Developing a Java Application using the CA Gen Studio JMMI Interface

Contents

Introduction

This paper shows how to develop an uncomplicated Java application accessing and updating models in the CA Gen local encyclopedia using the JMMI Application Program Interface (API) software for CA Gen.

The JMMI API currently supports local access to workstation models for Java based applications on a Windows platform. The JMMI API includes read and update classes. These classes provide the capability to extract and update model information from models in the local encyclopedia.

The API is designed to be able to access CSE model in the future. The JMMI API is a set of Java classes and Windows DLLs. You should be familiar with the Java programming language and the software development tools for the platform on which you are working in order to use the JMMI API. You should also be familiar with the CA Gen meta-model.

These API classes allow CA Gen customers and Open Initiative partners to write software to retrieve and update encyclopedia information.

Sample Application

Let us develop a sample application for the imaginary Company developing commercial applications using CA Gen. The Company would like to extend its Quality Assurance process so that action block descriptions can be reviewed by external professional technical writers. These technical writers do not have a license or the skills to use CA Gen and all materials should be provided in some agreed text format.

This paper shows how to develop two Java classes to create two Java applications to run from the command line.

`DescriptionExtractor`

This class extracts descriptions from all action blocks within the model. The Class stores descriptions in text files. By default the name of the file is the name of the action block with `txt` as a file extension. All text files are stored in the dedicated subdirectory `/doc` . The application creates a subdirectory in the directory where a local encyclopedia is located. The application has two parameters: the full path to the local encyclopedia and the name of the model in the local encyclopedia.

Tip 1

*We are using the term **local encyclopedia** which can be a little bit misleading. This is really the location of the local model on your workstation.*

`DescriptionReloader`

This class reloads action block descriptions stored in text files back into the model. Verified action block descriptions are in the text files kept in the `/doc` subdirectory of the directory storing the local encyclopedia. The name of the file has to match the name of the action block. The name of the file is used to locate the action block in the model. The text files have to have `txt` as their file extension. The application has two parameters: the full path to the local encyclopedia and the name of the model kept in the local encyclopedia.

