Extending a Generator in the Eclipse IDE for EGL Developers

Draft: 3 November 2012

Copyright © IBM Corporation and others 2012. For full details, see the "Legal" topic at http://www.eclipse.org. Also see the Trademarks section on page 30 of this document.
Introduction

This document introduces the process of extending a generator in EGL Development Tools (EDT). Also included here are reference details so you can adapt the process to your own use.

The plan is to supplement this document with a step-by-step tutorial.

Reasons for extending a generator

You extend an EDT generator to provide a new capability for business developers who write code in EGL. You might want to customize the generated output for a target system that EDT already supports. Alternatively, you might want to create output for a new target system, possibly for a language not currently supported.

For example, you might write Java™ classes that have one of these effects:

- Provides a new EGL type.
- Generates programs that automatically issue runtime calls to a performance monitor.
- Customizes the code that EDT generates for assignment, comparison, or other language construct.
- Structures generated Java code for a mobile-phone application that relies on Android™ technology.
- Structures generated C# code for a web application that relies on the Microsoft™ .NET Framework.

This tutorial shows how to extend the EDT Java generator, but the instructions are useful for extending the JavaScript generator and for supporting a new output language.

Extension from the EGL developer's point of view

To extend a generator, you create Java classes that supplement existing logic. At generation time, one of those classes acts as a front end. This tutorial refers to the front-end class as the command processor because the class processes the arguments that are passed during invocation of the generator.

A command-line invocation of your extension starts with the name of the command processor. However, the name is not used in the EDT Integrated Development Environment (IDE), which hides the complexity. From the point of view of most EGL developers, a new extension is a new generator.
The two-pass generation process

An EGL generator conducts two passes for each EGL type that is an input to the generator:

1. **Pregeneration pass**

   The generator scans the EGL type to determine whether a generation is possible and valid. For example, information is collected about fields in the type. The generator stores the information for later use.

2. **Generation pass**

   The generator accesses the stored detail and writes output to a target-language source file, often in small increments such as “=5”.

Generator properties files

To extend an EDT generator, you might define text-based properties files of the following kinds:

- **templates.properties**
  
  A file of this kind references template classes, each of which does some processing at generation time; for example, to generate output for a specific language construct such as “assignment.”

- **nativeTypes.properties**
  
  A file of this type defines the set of native EGL types, which are types that cannot be handled as primitives of the language being generated.

- **primitiveTypes.properties**
  
  A file of this type defines the set of EGL types that are handled by the generated language. In this tutorial, the language is Java.

- **EGLMessages.properties**
  
  A file of this type defines the structure of each message that can be issued by the generator.

At generation time, the content of your properties files is merged with the content of similar files that are already in use, and your files are processed first. In this way, your template classes (for example) take precedence.

Never update the property files that are provided with an existing generator. The content of
the properties files that are delivered with EDT will likely change over time. By keeping your work separate, you shield it from at least some of those future changes.

The next sections review each type of properties file in turn.

templates.properties file
You are likely to create you own templates.properties file, whether to create new kinds of generated output or to override existing behavior. The file contains entries like the following one (but on a single line), with the name of a MOF model interface on the left and the name of a template class on the right:

```
org.eclipse.edt.mof.egl.Part =
org.eclipse.edt.gen.java.templates.PartTemplate
```

As suggested by the names, the word “part” refers to a custom EGL type such as a program named MyProgram.

nativeTypes.properties file
You are likely to create your own nativeTypes.properties file to provide access to generator external types. Those types are based on Java classes that are always available to the generator. In contrast, the parts defined by EGL developers are located by EGL runtime code and are not in a properties file at all.

Native types also include EGL libraries and exception-record types.

The nativeTypes.properties file specifies three names that are needed to identify a type:

- The name of the EGL external type, also known as the name of the EGL model interface. An example is “mycompany.mofModel.MyType”.
- The name of the Java interface; for example, “mygenerator.MyJavaInterface”.
- The name of the Java implementation; for example, “mygenerator.MyJavaImplementation”.

