Scribble

Developer Guide

by Kohei Honda and Gary Brown

Chapter 1.

Overview

Project Structure

2.1. Project Management

This section outlines the technology used to manage the different aspects of the project.

2.1.1. Source Code Management (SCM) using GIT

GIT is a relatively new source code management system.

The distinguishing feature of GIT, over other more common SCM systems such as subversion and cvs, is that it works as a network of repositories.

Most, if not all other, SCM technologies use a central repository that all users check code in and out of. GIT makes a local repository on the users box that is then synchronized with the network based repository.

When someone wishes to enhance the project in some way, they can opt to collaborate on that network based repository, or fork the repository and work independently of the main repository. At a suitable point in the future, they can then request that their changes are 'pulled' back into the main project, or simply remain as a separate project.

For the purposes of the Scribble project, this way of working seems ideal. Different groups, whether industry or academic, can take their own forks of the project and make use of them in their own specific ways.

When those groups produce something that they feel is generally useful for the Scribble community, they can request that the main Scribble project 'pulls' the relevant changes to incorporate the new functionality.

However, it also means that those separate groups can build their own specific features that can remain localised to their version of the project.

Although these forked projects would be separate from the main project, they can be configured to be notified when changes are made to the main project, so that they can opt to merge in the new changes. This way, it enables all of the satellite projects to remain up-to-date with changes in the new project.

GIT has been specifically designed to make forking and merging repositories easy.

The project is currently hosted at GIT Hub which provide useful tools around the GIT repository.

2.1.2. Issue Management

Currently issue management is handled by the Scribble project hosted on github.

However each separate (forked) project may maintain its own set of issues, relate to its own functionality. Although hopefully issues that effect the functionality in the core (shared) project should be reported to the main GIT project.

2.1.3. Project Build Management

The project build is performed using maven.

The maven plugins, used to build the OSGi bundles, are fairly common. Although specific maven plugins do exist for building OSGi bundles, they don't currently work well.

However, basically all that is required is the ability to package the relevant classes in a jar with a manifest that defines the relevant OSGi properties. Therefore this is what we use.

The other generally used maven plugins relate to compilation of the Java classes, execution and analysis of JUnit tests.

2.2. Distribution

The distribution mechanism is aimed at providing a zipped archive that contains the necessary environment.

This environment provides the ability to execute Scribble commands from the command line. However it is also possible to use the jars, contained in the `lib` and `bundle` sub-folders, directly within other Java applications.

To automatically include a new bundle in the distribution (`bundle` subfolder), it should be defined in the *org.scribble.modules* maven group.

Note

In the future, it will also be possible to use the OSGi bundles within an Eclipse environment. A separate update site will be made available, including the relevant bundles, and Eclipse specific plugins.

2.3. Integration

The `integration` branch of the project is concerned with providing integration of the OSGi bundles, defined in the `modules` branch, in different execution environments.

2.3.1. Command Line

The Scribble tools architecture is based on OSGi, which means that the OSGi compliant bundles can run within any OSGi compliant service container. However OSGi is a service framework, intended to manage services in a service container (or server).

Therefore, to leverage OSGi bundles (or services), from a command line invoked application, we need to select a specific OSGi implementation that supports this approach, as it is not defined as part of the OSGi standard.

Therefore, to provide this command line capability, we have selected the Apache Felix OSGi implementation. This is the reason that the *Felix* jars are included in the distribution's `lib` sub-folder, rather than just implementation independent OSGi jars.

Although it is possible to define new modules as part of the Scribble project, it is also possible to develop them independently and just place them within the `bundle` folder of the installed (unpacked) Scribble distribution. This will make them available as part of the command line commands (e.g. if the bundle represents an additional validation module).

2.3.2. Embedded Java

OSGi is about defining components, with well defined interfaces, and managing their isolation, to enable modules to be dyamically added or removed as necessary.

However, it is also possible to use these same components, based on the separate of interfaces and implementations, using any suitable factory or direct injection approach.

The bundles are just normal *jar* archives. They only have special significants when placed in an OSGi container.

2.3.3. Eclipse

To integrate the Scribble protocol model, parser and supporting validating modules into Eclipse, it is necessary to package them in the form of an update site.

The `integration/eclipse/updatesite` folder contains the Eclipse artifacts used to build the update site (specifically the `site.xml` file). When this file is opened within an Eclipse environment, that contains the Plugin Development Environment (PDE), it will open the file in a special editor. The update site can be built by selecting the *Build All* button.