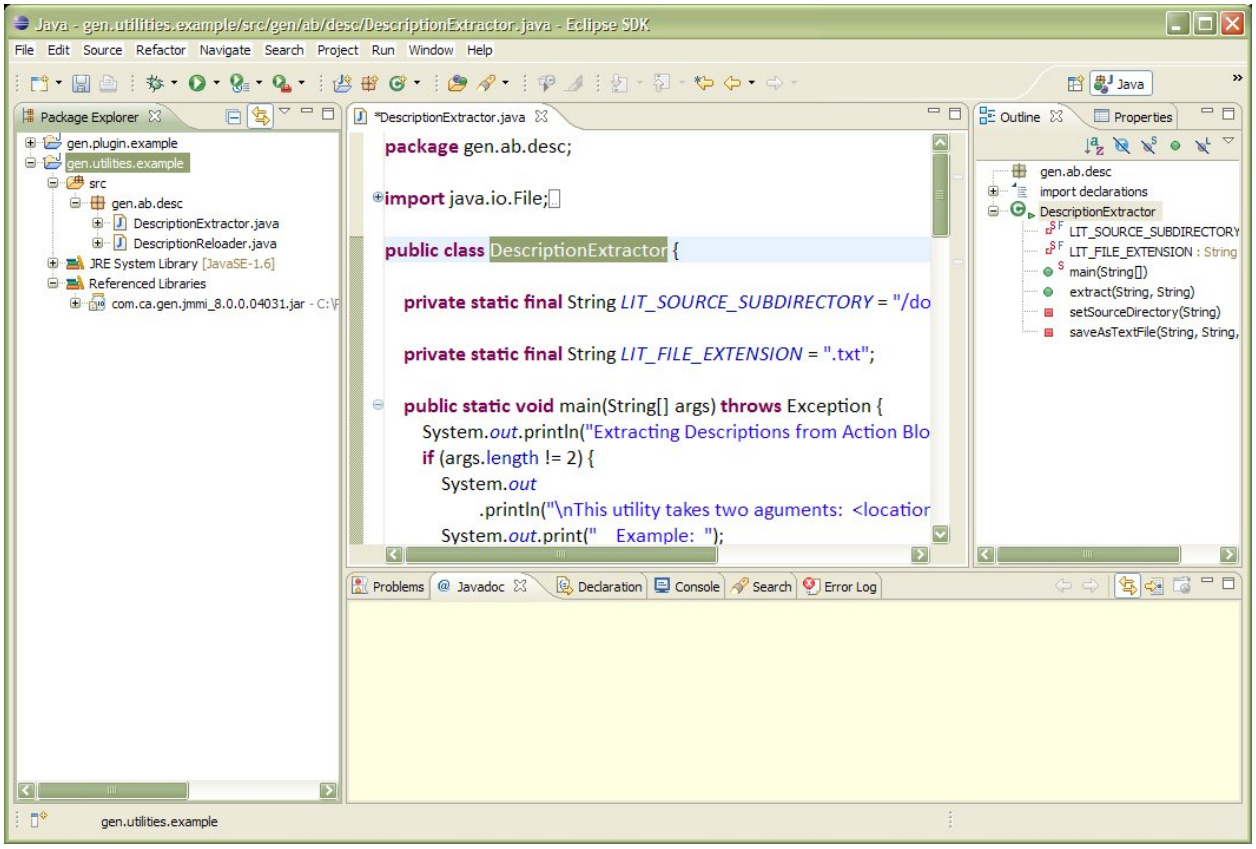
Development environment

You can use any Java development environment to develop and run these sample applications. Any Java development environment you choose has to be located on the windows platform with CA Gen 8.0 installed. The Eclipse SDK has been used to test and run the sample applications. Eclipse Classic 3.4 is used, the same version of Eclipse used in the beta-release of Gen Studio 8.0.

It is highly recommended that you use a version of any third-party software that has been certified for the current release of CA Gen.

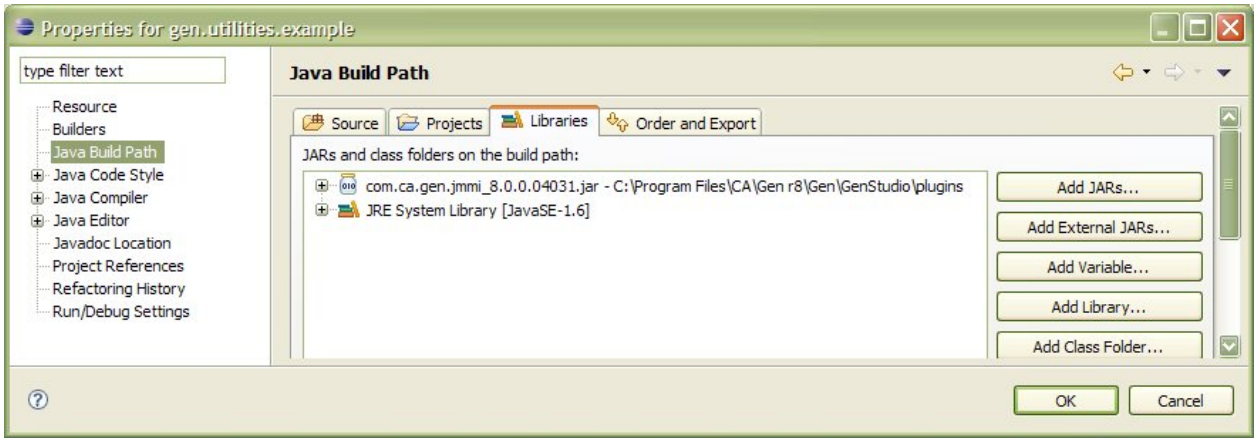
The first step is always to create a Java Project within the Eclipse workspace. The screen below shows Eclipse with an open Java project having two classes used in this example.

Picture1.



You need to set the Java Build Path of the project so your project sees classes of the Gen JMMI API. The following screen shows how to do it.

Picture2.



