Create more -- better -- code in Eclipse with JET

How to capture the best practices of experts and accelerate your model-driven development efforts

Chris Aniszczyk, Software Engineer, IBM, Software Group
Nathan Marz, Software Engineer, IBM, Software Group

Summary: The ability to create templates to codify best practices (that generate artifacts) is a powerful concept that can save you countless hours and reduce tedious coding. Get an introduction to the code generation framework, JET, which is an Eclipse technology project.

Date: 08 Aug 2006
Level: Introductory
Activity: 1737 views
Comments: 0 (Add comments)

Average rating

Code generation isn't a new concept. It's been around for a while and has been gaining popularity with the model-driven development (MDD) movement as a way to increase productivity. The Eclipse project has a technology project called JET that is a specialized code generator. JET can generate more than "code," however, and we refer to these noncode things as *artifacts* in this article.

JET? EMF? JET2?

An important thing new JET users need to understand is that the version of JET discussed here refers to an updated version of JET, commonly known as JET2. There is also an older version of JET in the Eclipse Modeling Framework (EMF) project that EMF used to generate code. JET2 is updated and part of the new Eclipse Modeling Framework Technology (EMFT) project. If you are curious about the old version of JET, see Resources.

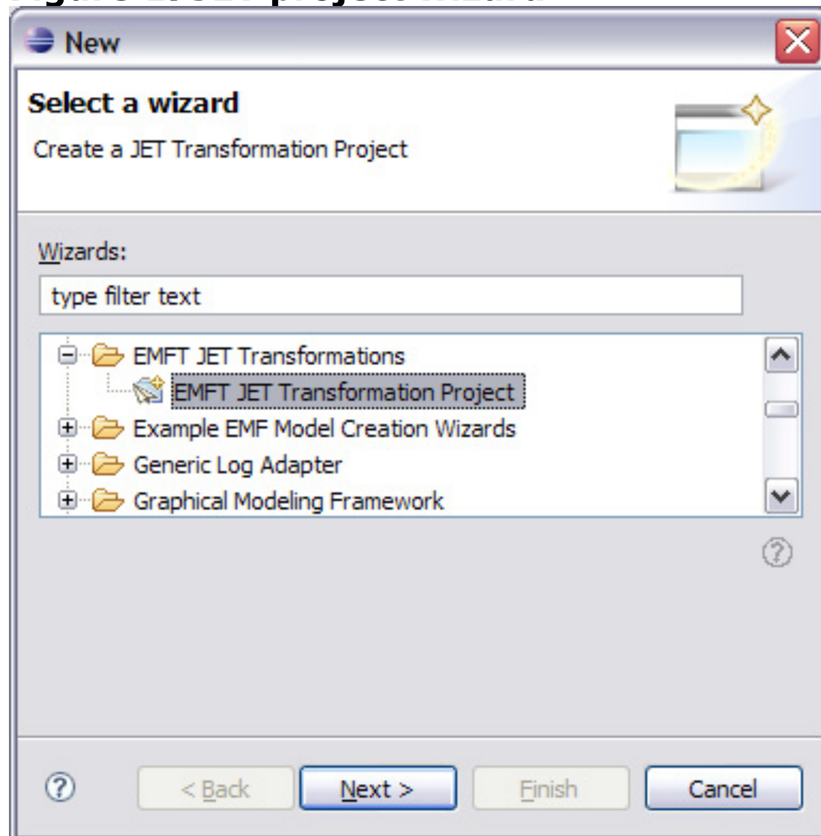
Getting started

In this section, we cover the basics of getting a JET project set up, discuss the structure of a JET project, and run a quick transformation.

Creating a JET project

Before we actually get our hands dirty with JET, we need to create a project. This is done using the standard Eclipse method of creating projects. In JET's case, we create a EMFT JET Transformation Project with the command **File > New > Other > EMFT JET Transformations > EMFT JET Transformation Project** (see Figure 1).

