "Scripts" property in the list of properties of the content panel.

The idea here is the same – you have two scripts as before. However, there are no Kendo UI widgets to modify here – you can only modify the markup of the content panel (either during initialization or after it has been drawn).

Two objects are exposed for the initialization script:

- "config" – this is the configuration of the Aware IM "panel" object, the code of which is in `AwareIM/Tomcat/webapps/AwareIM/aware_kendo/panel.js`

The markup of the panel is stored in the "bodyContent" property of the object config.bodyContent. This is the markup you are most likely to modify here
"parser" object – this is the controller (AwareApp_Dashboard in `AwareIM/Tomcat/webapps/AwareIM/aware_kendo/parsers/dashboard.js`

8.8 AwareApp object

When writing advanced scripts as described above you can use the `AwareApp` Javascript object that contains some useful static methods. This is an example of calling one of these methods:

```
var panelId = AwareApp.getPanelId ("main", "Accounts", "My Accounts");
```

The code of the object is located here:
AwareIM/Tomcat/webapps/AwareIM/aware_ext/awareApp.js

The following methods can be used:

1. getPanelId (frameName, tabName, contentPanelName)

This method returns the unique id of the content panel in a visual perspective.

```
var id = AwareApp.getPanelId ('main', 'Tab1', 'Content Panel1');
```

Parameters:
`frameName` – name of the frame in the visual perspective that contains the panel
`tabName` – name of the tab inside the frame that contains the panel
`contentPanelName` – name of the content panel

2. getFramePanelId (frameName)

This method returns the unique id of the frame in a visual perspective.

```
var id = AwareApp.getFramePanelId ('main');
```

Parameters:
`frameName` – name of the frame in the visual perspective

3. getTabPanelId (frameName, tabName)

This method returns the unique id of the tab in a visual perspective.

```
var id = AwareApp.getTabPanelId ('main', 'Tab1');
```


Parameters:
`frameName` – name of the frame in the visual perspective that contains the panel
`tabName` – name of the tab inside the frame that contains the panel

4. isRTL ()

Return true if the current user uses right-to-left layout

5. getMainTabPanel ()

If a visual perspectives has tabs return the tab panel holding the tabs.

6. getProcessId ()

Return the id of the currently running process or -1 if there are no processes currently running.

7. isTestingMode ()

Return true if the current user is running in the testing mode

8. startProcess, runQuery and other methods mentioned in the User Guide

Please refer to the "Links to Aware IM operations" section in the User Guide, that explains how to set up links to perform operations. All functions mentioned there can be used from your Javascript.

8.9 Using Javascript to integrate custom Cordova plugins for native mobile applications

Cordova plugins are components that provide access to some built-in features of mobile phones, (such as camera or contacts), for which there is no Javascript access. When components are integrated into the system these features become available through some special Javascript functions that the plugin makes available to the developer. Cordova plugins can only be used in native mobile applications.

Aware IM integrates some Cordova plugins out-of- the-box and provides rule actions that activate them (for example, MOBILE PUSH or MOBILE CAMERA SNAP INTO. However, there are many plugins around and it is impossible to integrate all of them into Aware IM.

Still there is a way to do this by adding some custom Javascript to your application. The following section explains how to do it.

This is the high level overview of what needs to be done:

1. Study the documentation of the Cordova plugin to fully understand Javascript methods that it exposes
2. Write the Javascript that calls the appropriate Javascript function that the plugin provides
    a. Give this function the data obtained from Aware IM if necessary. For example, read the data from the database and provide this data to the function of the plugin. The useful Aware IM function that can be used here is `AwareApp.getObjectData()`
    b. Handle the return of this function if necessary – for example write the data returned by the function to the database. Useful Aware IM functions for this are `AwareApp.createOrUpdateObject()` and `AwareApp.startProcessWithInit()`
3. Define panel operations or menu items in the mobile part of your business space version (using the Configuration Tool) that would run this Javascript. You should select an operation or menu item of the "Execute Javascript" type for this.
4. Build a native mobile application for your business space version using the "Build Native Mobile Application" command in the Configuration Tool. This will create a zip file.
5. Uzip this zip file somewhere. Find the `config.xml` file in the root of the unzipped application and open it for editing.
6. Find the section in this file that lists the plugins used by the application, for example: `<plugin name="cordova-plugin-camera" spec="2.0.0" />` Add the definition of the Cordova plugin you need to integrate – look up the documentation of the plugin for details of the plugin name and version number
7. Zip up the application again and use the PhoneGap build to create application files in the native format of the mobile phone

Let's look at an example. We will be integrating a Cordova plugin for Contacts into the CRM mobile sample application.