As you can see the *jar* file `com.ca.gen.jmmi_8.0.0.04031.jar` is added to the Java Build Path using the Libraries tab. The jar file with JMMI API can be found in the following directory:

```
C:\Program Files\CA\Gen r8\Gen\GenStudio\plugins
```

This directory is created during the installation of CA Gen 8.0 on your workstation. You need to be aware that the name of the jar file can be different depending on what Gen build you are currently using. Please always check the current name of the jar file or your Java classes will not compile correctly.

This paper comes with sample Eclipse projects and you may prefer to import complete projects into your workspace. You need to use the Eclipse Import function to do so. You choose the import option *Existing Projects into Workspace.* This Option will prompt you to enter the location of the folder or archive file with the project. The project will be compiled immediately after import is completed. You should not see any compilation errors if your project class path is set correctly.

This completes the preparation steps. The next sections explain the flow of logic for both applications and will explain how the Gen JMMI API works in practice.

Extracting action block descriptions from the model

Class `DescriptionExtractor` is designed to open a specific model in read-only mode, next it looks for action blocks, extracts the description from each action block and places the description in the text files. Each text file is written to the dedicated directory.

Invocation sequence

`DescriptionExtractor` class has a main method which allows execution of the class from the command line. The screen below shows the invocation sequence.

Picture3.

```
public static void main(String[] args) throws Exception {
   System.out.println("Extracting Descriptions from Action Blocks, 1.1");
   if (args.length != 2) {
      System.out
            .println("\nThis utility takes two aguments:  <location of model> and <model name>\n");
      System.out.print("    Example:  ");
      System.out
            .println("  <your model directory>\\desc.ief\" \"descriptions\"");
      return;
   }
   DescriptionExtractor descriptionExtractor = new DescriptionExtractor();
   descriptionExtractor.extract(args[0], args[1]);
}
```

You can see that the application takes two arguments and passes them as parameters to the instantiated class using the `extract()` method. The first parameter points to the local encyclopedia and the second has the name of the model.

Opening and accessing the model

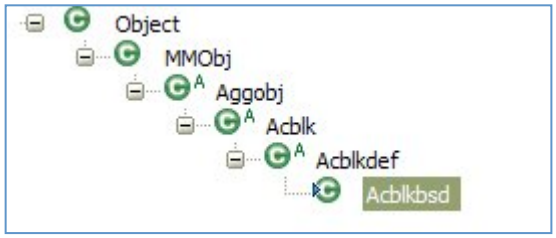
The essential logic accessing the Gen model is executed in the `extract()` method. There are two things you should be aware of before we analyse the logic of the `extract()` method.

Tip 2

All the data object instances within JMMI will at some point in time need to be referred to by a unique identifier. These identifiers are represented as Id objects in JMMI. There are 3 types of identifiers: `EncyId`, `ModelId`, *and* `ObjId`. *In general, ids are not usually instantiated directly. Instead, they are instantiated automatically when the associated data object is instantiated. Ids are then retrieved from the data object itself.*

Tip 3

Let us have a brief look at the type hierarchy of the class which represents an action block in the model. As you can see below Class `Acblkbsd` *extends* `Acblkdef` *which extends* `Acblk` *which extends* `Aggobj` *and which extends* `MMObj`. `Acblkbsd` *inherits all methods of classes above and adds some of its own methods as well. You can easily see that the type hierarchy matches what you can see in the Object Decomposition Report. However, there is a one* `MMObj` *class which does not have an equivalent in the Object Decomposition Report.* `MMObj` *is a base class for an object within a model. It has derived from it a set of classes generated based on the Gen meta-model schema and* `Acblkbsd` *is just one of them.*



You can develop your own logic using `MMObj` *or* `Acblkbsd`. *Both have a set of methods allowing you to access the properties of the object and possibly follow associations from your object to other objects in the model. The following sample code shows how you can extract the property of type 'NAME' from the object of type* `Acblkbsd` *using two different methods.*

```
// method 1
String abname = acblkbsd.getName();
// method 2
abname = acblkbsd.getTextProperty(PrpTypeCode.NAME);
```