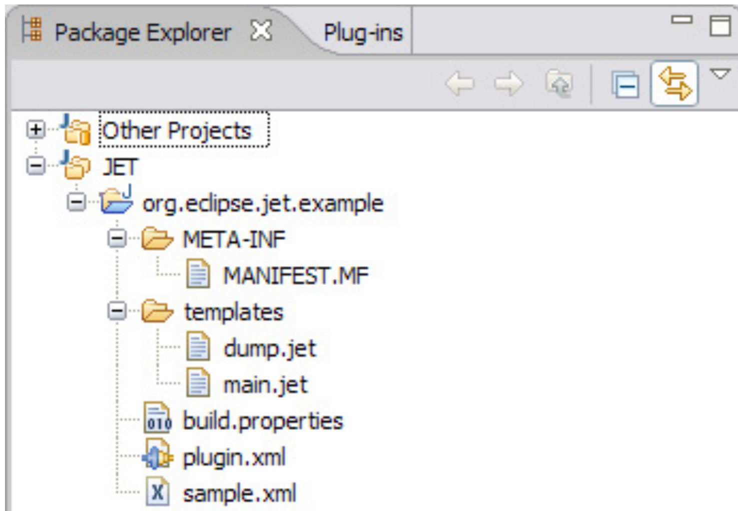
Figure 1. JET project wizard



JET project structure

To get an idea of how JET works, let's analyze the project structure. In previous section, we created a JET project (see Figure 2). In this JET project, we have six files we need to explore.

Figure 2. JET project structure



Eclipse project files (MANIFEST.MF, plugin.xml, build.properties)

These are the standard files created when one works with Eclipse projects. The important thing to note about these files is that in the plugin.xml, JET adds the extension org.eclipse.jet.transform automatically. By extending this extension point, we let JET know we are offering a JET transformation.

Control and template files (dump.jet, main.jet)

These are the template and control files used in your transformations. These will be discussed in detail in the Concepts section below.

Input model (sample.xml)

Here we find a sample input file used for our transformation. Note that this input can come from any source and isn't restricted to a project.