The documentation of the plugin can be found here: https://github.com/apache/cordova-plugin-contacts

As we can see the plugin provides the `navigator.contacts` object that can be used to create contacts, find existing contact or pick a particular one. Let's add the following functionality to the CRM application:

- From the form of a customer or from a customer list create a phone contact populated with the information from the customer record in the application
- Pick a contact from the list of phone contacts and send this contact an email that includes some information stored in the application

8.9.1 Creating a contact on the phone

We need to use the "create" method of the `navigator.contacts` object and provide contact data available in the customer record that we are parked on. Retrieving the data can be done using the `AwareApp.getObjectData` function. It has the following signature:


```
   getObjectData: function (objectName, objectId,
callBackFunction)
```

`ObjectName` and o`bjectId` identify the record to retrieve and c`allBackFunction` specifies a function that will be called when the data has been retrieved. The function will be called with the object storing the retrieved values.

How do we get object name and id? When we define an Aware IM operation of the "Execute Script" type Aware IM automatically defines the following objects that we can use in our Javascript:

- `parser`
- `context`

The `parser` object should be already familiar and the `context` object stores an array of objects with `objectName` and `objectId` attributes. The record we are parked on is the first and only one in this array. So to get `objectName` we use the following `context[0].objectName`; and to get object id we use `context[0].objectId`

So the Javasrcipt we need to write to create a contact looks like this:

```
AwareApp.getObjectData (
    context[0].objectName,
    context[0].objectId,
    function (objectData)
   {
         navigator.contacts.create ({
                "displayName": objectData["FirstName"] + " " +
objectData["LastName"],
                "birthday": kendo.parseDate
(objectData["DateOfBirth"], "dd/MM/yyyy", "en-US")
         });
    }
);
```

Note that here "displayName" and "birthday" are the names of the attribute of the Contact object on the phone exposed by the plugin, whereas "FirstName", "LastName" and "DateOfBirth" are the names of the attributes of the Customer object in the CRM application. Note also that all Aware IM attribute values are strings and if the plugin requires some other type (for example, date), then the strings need to be converted to the appropriate type.

The next step is to create operations of the "Execute Script" type to the form and customer list. We can add a panel operation to the "Editing Mobile" form of the Customer object and an operation with record to the "Customer – all mobile" query. We then specify the above script as a parameter of the operation.

8.9.2 Send email to the selected contact

We need to use the `pickContact` method of the `navigator.contacts` object to display a list of contacts, let the user pick one and then we need to start a process in the application to send an email to the email address of the contact picked by the user.

The email address returned by the plugin needs to be saved in some temporary object and then this object can be used in the process. So we will create a temporary business object (persistence type – memory) called `ContactParam` with the single EmailAddress attribute. We will then create a process called `SendEmailToContact` with the `ContactParam` object as its input. The process will then use the `SEND` action to send any email to this email address (the email can use tag expressions to retrieve some information from the system – for example, from `SystemSetting`s or from the logged in user).

To start a process we will use the `AwareApp.startProcessWithInit` function. It has the following signature:

```
startProcessWithInit: function (procName, renderOption, objName,
initValues, context)
```

Here `procName` is the name of the process to start, `renderOption` is where to display the results of the process (we can use `null`), `objName` is the name of the parameter object, `initValues` is the object storing values of the parameter object and context contains additional parameter objects (`null` in our case)

So our Javascript can look like this:

```
navigator.contacts.pickContact (function (contact)
{
   var email = contact.emals[0].value;
   AwareApp.startProcessWithInit (
      "SendEmailToContact",
      null,
     "ContactParam",
     { "EmailAddress" : email }
},
function (error { console.log (error); }
);
```

Then we just need to add a command of the "Execute Script" type to the mobile menu of the application.

Appendix A: correspondence of attribute types and Java types

The following table lists the mapping between the ***Aware IM*** attribute types and the Java types used in certain methods such as `getAttributeValue` and `setAttributeValue` of the `IObject` interface.

Attribute Type	Java Type
`Plain Text`	`String`
`Number`	`Long` or `Double` depending on the format of the `Number`
`Date`	`org.openadaptor.util.DateHolder`
`Timestamp`	`org.openadaptor.util.DateTimeHolder`
`Duration`	`com.bas.utils.Duration`
`Yes/No`	`Boolean`
Document or Picture	`org.openadaptor.dataobjects.DataObject` with the following sub-attributes: • `DOCDATA` (Binary) • `DOCTYPE` (Plain Text)
`Binary`	`byte []`