Once the plugins and features have been built, then the `updatesite` folder contents should be zipped up, and distributed as necessary. This can be downloaded for use as a local update site, or exploded into a suitable location on the network, to be made available as a remote update site.

The details of which features, and therefore plugins, are included in the update site, are defined in the `site.xml` file.

As well as incorporating the OSGi bundles defined in the `modules` sub-folder, the Eclipse integration has some additional Eclipse specific plugins.

For example, the Scribble Designer is included as an additional plugin within the Eclipse integration branch of the project structure.

Note

Currently the project is setup to only checkin the update site `.project` and `site.xml` files. The built `features` and `plugins` folders, as well as the generated jar files, will not be checked into the project.

However when the update site is built, it modifies the `site.xml` to reference the specific features (and versions) that it incorporates. Therefore, before any files are subsequently checked in, it will be necessary to remove the features from the `site.xml`, and re-add them, to reset them to the generic version.

In the future, a more automated mechanism will be used to build the update sites. However, until then, it will be necessary to use this manual approach.

2.4. Modules

The `modules` sub-folder contains all of the OSGi compliant bundles that can be used in any of the integration environments.

Note

TO DO: List the main modules and how they are related.

2.5. QA

There are two types of QA that are performed as part of the project:

1. Local test cases
   Unit tests would be used to test the individual classes within the specific implementation of an interface.

2. Integration tests
   Where multiple implementations of a particular module could exist, an integration test strategy may be useful to ensure that all implementations of the same interface behaviour in the same way.

This section will discuss the second type of QA, aimed at ensuring multiple implementations behaviour in the same way.

2.5.1. Protocol Parser - Conformance Test Kit (CTK)

This part of the project structure provides a set of tests to check that the parser (being tested) processes the supplied set of test 'protocol' descriptions, and returns the correct object model.

The test protocol descriptions are stored in the `src/test/resources/tests` folder.

Each test is accompanied by a junit test, defined in the class *org.scribble.protocol.parser.ctk*. For example:

```
@org.junit.Test
public void testSingleInteraction() {
 TestScribbleLogger logger=new TestScribbleLogger();

 Model<Protocol> model=getModel("SingleInteraction", logger);

 assertNotNull(model);
```

```
  assertTrue(logger.getErrorCount() == 0);

  // Build expected model
  Model<Protocol> expected=new Model<Protocol>();

  Namespace ns=new Namespace();
  ns.setName("example.helloworld");
  expected.setNamespace(ns);

  Protocol protocol=new Protocol();
  expected.setDefinition(protocol);

  LocatedName ln=new LocatedName();
  ln.setName("SingleInteraction");
  protocol.setLocatedName(ln);

  ParticipantList rl=new ParticipantList();
  Participant buyer=new Participant();
  buyer.setName("buyer");
  rl.getParticipants().add(buyer);
  Participant seller=new Participant();
  seller.setName("seller");
  rl.getParticipants().add(seller);

  protocol.getBlock().getContents().add(rl);

  Interaction interaction=new Interaction();

  MessageSignature ms=new MessageSignature();
  TypeReference tref=new TypeReference();
  tref.setLocalpart("Order");
  ms.getTypes().add(tref);
  interaction.setMessageSignature(ms);
  interaction.setFromRole(buyer);
  interaction.setToParticipant(seller);

  protocol.getBlock().getContents().add(interaction);

  verify(model, expected);
 }
```

The *getModel()* method retrieves the protocol description from a named file, and invokes the parser implementation being tested.

The parser implementation is defined using the *scribble.protocol.parser* system property. If this property is not set, then it will default to the ANTLR based implementation.

Once the model has been retrieved using the parser, the unit test will construct an 'expected' object model.

The final step in the unit test is to invoke the *verify()* method to compare the model retrieved against the 'expected' version.

To perform the verification, each model is flattened to produce a list of 'model objects'. Then the verification mechanism iterates through the list, checking that the same entry in each list is identical - first checking they are the same class, and then invoking a 'comparator' implementation for that class.

The 'comparator' implementations are defined in the *org.scribble.protocol.parser.ctk.comparators* package. The comparator implementations are registered in the static initializer for the *org.scribble.protocol.parser.ctk.ProtocolParserTest* class.

2.6. Samples

The `samples` sub-folder will provide samples that reflect different aspects of the Scribble notation.