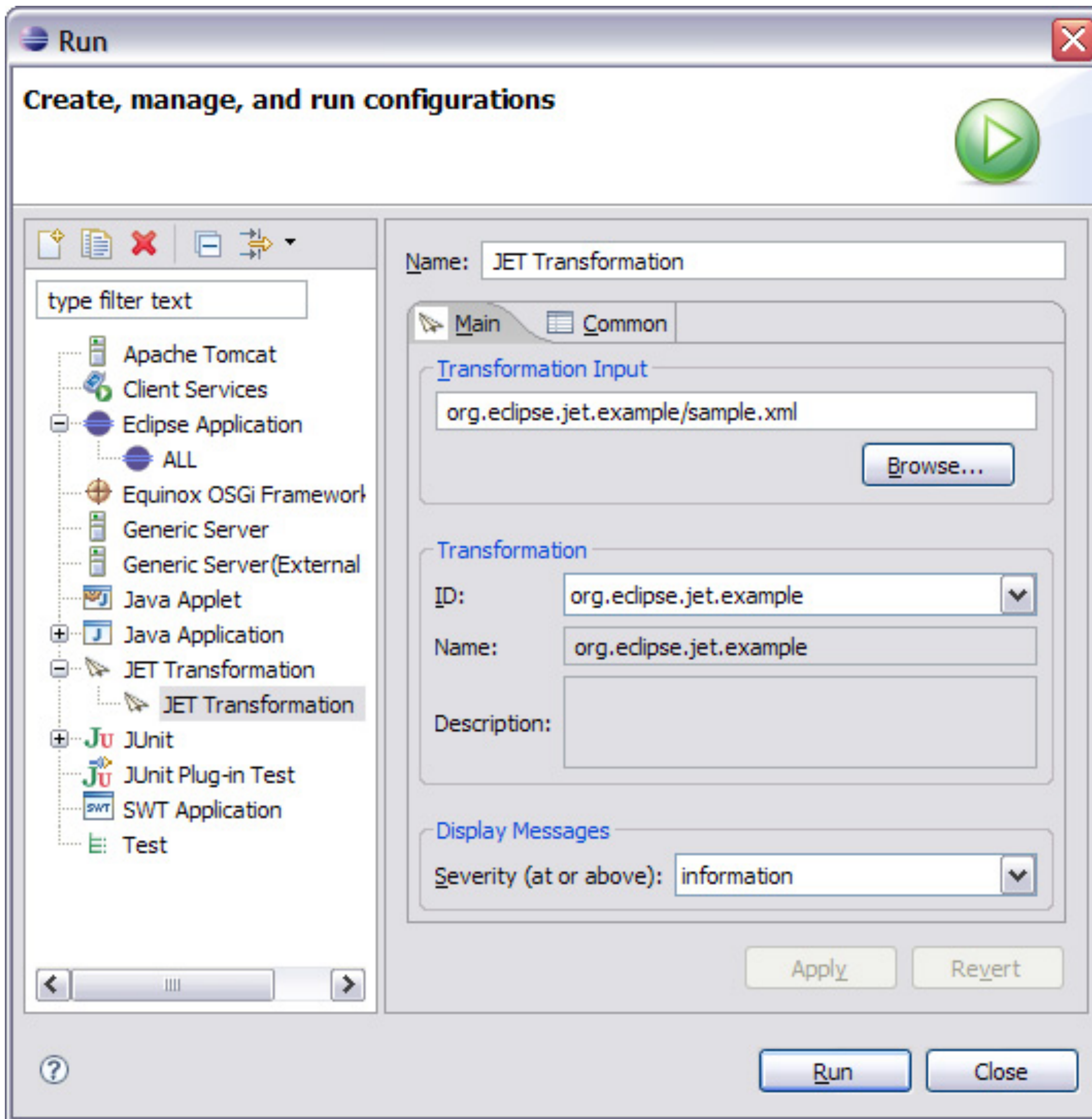
Changing the start template

By default, JET defines a start template to be main.jet. This option is configurable in the plugin.xml (org.eclipse.jet.transform extension) under the `startTemplate` attribute. There are other configuration options under this extension, so feel free to explore.

Running a JET transformation

Once we have a project in place with templates, control files and an input model, we can run a transformation. JET has provided a convenient way to invoke transformations via the familiar Eclipse concept of launch configurations (see Figure 3). To access the JET launch configuration, we go to **Run > JET Transformation**, fill in the appropriate options, and press **Run**.

Figure 3. JET launch configuration



Concepts

JET is a language for specifying templates that output artifacts. A collection of templates that implement some application is called a *blueprint* (in our terms). The paradigm of JET is the following equation:

Parameters + Blueprint = Desired Artifacts

A blueprint is created with JET, and the parameters are provided by a blueprint user. A blueprint consists of three distinct sets of files:

1. Parameters

The parameters to a blueprint are given in XML format. This allows great expressiveness, as XML allows for hierarchical relationships and for each node to have attributes. The input parameters are referred to as the *input model*. A

blueprint should define a schema describing what parameters it is expecting. For example, the following is an example input for a blueprint that creates a network sniffer:

Listing 1. Input for a network sniffer blueprint

```
<app project="NetworkSniffer" >
        <sniffer name="sampler" sample_probability=".7" >
                <logFile name="packet_types" />
                <packet type="TCP" subType="SYN" >
                        <logToFile name="packet_types" />
                        <findResponse type="TCP" subType="ACK" timeout="1" />
                </packet>
                <packet type="UDP" >
                        <logToFile name="packet_types" />
                </packet>
        </sniffer>
</app>
```

The blueprint would transform these input parameters into the code that implements this network sniffer. The parameters to the blueprint can be thought of as a custom programming language, and the blueprint will act like a "compiler" and transform the input into native artifacts.

2. Control files
These files control the execution of the code generation. The most important tag from the control tags is `<ws:file>`, which will execute a template and dump the results into the specified file. Execution of the code generation begins in main.jet, which is just like the main function of a program.

3. Template files
A template file specifies how and under what conditions to generate text. This text could be code, configuration files, or anything.

More about XPath

If XPath is a foreign concept to you, review the XPath Resources.