Tip 4

As you can see the properties above can be fetched in two different ways. Also MMObj objects can be instantiated in two different ways. In the `extract()` *method, you use*

```
Acblkbsd acblkbsd = (Acblkbsd)MMObj.getInstance(model, objId)
```

That is the generic way. The specific way would be:

```
Acblkbsd acblkbsd = Acblkbsd.getInstance(model, objId)
```

The difference is the lack of a cast. The cast is actually done by the `Acblkbsd.getInstance` *method for you.*

Let us return to the `extract()` method. The following screen shows how very few statements are required to do this activity.

```
public void extract(String path, String name) throws IOException {

  try {
     Ency ency = EncyManager.connectLocalForReadOnly(path);
     Model model = ModelManager.open(ency, ency.getModelIdByName(name
           .toUpperCase()));
     setSourceDirectory(path);
     List<ObjId> list = model.getObjIds(ObjTypeCode.ACBLKBSD);
     for (ObjId objId : list) {
        Acblkbsd acblkbsd = (Acblkbsd) MMObj.getInstance(model, objId);
        saveAsTextFile(path + LIT_SOURCE_SUBDIRECTORY, acblkbsd.getName(), acblkbsd.getDesc());
     }
     System.out.println("Task completed.");
  } catch (EncyException e) {
     System.out.println(" Problem connecting to the encyclopedia.");
  } catch (ModelNotFoundException e) {
     System.out.println(" Model not found.");
  }
```

So how is it done?

1. `EncyManager` is a static factory class that manages the `Ency` objects. Its `connectLocalForReadOnly()` method has one parameter and returns `Ency` object. This parameter points to your local encyclopedia. The method can produce an exception `EncyException` if the location is incorrect. In practice, parameter `path` has to point to the directory having four uncorrupted *dat* files. `Ency` object has a number of methods and we use one of them to find model matching name passed by the parameter `name`.
2. `ModelManager` is a static class that manages the `Model` objects. This class has a static method `open()` opening the model and returning an object representing the opened model. It takes `Ency` object as its first parameter and a `ModelId` as its second. As you can see to get the second parameter, method `getModelIdByName()` of `Ency` object is called. Method `open()` can throw `ModelNotFoundException` exception if the model is not found.
3. The fragment of the code above uses `ency.getModelIdByName()` method. This is probably the best approach if you want to be sure that the local encyclopedia has a model exactly matching the specified name. Instead, you can assume that a local encyclopedia's `modelId` will always be 0, so it gets the `modelId` by calling `ModelId.getInstance(0)`.
4. Method `setSourceDirectory()` will create a subdirectory in the model directory to keep extracted action block descriptions if the subdirectory does not exist.
5. The next step will be to find objects of specific type within the model. Method `getObjIds()` will return a list of object ids for the objects of type ACBLKBSD. As we know from the Object

Decomposition Report the model can also store objects of type ACBLKBAA; both types can have action block descriptions. This sample application ignores type ACBLKBAA for simplicity, but readers are free to modify the logic including both types of action blocks.

6. We can step through the list of objects, extracting instances of the object for each object id on the list. Static method `getInstance()` of the `MMObj` class returns an object. We can correctly assume that the object has to be of type `Acblkbsd`. You need to be sure that the list only contains objects of the same type otherwise you may get incorrect cast exceptions.
7. Method `saveAsTextFile()` is responsible for creating a text file with the extracted action block description. This method has three parameters: the location where text file should be created, the name of the action block and the description itself. You can notice that two specific get methods are used to extract two property values from the action block object. Properties are type of NAME and type of DESC.

This concludes the main flow of logic. As result a number of text files are created in the designated directory each with a name derived from the action block name containing the text of the descriptions.