Architecture

3.1. OSGi Infrastructure

The Scribble architecture is based on OSGi, to provide a means of managing the individual modules, but without causing tight coupling.

Service bundles enable implementations to be specified that implement defined interfaces. Other services can then request access to services that implement a particular interface.

The OSGi service container takes responsibility of managing the services, and providing access to requesting components.

This provides flexibility for Scribble tooling in two respects:

1. Replaceable
   The implementation of a particular Scribble interface can easily be replaced. This enables different research or industry groups to replace specific modules, with alternative implementations, while still reusing other modules.

2. Extensibility
   Some aspects of the architecture allow for multiple implementations of the same interface. Therefore, using OSGi, enables additional implementations of the same interface to be easily plugged in, without having to define any additional configuration information.

3.2. Core Components

3.2.1. Protocol Notation

3.2.2. Protocol Parser

3.2.3. Protocol Model

3.2.4. Error Logging

There is a generic logging API within the Scribble framework that can be used for reporting errors, warnings, information or debuging details. This API is *org.scribble.common.logging.ScribbleLogger*.

The methods generally take two parameters, a message and a property map. The message is simply a description of the issue being reported. The property map contain specific details about the issue being reported.

For example, when the parser detects a problem, it can report the nature of the problem, and provide the location of the issue in the source file.

3.2.4.1. Internationalization

To enable errors reported from the Scribble parser and validation modules, in a number of different languages, internationalization should be used.

The following code fragment provides an example of how internationalization can be achieved, using parameterised messages.

```
logger.error(org.scribble.common.util.MessageUtil.format(
    java.util.PropertyResourceBundle.getBundle(
      "org.scribble.common.validation.rules.Messages"),
          "_EXISTING_DECLARATION",
          new String[]{elem.getName()}),
          obj.getProperties());
```

The main message content is obtained from a properties file, with the name being supplied as the first parameter to the *getBundle* method. The property file must be placed the correct package within the `src/main/resources` folder, to ensure the properties are correctly packaged by maven.

The messages within the property files can have values that include parameters. Parameters are numbered in sequential order, and defined between curly braces (e.g. {n} where 'n' is the number). For example,

```
_EXISTING_DECLARATION=Declaration already exists for name '{0}'
```

This message only has a single parameters.

In the previous code fragment, the *MessageUtil.format()* method takes the message as the first parameter, and an array of strings as the parameter values to be substituted in the message. So in the code fragment, the value in the *elem.getName()* would be substituted in the *{0}* parameter of the *_EXISTING_DECLARATION* message, and then reported to the Scribble logger.

3.2.5. Type Resolution

Note

Placeholder for component to resolve references from one model to other models and types. This will be required where one protocol references an external protocol - to enable other validation to span across the multiple protocols seemlessly.

3.2.6. Protocol Validation Manager and Validators

The validation manager, when used in a OSGi runtime environment, will listen for the activation of any implementations of the `org.scribble.protocol.validation.Validator` interface.

This means that the validation of any model can be performed using the *org.scribble.protocol.validation.ValidationManager*, rather than having to obtain instances of multiple implementations of the `Validator` interface.

When the `ValidatorManager` is used outside of an OSGi environment, it is necessary for the validators to be added to the manager by other means.

3.3. Command Line Actions

The first step is to define the command implementation of the *org.scribble.command.Command* interface. This can be created in the *org.scribble.command* Eclipse plugin.

To the initialise the command, as part of an OSGi runtime environment, the command implementation can be instantiated in the *org.scribble.command* plugin's *Activator*, and then registered with the bundle context.

If the command requires other OSGi services, then these can be established by setting up service listeners for the relevant service interface classes. When OSGi services are registered, then the relationship can be established.

This command mechanism will generally only be used as part of the command line approach, and therefore does not need to be initialised in other ways. However other dependency injection techniques could be used if appropriate.

The only remaining step is to create the scripts necessary to enable a user to invoke the command. This can be achieved by copying one of the existing scripts, in the `distribution/src/main/release` folder (such as `parse.sh`), and modify the parameter details as necessary.

Note

The first parameter of the Java application, invoked within the script, must be the name of the command. This must match the value returned by the *getName()* method on the command interface.

Developing a Validator

This section will describe how to create a validator as part of the Scribble project structure. The same approach can also be used to create a validator module outside the scope of the Scribble project.

To explain how to create a validator, we will use the 'connectedness' validator as an example.