XPath

Since the input to any JET blueprint is an XML model, the XPath language is used to refer to nodes and attributes. In addition, XPath has its own way of using variables within an expression, which is heavily used within JET. They key points are the following:

1. A *path expression* is similar to a filesystem path. An path is a series of steps

separated by forward slashes (`/`).
2. The steps are evaluated from left to right and generally descend the model's tree as they do so.
3. Each step generally identifies tree nodes by their name (although other possibilities exist).
4. Steps may have an optional filter condition written in square brackets (`[` and `]`) at the end of the step.
5. An initial slash (`/`) indicates that an expression starts at the root of the model tree.
6. Path expressions can also start with a variable, which is a name preceded by a dollar sign (`$`).

The key points to remember about XPath within JET:

1. Variables are defined by several JET tags -- look for a `var` attribute. They may also be defined by the `c:setVariable` tag.
2. JET tags requiring a path expression have a select attribute.
3. Any tag attribute may include a dynamic XPath expression, which is an XPath expression surrounded by braces (`{` and `}`).

JET tags

The following examples will use the following input model.

Listing 2. Input model

```
<app middle="Hello" >
        <person name="Chris" gender="Male" />
        <person name="Nick" gender="Male" />
        <person name="Lee" gender="Male" />
        <person name="Yasmary" gender="Female" />
</app>
```

ws:file

This tag belongs in the *control* section of the blueprint, and it initiates a template. For example:

```
<ws:file template="templates/house.java.jet"
path="{$org.eclipse.jet.resource.project.name}/house1.java">
```

will run the house.java.jet template on the input model and dump the results at $(Project Root)/house1.java. The `{$org.eclipse.jet.resource.project.name}` is a dynamic XPath expression, which replaces that part of the string with the value of the org.eclipse.jet.resource.project.name variable. This variable is defined by

the JET engine.

c:get

This tag will write out the result of an XPath expression. For example, `Pre<c:get select="/app/@middle" />Post` will write out `PreHelloPost`. Notice that the `select` parameter takes an XPath expression. To use an XPath expression within parameters that expect a static string, dynamic XPath expression invocation can be done by wrapping the expression within braces (`{` and `}`).

c:iterate

This tag will iterate over nodes of a certain name, executing the body of the `iterate` for each node. For example:

```
<c:iterate select="/app/person" var="currNode" delimiter="," >
Name = <c:get select="$currNode/@name" />
</c:iterate>
```

will output `Name = Chris, Name = Nick, Name = Lee, Name = Yasmary`.

Iterate tags are also commonly used around start tags in the control templates. For example, if one wanted to create a Java™ class for each person in the model, the following would do the trick:

```
<c:iterate select="/app/person" var="currPerson">
<ws:file template="templates/PersonClass.java.jet"
path="{$org.eclipse.jet.resource.project.name}/{$currPerson/@name}.java"/>
</c:iterate>
```

This will create four Java classes in separate files: Chris.java, Nick.java, Lee.java, and Yasmary.java. Notice the `{$currPerson/@name}` string within the `path` attribute of the start tag. Since the `path` parameter doesn't expect an XPath expression (like `select` parameters do), the `{...}` characters signal the JET engine to replace that section of the string by evaluating the XPath expression within. `$currPerson/@name` tells the engine to replace that string with the `name` attribute of the `currPerson` node, which is a variable defined in the `iterate` tag.

Additionally, within the PersonClass.java.jet template, it can refer to the `currPerson` node variable defined in the `iterate` tag. For example, suppose PersonClass.java.jet looked like the following:

Listing 3. PersonClass.java.jet

```
class <c:get select="$currPerson/@name" />Person {
        public String getName() {
                return "<c:get select="$currPerson/@name" />";
        }
        public void shout() {
                System.out.println("Hello!!!");
        }
}
```

Yasmary.java will look like:

Listing 4. Yasmary.java

```
class YasmaryPerson {
        public String getName() {
        return "Yasmary";
    }
    public void shout() {
        System.out.println("Hello!!!");
    }
}
```

Lee.java will look like:

Listing 5. Lee.java

```
class LeePerson {
        public String getName() {
                return "Lee";
        }
        public void shout() {
                System.out.println("Hello!!!");
        }
}
```

`c:choose` and `c:when`

These tags allow the template to dump text conditionally depending on the value. For example, the following code:

Listing 6. c:choose/c:when Example

```
<c:iterate select="/app/person" var="p" >
        <c:choose select="$p/@gender" >
                <c:when test="'Male'" > Brother </c:when>
                <c:when test="'Female'" > Sister </c:when>
        </c:choose>
</c:iterate>
```

will output:

```
Brother
Brother
Brother
Sister
```

Notice that the `c:when` tags require a `test` parameter, which expects an XPath expression. Since we just want to compare the `select` parameter against a constant, we wrap the constant in single quotes (`''`).

c:set

This tag lets the template change the attributes of the input model on the fly. An example is when one string maps throughout the outputted text in many ways, like `Chris` may map to `Chris`, `chris`, `ChrisClass`, `CHRIS_CONSTANT`, etc. `c:set` will set the specified attribute to its contents. The following example stores an attribute called `className` for each person and simply adds the word `Class` after the name.

Listing 7. c:set example

```
<c:iterate select="/app/person" var="p" >
        <c:set select="$p" name="className" >
        <c:get select="$p/@name" />Class</c:set>
</c:iterate>
```

setVariable

This tag lets a template declare and use a global variable, using the full power of XPath to manipulate that variable at any point. For example, suppose one wanted to write out how many person nodes were provided in the input model. The following would work:

Listing 8. c:setVariable example

```
<c:setVariable select="0" var="i" />
        <c:iterate select="/app/person" var="p" >
                <c:setVariable select="$i+1" var="i" />
```

```
        </c:iterate>
Number of people = <c:get select="$i" />.
```

This outputs `Number of people = 4.`

Variables can be written out using `get`, as the above example shows.

There are more than 45 tags, which allows great expressiveness in outputting text. Much of the expressiveness comes from there being tags for conditional logic, dynamically changing the input model, and controlling the flow of execution.

Extending JET

JET is extensible through the use of Eclipse's extension point mechanism. Following are six extension points JET offers.

org.eclipse.jet.tagLibraries
: This extension point is responsible for defining tab libraries. JET already includes four tag libraries (control, format, workspace, java), if you want to add your own tag functionality, start here.

org.eclipse.jet.xpathFunctions
: This allows for custom XPath expressions available during JET XPath execution. An example of this is in JET's ability to use camelcase in XPath expressions by extending this extension point (see the CamelCaseFunction in the JET source code).

org.eclipse.jet.transform
: This is used to declare that your plug-in is offering a JET transform to the world. This is the place to change what start template you would use (instead of main.jet).

org.eclipse.jet.modelInspectors
: This allows you to define inspectors that enable the JET XPath engine to interpret loaded Java objects as XPath nodes. An inspector is an object that adapts objects to the XPath information model. As an example, JET uses a model to navigate the Eclipse workspace. Note that this is a provisional API, and that it is likely to evolve over time.

org.eclipse.jet.modelLoaders
: This allows you to define how models consumed by JET transformations and the JET `<c:load>` tag are loaded from the filesystem. An an example, JET provides the model loader org.eclipse.jet.resource, which will load an Eclipse IResource (file, folder, or project) and allow navigation of the Eclipse workspace from that resource.

org.eclipse.jet.deployTransforms
: This allows you to package a JET transform in a plug-in (bundle) for easy distribution. This can be used by UIs to see what transformations are available.

Example: Write code that writes code

The following example is a template for creating a class with an arbitrary amount of properties. Each property will have a getter and setter associated with it, along with some initial value. In addition, the template adds simple logging to each function by printing out to the command line the name of the function called.