The following rules apply to the file entries:

- If the three names are different from one another, two entries are in the properties file:

  ```
  mycompany.mofModel.MyType=mygenerator.MyJavaInterface
  mygenerator.MyJavaInterface=mygenerator.MyJavaImplementation
  ```

- If the MOF model interface name is the same as the Java interface name, a single entry is sufficient. That entry relates the Java interface to the Java implementation:

  ```
  mygenerator.MyJavaInterface=mygenerator.MyJavaImplementation
  ```

- If the three names are identical, a file entry is not required.
**primitiveTypes.properties file**

You are not likely to create your own primitiveTypes.properties file. The generator already defines EGL types that are implemented by primitive types in the generated language.

The file includes the EGL type name and the related generated-language primitive type. For example, the EGL type named `egl.lang.Int32` is equivalent to a Java int, and the file entry is as follows:

```
egl.lang.Int32=int
```

A primitive type might be implemented by a Java language class. For example, the EGL type named `egl.lang.AnyDecimal` is equivalent to the Java class named `java.math.BigDecimal`. Here is the file entry:

```
egl.lang.AnyDecimal=java.math.BigDecimal
```

Last, a generated-language primitive might be available as a primitive and as a class. The situation applies to the Java integer. For example, the following entries relate an EGL type, a Java primitive, and the related Java class:

```
egl.lang.Int32=int
int=java.lang.Integer
```

Those two entries are necessary to support both aspects of the Java type. The entries can be in either order.

**EGLMessages.properties file**

The EGLMessages.properties file contains entries like the following one, with the message number on the left and the message itself on the right:

```
9998=Exception occurred: {0}
```

As shown, the message can include place-holders for text to be inserted by the generator.

**Context object**

The methods in your template classes reference a Context object. The object provides access both to the data that is specific to a generation and to the logic that is available in other template classes.

**Access of data that is specific to a generation**

Two kinds of generation data are particularly important: attributes and smap data.
**Generation attributes**

You can save data in one method, such as a method invoked during pregeneration, and then use that data in logic that runs later. The capability is provided by the `putAttribute` and `getAttribute` methods of the `Context` object. The methods are inherited from the `HashMap` class.

When storing data, you assign a two-part composite key. The first key is an object such as a field being generated. The second key or “sub-key” is a constant that identifies a value related to the first. For example, the sub-key might identify the field length or content.

If you have details that are globally meaningful to the generation, you can assign the `Context` class itself as the first key. For example, you might use the class to store a list of the libraries that are referenced by the program being generated. Here is the retrieval code:

```java
List<Library> libraries =
    (List<Library>) ctx.getAttribute(ctx.getClass(),
                                Constants.SubKey_partLibrariesUsed);
```

**smap data**

When EGL developers debug their source code, they perceive themselves to be interacting with that code even though the generated code is running. Also, when they review a variable in the Eclipse **Variables** view, they see details on the EGL type rather than a type that is specific to the generated output language.

If the output language is Java, the EGL debugger relies on **smap data** to handle the cross reference between the EGL and generated source. The smap data originates in the `Context` object.

For details on the technology that was used to map each EGL line to one or more lines of Java code, see Java Specification Report (JSR) 000045:

http://jcp.org/aboutJava/communityprocess/final/jsr045/index.html

The mapping of EGL variable types to Java variable types is an EDT innovation.

**Access of logic in other template classes**

The pre-existing EGL generators define an inheritance chain. For example, the `AssignmentStatementTemplate` class extends the `EGLClassTemplate` class, which extends the `PartTemplate` class. However, the inheritance chain is dynamic. It is defined by a look-up mechanism that involves the MOF model and a set of template classes.

The generator and the generator’s extender use the look-up mechanism to search for template classes. This mechanism adds flexibility because you can substitute your own template class anywhere in the inheritance chain without changing any other class. For
example, you might substitute your own logic for the `EGLClassTemplate` class just mentioned.

The `invoke` method of the Context object searches the inheritance chain for a method that has the following characteristics:

- Has a name that matches the first argument in the `invoke` method invocation.
- Has a series of parameters that are compatible in type and position with the subsequent arguments in that invocation.