Saving descriptions

`DescriptionExtractor` class has two extra methods: creating the directory for the text files and creating the text files. Implementation code for both methods follows:

```
private void setSourceDirectory(String path) {
   File directory = new File(path + LIT_SOURCE_SUBDIRECTORY);
   if (!(directory.exists() && directory.isDirectory())) {
      if (directory.exists() && directory.isFile()) {
         directory.delete();
      }
      directory.mkdir();
   }
}
```

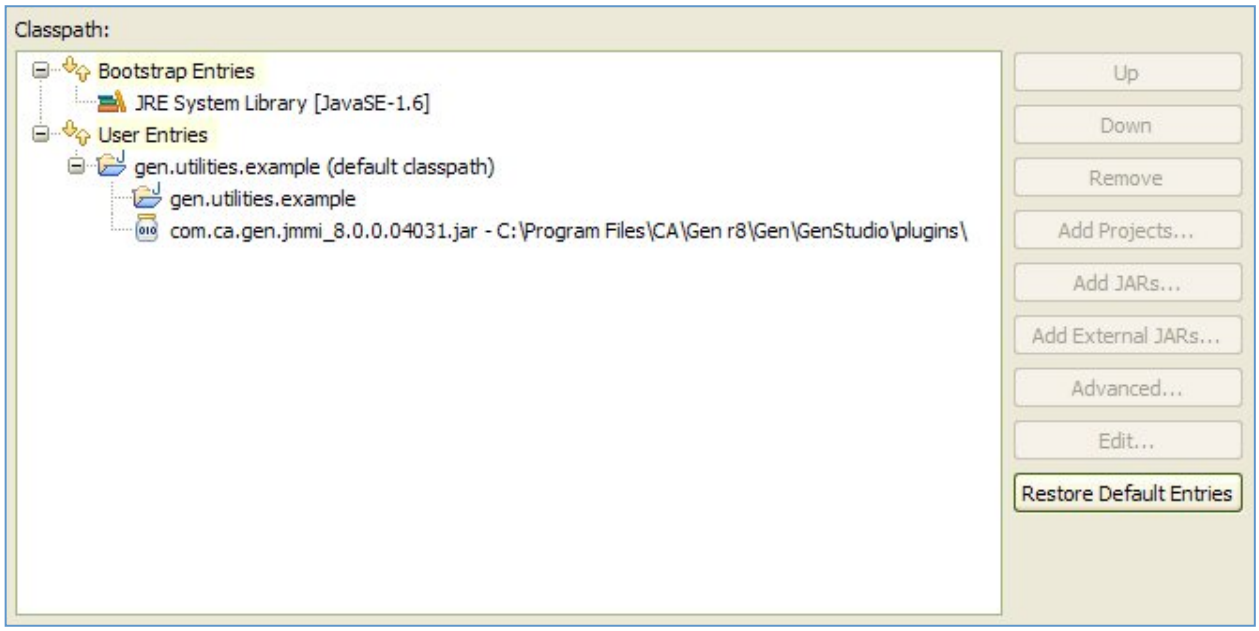
```
private void saveAsTextFile(String sources, String abName, String description)
      throws IOException {
   String name = sources + File.separator + abName + LIT_FILE_EXTENSION;
   File file = new File(name);
   if (file.exists()) {
      file.delete();
   }
   file.createNewFile();
   PrintWriter printWriter = new PrintWriter(file);
   printWriter.println(description);
   printWriter.close();
   System.out.println("a new file " + name + " has been created.");
}
```

Full source of the class is part of the attached Java Eclipse project.

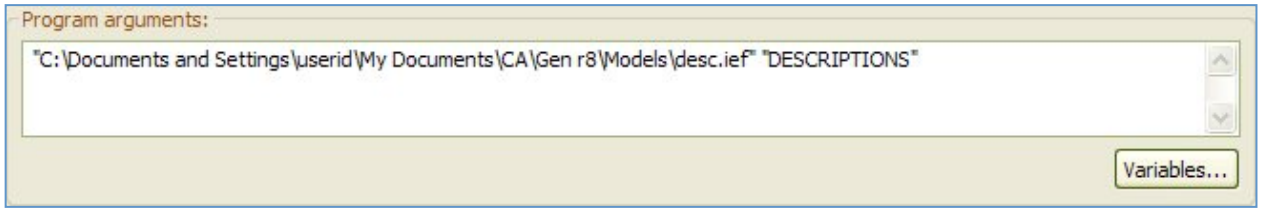
Executing the application

The application can be executed from inside Eclipse or as a standalone application executed from the command line. The screens below show how to run the application from Eclipse.