Listing 9. Properties template

```
class <c:get select="/app/@class" /> {
<c:iterate select="/app/property" var="p" >
        private <c:get select="$p/@type" /> <c:get select="$p/@name" />;
</c:iterate>

        public <c:get select="/app/@class" />() {
        <c:iterate select="/app/property" var="p" >
                this.<c:get select="$p/@name" /> = <c:choose select="$p/@type" >
                <c:when test="'String'">"<c:get select="$p/@initial" />"</c:when>
                <c:otherwise><c:get select="$p/@initial" /></c:otherwise>
                </c:choose>
;
        </c:iterate>
        }

<c:iterate select="/app/property" var="p" >
        public void set<c:get select=\
        "camelCase($p/@name)" />(<c:get select="$p/@type" />
        <c:get select="$p/@name" />) {
                System.out.println\
                ("In set<c:get select=\
                "camelCase($p/@name)" />()");
                this.<c:get select="$p/@name" /> = <c:get select="$p/@name" />;
        }

        public <c:get select=\
        "$p/@type" /> get<c:get select="camelCase($p/@name)" />() {
                System.out.println("In get<c:get select="camelCase($p/@name)" />()");
                return <c:get select="$p/@name" />;
        }

</c:iterate>
}
```

Here is an example input model for this template:

Listing 10. Input parameters

```
<app class="Car">
        <property name="model" type="String" initial="Honda Accord" />
        <property name="horsepower" type="int" initial="140" />
        <property name="spareTires" type="boolean" initial="true" />
</app>
```

These input parameters generate the following class:

Listing 11. Generated class

```
class Car {
        private String model;
        private int horsepower;
        private boolean spareTires;

        public Car() {
                this.model = "Honda Accord";
                this.horsepower = 140;
                this.spareTires = true;
        }

        public void setModel(String model) {
                System.out.println("In setModel()");
                this.model = model;
        }

        public String getModel() {
                System.out.println("In getModel()");
                return model;
        }

        public void setHorsepower(int horsepower) {
                System.out.println("In setHorsepower()");
                this.horsepower = horsepower;
        }

        public int getHorsepower() {
                System.out.println("In getHorsepower()");
                return horsepower;
        }

        public void setSparetires(boolean spareTires) {
                System.out.println("In setSparetires()");
                this.spareTires = spareTires;
        }

        public boolean getSparetires() {
```

```
                System.out.println("In getSparetires()");
                return spareTires;
        }

}
```

Example: Write code that does stuff

To highlight the fact that JET doesn't need to be used only to generate code, the following example is a template that generates e-mail messages of varying moods. The purpose of each generated e-mail is to ask someone for various items. I provide the control file (main.jet) and the template it calls (email.jet).

Listing 12. main.jet

```
<c:iterate select="/app/email" var="currEmail" >
        <ws:file template="templates/email.jet"
        path="{$org.eclipse.jet.resource.project.name}/{$currEmail/@to}.txt" />
</c:iterate>
```

Listing 13. email.jet

```
<c:setVariable var="numItems" select="0" />
<c:iterate select="$currEmail/request" var="r">
        <c:setVariable var="numItems" select="$numItems+1" />
</c:iterate>
<c:set select="$currEmail" name="numItems"><c:get select="$numItems" /></c:set>
        <c:choose select="$currEmail/@mood" >
        <c:when test="'happy'">My dear</c:when>
        <c:when test="'neutral'">Dear</c:when>
        <c:when test="'angry'">My enemy</c:when>
</c:choose> <c:get select="$currEmail/@to" />,

I am writing you <c:choose select="$currEmail/@mood" >
<c:when test="'happy'">in joy </c:when>
<c:when test="'neutral'"></c:when>
<c:when test="'angry'">in burning anger </c:when>
</c:choose>to ask for <c:choose select="$currEmail/@numItems" >
<c:when test="1">
a <c:get select="$currEmail/request/@item" />.
</c:when>
<c:otherwise>
the following:

<c:setVariable var="i" select="0" />
<c:iterate select="$currEmail/request" var="r">
        <c:setVariable var="i" select="$i+1" />
```

```
        <c:get select="$i" />. <c:get select="$r/@item" />
</c:iterate>

</c:otherwise>
</c:choose>
<c:choose select="$currEmail/@mood">
        <c:when test="'happy'">Please</c:when>
        <c:when test="'neutral'">Please</c:when>
        <c:when test="'angry'">Either suffer my wrath, or</c:when>
</c:choose> send me <c:choose select="$currEmail/@numItems">
<c:when test="1">
this item</c:when>
<c:otherwise>
these items</c:otherwise>
</c:choose> <c:choose select="$currEmail/@mood" >
<c:when test="'happy'">at your earliest convenience.</c:when>
<c:when test="'neutral'">promptly.</c:when>
<c:when test="'angry'">immediately!</c:when>
</c:choose>

<c:choose select="$currEmail/@mood" >
<c:when test="'happy'">Your friend,</c:when>
<c:when test="'neutral'">Sincerely,</c:when>
<c:when test="'angry'">In rage,</c:when>
</c:choose>

<c:get select="/app/@from" />
```