If the method is not found, the generator throws a TemplateException.

Here are examples of the `invoke` method:

```java
ctx.invoke(preGenField, part, ctx, field);
ctx.invoke(genExpression, stmt.getAssignment(), ctx, out);
```

The `invoke` method takes at least three arguments:

- First is the name of the method to invoke. Two conventions are typically in effect:
  - The method name begins with “gen” or “preGen” to identify the pass for which the method was designed.
  - The argument is a constant that is named for the method name. However, this convention is not always followed; the argument might be a literal.
- Second is an object that implements a MOF model interface. That interface is related to the first template class in the inheritance chain, as further explained in a later section.
- Third is the Context object.

If the `invoke` function is meant for the generation pass, the fourth argument is a Java object of type `TabbedWriter`. The invoked function uses that object to send content to the output source file.

Regardless of whether you coded the `invoke` function for the generation pass, you can add custom arguments to match the signature of the method being invoked. For example, you might have coded a template as follows:

```java
public void genInstantiation (Type type, Context ctx,
                               TabbedWriter out, Field arg){}
```

A related `invoke` method might be as follows:

```java
ctx.invoke (genInstantiation, field.getType(), ctx, out, field);
```
When you write the method being invoked, you use the custom arguments as you see fit.

The Context object also provides a second invocation method: invokeSuper. That method provides a way for you to invoke, from one method, the same-named method that is in another of your templates. That other template implements an interface at a higher level in the dynamic MOF hierarchy.

For example, the org.eclipse.edt.mof.egl.AnnotationType is a MOF model interface and is defined as follows:

```java
public interface AnnotationType extends EClass, Part
```

At this writing, if you invoke invokeSuper from a method in the template for AnnotationType, the order of events is as follows:

1. Search the templates related to EClass, including the templates related to all subtrees of EClass; but do not search the templates related to Part.
2. Search the template for AnnotationType itself.

However, this behavior is under review. For the current situation, see the Eclipse Bugzilla entry 387322 at the following site: [https://bugs.eclipse.org/bugs/show_bug.cgi?id=387322](https://bugs.eclipse.org/bugs/show_bug.cgi?id=387322).

Here are examples of the invokeSuper method:

```java
ctx.invokeSuper(this, preGenField, part, ctx, field);
ctx.invokeSuper(this, genExpression, stmt.getAssignment(), ctx, out);
```

The first argument in this case is always this: the template object from which the search begins. The subsequent arguments are as described for the invoke function.

**Flow of control at generation time**

The next sections outline the flow of control for a new generator.

**The command processor**

In an extended generator, the flow of control begins in a new command processor, which this tutorial calls EGL2Andy:

```java
public class EGL2Andy extends EGL2Java {
    public EGL2Andy() {
        super();
    }
```
public static void main(String[] args) {
    EGL2Andy genPart = new EGL2Andy();
    genPart.generate(
        args, new JavaCoreGenerator(genPart), null, null);
}

The `EGL2Andy` class extends `EGL2Java`, which is the base command processor for the EGL Java generator. Other extensions might extend `EGL2Andy` or another derived processor.

The `main` method accepts the command-line arguments that are provided by the EGL developer. The `EGL2Andy` constructor invokes the constructor for the `EGL2Java` superclass, and that constructor in turn invokes the constructor for the `AbstractGeneratorCommand` abstract class. The `AbstractGeneratorCommand` constructor defines a set of parameters that are available for any EGL generator and relies on its `CommandProcessor` superclass to store them for later use in the same run.

The last statement in the `main` method invokes the `generate` method in the `AbstractGeneratorCommand` class. That method accepts the command-line arguments from the EGL developer, along with a new `JavaCoreGenerator` instance, which extends the `Generator` abstract class.

In summary, two hierarchies are of interest:

- `EGL2Andy > EGL2Java > AbstractGeneratorCommand > CommandProcessor`
- `JavaCoreGenerator > Generator`

**The `generate` method in the `AbstractGeneratorCommand` class**

[ this section is wrong but suggestive.... ]

Here is the structure of the `generate` method in the Java generator:

```java
    public void generate(Part part) throws GenerationException {
        try {
            /* 1 */  context.putAttribute
                (context.getClass(),
                Constants.SubKey_partBeingGenerated, part);
```
Statement 1 places the part object into the Context class so that other logic can retrieve details about the part.