SETTING THE CLASSPATH



PASSING TWO ARGUMENTS



RESULT

```
Extracting Descriptions from Action Blocks, 1.1
a new file C:\Documents and Settings\stama09\My Documents\CA\Gen r8\Models\desc.ief/doc\ACTION_BLOCK_1.txt has been created.
a new file C:\Documents and Settings\stama09\My Documents\CA\Gen r8\Models\desc.ief/doc\ACTION_BLOCK_2.txt has been created.
Task completed.
```

You need to use the Export feature of Eclipse to create a jar file if you want your utility to be executed outside the Eclipse SDK. Assuming that Extractor.jar is a name given to the exported jar file, the application can be executed from the command line as follows:

```
C:\aa>java -jar Extractor.jar "C:\Documents and Settings\stama09\My Documents\CA
\Gen r8\Models\desc.ief" "DESCRIPTIONS"
Extracting Descriptions from Action Blocks, 1.1
a new file C:\Documents and Settings\stama09\My Documents\CA\Gen r8\Models\desc.
ief/doc\ACTION_BLOCK_1.txt has been created.
a new file C:\Documents and Settings\stama09\My Documents\CA\Gen r8\Models\desc.
ief/doc\ACTION_BLOCK_2.txt has been created.
Task completed.
```

Reloading descriptions to the model

Reloading descriptions from the text files to the model has equally simple logic. Class `DescriptionReloader` is designed to open a specific model for update, search for text files at the specified location, find action blocks in the model whose name matches the name of the text file and update the description property of each identified action block in the model. Once all action blocks have been updated, changes to the model are committed and the model is closed.

Invocation sequence

`DescriptionReloader` has a `main()` method which allows execution of the class from the command line. The following source code shows the invocation sequence.

```
public static void main(String[] args) throws Exception {
   System.out.println("Reloading Descriptions to Action Blocks, 1.1");
   if (args.length != 2) {
      System.out
            .println("\nThis utility takes two aguments:  <location of model> and <model name>\n");
      System.out.print("    Example:  ");
      System.out
            .println("  \"C:\\Users\\User\\Documents\\CA\\Gen r8\\Models\\desc.ief\" \"descriptions\"");
      return;
   }
   DescriptionReloader descriptionReloader = new DescriptionReloader();
   descriptionReloader.reload(args[0], args[1]);
}
```

The application takes two parameters and passes them to the instantiated class using the `reload()` method. It assumes that the first argument points to the local encyclopedia and the second argument has the name of the model.

Opening and updating the model

We are going to explain the following piece of logic.

```
public void reload(String path, String name) throws IOException {
   Ency ency;
   Model model = null;
   try {
      ency = EncyManager.connectLocal(path);
      ModelId modelId = ency.getModelIdByName(name.toUpperCase());
      model = ModelManager.open(ency, modelId);
      model.beginUnitOfWork();
      if (isSourceDirectory(path)) {
         File directory = new File(path + LIT_SOURCE_SUBDIRECTORY);
         File[] files = directory.listFiles();
         for (File file : files) {
            if (file.isDirectory()
                  || !file.getName().endsWith(LIT_FILE_EXTENSION)) {
               continue;
            }
            Acblkbsd acblkbsd = findActionBlock(model, file.getName());
            if (acblkbsd == null) {
               continue;
            }
            updateActionBlock(acblkbsd, file);
            System.out.println(". Completed");
         }
         model.commitUnitOfWork();
         model.save();
         System.out.println("Task completed.");
         return;
      }
      System.out.println("Cannot find source directory with descrptions");
   } catch (EncyException e) {
      System.out.println(" Problem connecting to the encyclopedia.");
      if (model != null) {
         model.rollbackUnitOfWork();
      }
   } catch (ModelNotFoundException e) {
      System.out.println(" Model not found.");
      return;
   }
   model.close();
```

1. Static factory class `EncyManager` has a method `connectLocal()` which returns `Ency` object. We use this method each time we update models stored in the encyclopedia. This method can throw exception `EncyException` if the specified path is incorrect.
2. Next step is to find the model in the encyclopedia using the model name. The method `getModelIdByName()` returns `ModelId`. This method can throw exception `ModelNotFoundException` if the model cannot be found.
3. We are using static factory class `ModelManger` and its method `open()` to open the model. The method has two parameters: `Ency` object and `ModelId` object. This method can throw a number of exceptions such as the model is not available or is already opened or there are some communication problems.