Here is a sample input model for this template:

Listing 14. sample.xml

```
<app from="Nathan" >
        <email to="Chris" mood="angry" >
                <request item="well-written article" />
        </email>
        <email to="Nick" mood="happy" >
                <request item="Piano" />
                <request item="Lollipop" />
                <request item="Blank DVDs" />
        </email>
</app>
```

Applying the `mood` e-mail blueprint on these parameters generates the following two files.

Listing 15. Chris.txt

```
My enemy Chris,

I am writing you in burning anger to ask for a well-written article.
Either suffer my wrath, or send me this item immediately!

In rage,
Nathan
```

Listing 16. Nick.txt

```
My dear Nick,

I am writing you in joy to ask for the following:

1. Piano
2. Lollipop
3. Blank DVDs

Please send me these items at your earliest convenience.

Your friend,
Nathan
```

Conclusion

Before we conclude, we would like to thank Paul Elder for his valuable comments. On the whole, JET can be used for more than just simple code generation. JET is a new Eclipse technology project we expect many developers will use in their endeavors.

Resources

Learn

- Find more information about JET at the Eclipse Modeling Framework Technologies JET project.
- Learn more about the Eclipse Modeling Framework Technology (EMFT) project at Eclipse.org.
- Learn about code generation using another technology, XSLT, from the developerWorks tutorial "Code generation using XSLT."
- For more information, read the "JET Tutorial Part 1 (Introduction to JET)."

- Learn about more advanced topics of the old version of JET by reading "JET Tutorial Part 2 (Write Code that Writes Code)."
- Learn how to effectively process XML using XPath by reading "Effective XML processing with DOM and XPath in Java."
- Review the W3C XPath 1.0 Specification for more information.
- Learn more about XPath by reviewing XPath by Example.
- Learn more about XPath by reviewing XPath Introduction.
- Stay current regarding Eclipse happenings by visiting Planet Eclipse
- Learn more about the Eclipse Foundation and its many projects.
- For an excellent introduction to the Eclipse platform, see "Getting started with the Eclipse Platform."
- Expand your Eclipse skills by visiting IBM developerWorks' Eclipse project resources.
- Browse all of the Eclipse content on developerWorks.
- Visit the developerWorks Open source zone for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.
- Stay current with developerWorks technical events and webcasts.

Get products and technologies

- Check out the latest Eclipse technology downloads at IBM alphaWorks.
- Innovate your next open source development project with IBM trial software, available for download or on DVD.

Discuss

- The EMFT EMFT newsgroup is a great place to start if you have questions about JET.

- Want to contribute to JET? Discuss your contributions on the EMFT mailing list.
- The Eclipse newsgroups has lots of resources for people interested in using and extending Eclipse.
- Get involved in the developerWorks community by participating in developerWorks blogs.

About the authors

Chris Aniszczyk is a software engineer at IBM Lotus focusing on OSGi related development. He is an open source enthusiast at heart, and he works on the Gentoo Linux distribution and is a committer on a few Eclipse projects (PDE, ECF, EMFT). He's always available to discuss open source and Eclipse over a frosty beverage.

Nathan Marz is a student at Stanford University and a graduate of IBM's Extreme Blue internship program. He was a technical intern on the Blueprint project.

Trademarks | My developerWorks terms and conditions