Statement 2 initiates the pregeneration pass. Specifically, the logic searches an inheritance chain for the \texttt{preGenPart} method, starting in the template class that is identified in a template.properties file entry. For example, if the part is stereotyped as a basic program, the following file entry applies, but on a single line:

\begin{verbatim}
egl.core.BasicProgram =
org.eclipse.edt.gen.java.templates.egl.core.BasicProgramTemplate
\end{verbatim}

In this case, the \texttt{preGenPart} method is not found in the \texttt{BasicProgramTemplate} class, and the search continues, as described in a later section. Assume that the method is found, and runs.

Statement 3 determines whether EGL error messages were issued during the pregeneration pass. If yes, the method ends. If no, statement 4 initiates the generation phase.

Any IOException is rethrown as a GenerationException. Any TemplateException issues error messages and, at this writing, sends them to \texttt{System.out}.

**The search for a method in a template class**

Previous sections mentioned the search that occurs after invocation of the Context object \texttt{invoke} method. This section gives further detail, as is needed for most generator extension.

The generator first seeks a template.properties file entry for the part being generated.

Here is the beginning of the IR XML format for a part named \texttt{Pgm1}:

\begin{verbatim}
<?xml version="1.0" encoding="iso-8859-1"?>
<Program ID="1"
\end{verbatim}
The search seeks a template.properties entry that matches the fully qualified name of the part: `programs.Pgm1`. If the search finds the entry and then finds that the class includes a method with the appropriate name and signature, the search ends successfully and the method runs.

However, the search continues in either of the following cases:

- No template.properties entry is present for that program; or
- An entry is present, but the template class does not include the method of interest.

You are unlikely to have written a template.properties entry for a specific part.

The subsequent search seeks one after another template.properties entry, each of which relates an EGL MOF model interface and a template class. You might think of the internal process as stepping up a ladder, with each rung representing a more abstract MOF model interface. Any template class that is found for a given MOF model interface is a candidate to contain the method of interest.

In the current example, the search next considers the MOF model interface for the `BasicProgram` stereotype; specifically, `egl.core.basicprogram`. No `properties.template` entry is present.

The search next considers a more general case: the program classifier; specifically, `org.eclipse.edt.mof.egl.Program`. That MOF model interface is represented in the following template.properties entry:

```
org.eclipse.edt.mof.egl.Program =
org.eclipse.edt.gen.java.templates.ProgramTemplate
```

The `ProgramTemplate` class does not include a method named `preGenPart`, and the subsequent search is yet more general, relying on the extension hierarchy of the MOF model interface for the Program classifier.

Here is the start of the hierarchy, starting with the EGL external type that represents the Program classifier:
The search seeks a properties-file entry for each of the MOF model interfaces in turn. In this case, a templates.properties entry is present for the Part interface:

```java
org.eclipse.edt.mof.egl.Part =
org.eclipse.edt.gen.java.templates.PartTemplate
```

The template class includes a method named `preGenPart`.

Here is some of that method, which stores general details and accesses another method in a template class:

```java
public void preGenPart(Part part, Context ctx) {
    ctx.putAttribute(ctx.getClass(),
                     Constants.SubKey_partLibrariesUsed,
                     new ArrayList<Library>());
    ctx.putAttribute(ctx.getClass(),
                     Constants.SubKey_partRecordsUsed,
                     new ArrayList<Record>());
    ctx.invoke(preGenClassBody, part, ctx);
}
```

Incidentally, the hierarchy of EGL MOF interfaces described earlier was simplified to ignore multiple inheritance. The LogicAndDataPart type extends both the StructPart and Container types. In practice, the search considers the extends hierarchy for an earlier-listed interface before considering the hierarchy for a later-listed one.