4. The model is opened with for update so we need to begin unit of work and commit unit of work after all action blocks are successfully updated. On some rare occasions it may be necessary to rollback unit of work. Model should be saved and closed after successful completion. Class model has a set of methods to maintain the integrity of the model. They are `beginUnitOfWork()`, `commitUnitOfWork()` and `rollbackUnitOfWork()`. Methods `save()` and `close()` match functions which you know from using the Gen Toolset. They both ensure that changes to the model are saved and the model closed properly.
5. From a closer look at the logic flow you can see that the application searches for text files in the specified directory with the name of the file used to find matching action blocks in the model. A purpose written method `findActionBlock()` takes two parameters and returns a non-null object for each action block which matches the specified name.

```
private Acblkbsd findActionBlock(Model model, String name)
    throws EncyUnsupportedOperationException {
  String abName = name.substring(0, name.lastIndexOf(LIT_FILE_EXTENSION,
      name.length()));
  System.out.print("Processing " + abName);
  if (list == null) {
    list = model.getObjIds(ObjTypeCode.ACBLKBSD);
  }
  for (ObjId objId : list) {
    Acblkbsd acblkbsd = (Acblkbsd) MMObj.getInstance(model, objId);
    if (acblkbsd.getName().equals(abName)) {
      return acblkbsd;
    }
  }
  System.out.println(" Action block not found.");
  return null;
}
```

6. Once a matching action block is found method `updateActionBlock()` is executed. This method takes two parameters, the first is an object representing the action block and the second a handle to a text file with the description.

```
private void updateActionBlock(Acblkbsd acblkbsd, File file)
    throws IOException {
  acblkbsd.setDesc(retrieveDescription(file));
}
```

7. The update method is very simple. Class `Acblkbsd` has one convenient method for each property. Method `setDesc()` takes a string of characters and overwrites the existing value of the property of type 'DESC'. A new action block description is returned by purpose written method `retrieveDescription()`. This method has a single parameter which is the file handle to the text file.

```
private String retrieveDescription(File file) throws IOException {
   BufferedReader bufferedReader = new BufferedReader(new FileReader(file));
   StringBuffer buffer = new StringBuffer();
   String line = bufferedReader.readLine();
   while (line != null) {
      buffer.append(line + '\n');
      line = bufferedReader.readLine();
   }
   String result = null;
   if (buffer.length() > 3950) {
      result = buffer.substring(0, 3950);
      System.out.print(". Truncated to size of 3950 characters.");
   } else {
      result = buffer.toString();
   }
   return result;
}
```

8. You should be aware that the description property can have a limited number of characters and any description longer than 3950 characters will be truncated.
9. Lastly you need to be aware of NLS and codepage issues. The set property methods are going to convert from the Unicode Java strings to the model's code page. Care needs to be taken by the user to ensure that appropriate data is passed in. Along with that your calls to open `BufferedReaders` can also take a file encoding as a `parm`. Without the `parm` the machine's default file encoding will be used.

The job of reloading descriptions from the text files into action blocks is completed when the model is saved and closed.

Running the application

The preparation to run the application is no different from what we did for the Description Extractor class and there is no need to repeat it here. It will produce the following output:

```
Reloading Descriptions to Action Blocks, 1.1
Processing ACTION_BLOCK_1. Completed
Processing ACTION_BLOCK_2. Completed
Task completed.
```

Summary

These API classes allow CA Gen customers and Open Initiative partners to write software to retrieve and update encyclopedia information. Sample applications used in this knowledge document do not show

the full potential of this new interface, but it should convince you that writing such application need not be too difficult.