4.1. Create the Validator OSGi bundle

This section will explain how to create the OSGi bundle, for the validator, from within the Eclipse environment.

The first step is to create the plugin project, using the *New->Project->Plugin Project* menu item. When the dialog window is displayed, uncheck the "Use default location" checkbox, and browse to find the appropriate location for the new project.

For this project, the location will be the `modules/org.scribble.validator.connectedness` folder within the Scribble project structure. It will be necessary to create the folder for the *org.scribble.validator.connectednesss* part of the location - the folder being named after the OSGi bundle identity.

Ensure the 'Create java project' checkbox is ticked, and then set the source folder to be `src/main/java` and set the *Target Platform* to a standard 'OSGi Framework'.

Then press the *Next* button to set some details associated with the plugin, such as the version, description, provider, etc.

In this example, we will be registering the validator using the OSGi *registerService* method. This is performed in the bundle activator, whose class is set in the plugin details. For example, in the start method of the created Activator, we would have:

```
public void start(BundleContext context) throws Exception {
       Properties props = new Properties();

       Validator validator=new ConnectednessValidator();

       context.registerService(Validator.class.getName(),
   validator, props);
}
```

Note

Need to investigate whether it is also possible to use the OSGi Declarative Services approach. Issue may be location of the component.xml file, so that both Eclipse and the maven archive plugin picks them up.

4.2. Establish Bundle Dependencies

Depending on the type of bundle being developed, it may have a different set of dependencies than the ones required by this 'connectedness' validator. However the configuration approach will be the same.

Go to the `META-INF/MANIFEST.MF` file and select it. This will cause the plugin manifest editor to be displayed.

Select the *Dependencies* tab and select the other bundles that will be required, or preferrably select the packages to be imported (as this avoids dependency on specific bundles, and instead just identifies the packages required). For this example validator, we just need to add the packages from the *org.scribble.common* bundle which is used by all Scribble plugins. However if additional packages were required, then they could be added as imported packages.

4.3. Implement the Module

Each modules will be different, and therefore discussing specific implementation details will not be possible.

However validation modules will tend to access the complete model, but possibly only be interested in certain parts of it. Therefore usually the validation modules will define an implementation of the `org.scribble.protocol.model.Visitor` interface.

The actual main class within the validator module would implement the `org.scribble.protocol.validation.Validator` interface.

4.4. Implement the Tests

Tests can be implemented in two ways, depending upon the nature of the bundle.

If the bundle is representing an implementation of a common interface, where the result returned from the bundle is the key point, then integration tests associated with the interface can be useful.

For example, there is a special bundle used to provide a *Conformance Test Kit* for the protocol parser. This means that the same set of integration tests can be used regardless of the implementation of the parser that is used.

The other set of tests that are useful are specific to the bundle implementation. These tests will usually be provided within the bundle itself, by defining the JUnit test classes within the `src/test/java` location, with any required resources being placed in `src/test/resources`. Both of these locations also need to be added to the Eclipse project's classpath.

The next step is to create the JUnit test. First create the appropriate package within the `src/test/java` location.

Then select the *New->Other->JUnit Test Case* menu item associated with the package. When this is first performed, you will be asked which version of JUnit should be used. Select the *New JUnit 4* radio button.

After pressing the *Next* button, you will be asked about adding JUnit to the classpath.

Choose the *Open the build path property page* option and press the *Ok* button. The reason for not adding JUnit directly, is that this would cause it to be included in the list of bundle dependencies in the OSGi manifest, which would mean that the runtime environment that includes the validator would also have a dependency on JUnit.

Therefore we need to add the JUnit jars to the Eclipse project in a different way. In the build path dialog, select the *Libraries* tab and then select the *Add Library* button. When the list of libraries is presented, select the *JUnit* entry and press *Next* where the JUnit version should be set to *JUnit4* and then press the *Finish* button.

The final step is to associate JUnit with the Maven `pom.xml` file associated with the bundle. This is achieved by adding the following snippet to the dependencies section:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <scope>test</scope>
</dependency>
```

The version of the JUnit bundle is defined in the top level `pom.xml` for the Scribble project.

4.5. Create the Maven POM

The best approach is to copy the `pom.xml` file from one of the other modules, and simply update the relevant sections (e.g. artifact id, description and dependencies).

Once the `pom.xml` for the module has been defined, it needs to be linked into the `pom.xml` (in the modules section) associated with the parent `modules` folder.