If you want to confirm the hierarchy, review the `org.eclipse.edt.mof.egl` EGL package in Eclipse, under the `org.eclipse.edt` folder, `core` subfolder. The search hierarchy cannot extend beyond Eobject, which is the basis of all the EGL MOF model interfaces.

In summary:

- The order of search is guided by a hierarchy of MOF model interfaces, from the specific to the general. Although you can add a specific part, the typical ordering is stereotype, classifier, and a more general hierarchy.
- At each step, if a template.properties entry is found and if the method of interest is in the template class that is referenced from that entry, the search ends and the method runs. If the entry is missing or if the method is not present, the search continues.

If the search fails, the generator throws a TemplateException.
Contributors

To extend a generator or to create a new one, you create a plugin that extends the following extension points:

• org.eclipse.edt.ide.core.GenerationContributors
  
  Your extension...

• org.eclipse.edt.ide.core.generators
  
  Your extension...

• org.eclipse.edt.ide.ui.edtGeneratorTabs
  
  Your extension defines the IDE additions that are displayed to the EGL developer.
Preparation steps

You install two kinds of software: an Eclipse installation at revision level 3.6 or greater, and the EDT plugins that you load there. You then set up your workspace.

Installing and setting up Eclipse

If you need to install a new version of Eclipse, do as follows:

1. Go to the Eclipse download page for version 3.7, which is named “Indigo”:
   [http://www.eclipse.org/downloads/packages/release/indigo/]
2. Click a package; for example, Eclipse IDE for Java EE Developers. A page is displayed with package details.
3. At the right, click the entry for the Windows® platform of interest; for example, Windows 32-bit. A download page is displayed.
4. Click a download site; for example, Indiana University [http]. A download begins.
5. Save the downloaded zip file and extract it in place. Later, you can delete the extracted code with no effect to the operating-system registry, which is not changed.

To start Eclipse:

1. Open the eclipse folder and right-click eclipse.exe. The Workspace Launcher dialog is displayed.
2. Specify a workspace location; then click OK.
3. Ensure that the Java compiler is set to 1.6 or higher:
   a. Click Window > Preferences. The Preferences page is displayed.
   b. Expand Java and then click Compiler.
   c. Set the Compiler compliance level. If necessary, update the Java installation on your machine.
   d. Click OK.

Installing the EDT plugins

To install the EDT plugins:

1. Go to the EDT download page [http://www.eclipse.org/edt/download/] and copy the address of a recent build. You might try the nightly build.
2. In Eclipse, click **Help > Install New Software**. The Install page is displayed.
3. Click **Add**. The Add Repository dialog is displayed.
4. In the **Name** field, type a description such as “EDT.” In the **Location** field, paste the repository address from step 1.
5. Click **OK**.
6. In the middle of the Install page, select the check box that identifies the build of interest.
7. Click **Next**. The Installation Details page is displayed.
8. Click **Next**. The Review Licenses page is displayed.
9. Select **I accept the terms of the license agreement** and click **Finish**.
10. In response to a request to restart Eclipse, click **Restart Now**.

**Creating the plugin for this tutorial**

You import an example plugin and then update a copy of it:

1. When Eclipse starts again, click **Window > Open Perspective**. The Open Perspective dialog is displayed.
2. Double-click **Java**.
3. In the Package Explorer view, right-click and then click **Import**. The Import dialog is displayed.
4. Expand **Plug-in Development** and then click **Plug-ins and Fragments**. Click **Next**. The Import Plugins and Fragments page is displayed.
5. At the first set of radio buttons, click **Target Definition** and ensure that the adjacent list box is set to **Running Platform**. At the last set of radio buttons, click **Projects with Source Folders**.
6. Click **Next**. The Selection page is displayed. Ensure that **Show latest version of plugins only** is selected near the bottom.
7. At the left, double-click **org.eclipse.edt.gen.generator.example**.
8. Click **Finish**.
9. In the Package Explorer view, right-click **org.eclipse.edt.gen.generator.example** and then click **Copy**.
10. Right-click in the view and then click **Paste**. The Copy Project dialog is displayed.
11. In the **Project name** text box, type **mygenerator.andy**. The name “andy” refers to a generator that creates output for Android™ devices.
12. Click **OK**.
13. Near the bottom of the plugin, double-click **plugin.properties** and reset the following properties:
For pluginName, type Andy Generator Plugin
b. For providerName, type My Company
c. For generatorName, type Andy Generator
d. For Description, type Gives practice

Save the file and close it. The changes provide values used in two EDT Eclipse extensions that you configure later.

14. Double-click plugin.xml. The plugin editor is displayed.
15. In the Overview tab, do as follows:
   a. For ID, type mygenerator.andy
   b. For version, type 1.0

The changes document the plugin.

16. Click the plugin.xml tab and click into the plugin.xml file.

17. At the bottom of the file are extension details that are commented out to ensure that the logic of the example plugin is not available to an EGL developer. To ensure that the logic of your plugin will be available, uncomment that detail.

18. Press Ctrl-F and, at the Find/Replace dialog, set the Find text box to ExampleGenerator and set the Replace text box to AndyGenerator. Click Replace All.

19. Save the file but don't close it.

20. In the Project Explorer, expand mygenerator.andy and then src.

21. Change the package names in the new project:
   a. Right-click org.eclipse.edt.gen.generator.example and then click Refactor > Rename. The first Rename Package page is displayed.
   b. In the New name text box, type mygenerator.andy.
   c. Check every check box and ensure that the File name patterns text box includes a single asterisk (*).
   d. Click Preview. A second Rename Package page is displayed. Ignore the message related to changing the name of a main method.
   e. Click Continue and, at the last page, click OK. The src folder now includes the following packages: mygenerator.andy.mygenerator.andy.ide, and mygenerator.andy.templates.

22. In the src folder, expand mygenerator.andy.

23. Right-click EGL2Example.java and then click Refactor > Rename. The first Rename Compilation Unit page is displayed. Use step 20 as a guide to change all instances of EGL2Example to EGL2Andy.

When you refactor the names in a Java class, you navigate two or three pages.
24. Similarly, change the following class names and related identifiers:
   a. ExampleGenerationContributor to AndyGenerationContributor
   b. ExampleGenerator to AndyGenerator

25. Expand mygenerator.andy.ide and change the following class names and related identifiers:
   a. EclipseExampleGenerator to EclipseAndyGenerator
   b. ExampleGeneratorTabProvider to AndyGeneratorTabProvider

26. In the mygenerator.andy.ide package, open the Activator.java file. At the bottom, a value in the initializeDefaultPreferences method sets the default directory for generated output. Change “generatedOutput” to “generatedJavaFromAndy” and then save the file and close it.

27. Still open in your workspace is the plugin.xml file, labeled as mygenerator.andy. The file defines extensions for the three extension points mentioned earlier.

28. To complete your extension to org.eclipse.edt.ide.core.GenerationContributors, update the attribute values in the GenerationContributor element:
   ◦ Set the id value: AndyGenerationContributor.
   ◦ Accept the class value, which refers to the new contributor that you provided in one of your plugin packages: mygenerator.andy.andyGenerationContributor.
   ◦ Accept the provider value, which refers to the following, new extension for the org.eclipse.edt.ide.core.generators extension point: mygenerator.andy.ide.andyGeneratorProvider.
   ◦ Accept the requires value, which references the id value for the generator contributor being extended. In this case, the required contributor is the Java generator, and the id value is JavaCoreGenerationContributor.

29. For org.eclipse.edt.ide.core.generators, consider the attribute values in the provider element:
   ◦ Accept the id value: mygenerator.andy.ide.andyGeneratorProvider.
   ◦ Accept the class value, which identifies the command processor: mygenerator.andy.andyGenerator.
   ◦ Accept the name value, which assigns a generator name for display in the IDE. The current entry references a value (GeneratorName) that you set in the plugin.properties file. That file is part of the Eclipse support for internationalization.
   ◦ Accept the compiler value, which is the id of the compiler that invokes this generator: org.eclipse.edt.ide.compiler.edtCompiler.
   ◦ Accept the version value, which assigns the generator version number for display in the IDE: 1.0.
Accept the language value, which assigns the language name for display in the IDE: Java.

Accept the description value, which references a value (GeneratorDescription) that you set in the plugin.properties file. The value is not used.

Accept the provider attribute value, which assigns the provider name for display in the IDE: Eclipse.

Accept the id value: mygenerator.andy.ide.AndyGeneratorProvider.

For org.eclipse.edt.ide.ui.edtGeneratorTabs, consider the attribute values in the generatorTab element:

- Accept the generatorID value, which refers to the new generator ID: mygenerator.andy.ide.AndyGeneratorProvider.
- Accept the class value, which references the class the structures the generator tab in the IDE: mygenerator.andy.ide.AndyGeneratorTabProvider.

Changing the character encoding to UTF-8

Consider using the Unicode standard UTF-8 to encode text files in your plugins. That encoding makes possible the inclusion of a wide variety of characters, from Latin to Chinese.

To set that encoding in the new plugin, do as follows:

1. Right-click mygenerator.andy.
2. Click Properties. The Properties page is displayed.
3. If the Resource pane is not there, click Resource to display that pane.
4. In the Text file encoding section, click Other and then select UTF-8.
5. Click OK.

Removing the old example

If you prefer to have fewer plugins in your workspace, remove the example:

1. Right-click org.eclipse.edt.gen.generator.example.
2. Click Delete. The Delete Resources page is displayed.
3. Click Delete project contents on disk. The original plugin will still be available in the Eclipse installation directory, and you can use that plugin as the basis of other generators.
4. Click OK.
Creating the launch configuration

You will view the affect of your changes by running a second Eclipse instance. Configure that instance as follows:

1. In the tree of configuration types, click **Eclipse Application**.
2. Click the “New” button.
3. In the **Name** field, type **EDT IDE**.
4. On the Main tab, in the **Program to Run** section, click **Run a product** and select **org.eclipse.platform.ide**.
5. Click the Arguments tab.
6. Set the program arguments:
   -os ${target.os} -ws ${target.ws}
   -arch ${target.arch} -nl ${target.nl} -consoleLog
7. Set the VM arguments:
   -Xms40m -Xmx768m -XX:PermSize=256m -XX:MaxPermSize=256m
8. Click **Apply** and then **Close**.
Reference

The reference topics are as follows:

- Command-line arguments
- InstallParameter
Command-line arguments

This topic describes the pre-existing command-line arguments. Each is based on an install parameter that is defined in the core or Java generator. When you invoke the generator, precede each argument with a hyphen and then specify the value.

For details on defining new install parameters, see “InstallParameter.”

**checkOverflow**

**Purpose:** To specify whether the generated code will check for numeric overflow.

Not checking for numeric overflow can result in smaller programs with better performance.

This option is not available for JavaScript generation.

**Variations:** -checkOverflow -overflow -co

**Status:** optional

**Input type:** Boolean

**Default:** false

**Parameter name:** Constants.parameter_output

**Class where defined:** EGL2Java

**output**

**Purpose:** To specify the path for the generated output. The path either is fully qualified or is relative to the directory at which the command is typed.

**Variations:** -output -out -o

**Status:** required

**Input type:** String

**Default:** none

**Parameter name:** Constants.parameter_output
Class where defined: AbstractGeneratorCommand

part

Purpose: To specify the fully qualified name of the part being generated.

At the command line, specify both the package and part; for example, “myPkg.MyProgram”. If you want to request generation of multiple parts in the root directory, use an asterisk as a wild card, either in place of the part name or at the end.

Do not specify this value in the IDE, where the part is specified for you.

Variations: -part -p

Status: required

Input type: String

Default: none

Parameter name: Constants.parameter_part

Class where defined: AbstractGeneratorCommand

report

Purpose: To create an HTML-formatted generated output that provides details on which generator source code caused which generated output.

Variations: -report

Status: optional

Input type: Boolean

Default: false

Parameter name: Constants.parameter_report

Class where defined: AbstractGeneratorCommand
root

**Purpose**: To specify the fully qualified path for the input. This entry defines the root location of the EGL binary files and typically refers to a project-specific EGLBin directory.

**Variations**: -root -r

**Status**: required

**Input type**: String

**Default**: none

**Parameter name**: Constants.parameter_root

**Class where defined**: AbstractGeneratorCommand
InstallParameter

You install parameters by invoking the following method in a class derived from the CommandProcessor class:

```java
installParameter(boolean required,
    String internalName,
    String[] aliases,
    Object[] possibleValues,
    String promptText
```

- **required**

  Indicates whether the argument is required to invoke the generator. The invocation fails if an argument is required and none is specified.

- **internalName**

  Specifies the parameter name in the Java code.

  Given that the Context object is `ctx`, you can code an invocation of the following form to access the value of a command-line argument:

  ```java
  (cast) ctx.getParameter(internalName)
  ```

  where

  - `cast` is Boolean, String, or String[], to reflect the type specified for `possibleValues` in the call to `installParameter`.
    - If the type specification is String[], use String for `cast`.
    - If the type specification is Object[], use String[] for `cast`.

  - `internalName` is the parameter name, as defined in the call for `internalName` in the call to `installParameter`.

- **aliases**

  Provides a set of valid argument names. The names are case insensitive, and the generator ignores any name that duplicates an alias already specified as an argument name for the generator. You must specify at least one name.

  A hyphen precedes each command-line argument, but do not specify that hyphen when assigning the argument name.
• **possibleValues**

The type of parameter and the valid argument values. The first array element indicates the default, and the `null` keyword indicates that any value is valid.

Here are examples by type, with additional comments that assume a parameter name of `xyz`:

- If the type is Boolean, you might specify the following value for `possibleValues`:
  ```java
  new Boolean[] {true, false}
  ```

  Then, any of the following command-line inputs evaluates to true:
  ```
  -xyz
  -xyz true
  -xyz yes
  ```

  And either of the following inputs evaluates to false:
  ```
  -xyz false
  -xyz no
  ```

- If the type is `String[]`, you are allowing a command-line argument to be a single string. You might specify the following value for `possibleValues`:
  ```java
  new String[] {"here", "there", "everywhere else"}
  ```

  Then, any of the following command-line inputs evaluates to “here”:
  ```
  -xyz
  -xyz here
  -xyz “here”
  ```

  And the following input evaluates to “everywhere else”:
  ```
  -xyz “everywhere else”
  ```

  In another example, you might specify the following value for `possibleValues`, making any input valid:
  ```java
  new String[] {null}
  ```

  If `required` is set to false, the value can be an empty string. Either of the following command-line inputs is valid:
  ```
  -xyz
  -xyz “odd case”
  ```

  However, if `required` is set to true, only the second of the previous command-line inputs is valid.

  In a third example, you might specify the following value for `possibleValues`, ensuring that “apple” is the default value:
  ```java
  new String[] {"apple", null}
  ```
In this case, the following command-line input is valid if `required` is set to false, in which case the value resolves to an empty string:

```
-xyz
```

The previous value is not valid if `required` is set to true.

- If the type is `Object[]`, you are allowing a command-line argument to be a series of strings. You might specify the following value for `possibleValues`:
  ```javascript
  new Object[] {"red", "green", "blue and grey"}
  ```

  The examples for `String[]` apply, but in this case, the command-line argument can include multiple strings, as shown here:
  ```
  -xyz red "blue and grey" green
  ```

  The returned values are provided to a string array in argument order.

- **promptText**

  The comment that is returned to the EGL developer in the following cases:

  - A command-line argument is preceded with a question mark instead of a hyphen, as in the following example invocation of a command processor named `EGL2New`:
    ```
    EGL2New ?color -process ?part
    ```

    In this case, the EGL developer receives the comments you specified for the color and part parameters.

  - The invocation includes a question mark in place of any command-line arguments, as in the following example:
    ```
    EGL2Andy ?
    ```

    In this case, the EGL developer receives the comments that you specified for every argument.
Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at www.ibm.com/legal/copytrade.html.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Android is a trademark of Google Inc.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, and Windows NT are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product or service names, may be trademarks or service marks of others.