

*Take it easy,  
let your computer do the work*



*easyMda*  
*User's guide*

*easyMda*

#### Disclaimer

This open-source project is much inspired by an (similar) excellent project Taylor Mda by John Glibert. You may recognize a number of class- and method-names, but because the model (Java) is completely different from the model used in Taylor Mda (UML), almost all code is new.

easyMda user's guide, version 1.7.01 27. Aug. 2014  
robin.bakkerus@gmail.com

The main being used is OpenDyslexis (see <http://dyslexicfonts.com/>)  
Cover design : Niels Bakkerus ([www.bakkerus.nl](http://www.bakkerus.nl) )

## Table of Contents

Introduction.....	5
Part 1, Create your own cartridge the easy way.....	6
Part 1, Overall source-code project setup.....	14
Conclusion.....	18
Part 2, Enhance existing, and/or create new, jet-template.....	19
The Jet engine.....	20
Jet argument.....	23
Common Api.....	25
The Common api.....	26
How to replace that classname and package.....	28
How to manage those import statements.....	30
Generate lines of code at specific location with BufferUtils.....	32
How to generate JavaDoc.....	34
How to keep manual code, after regenerating an output file.....	34
How to retrieve data from other resources.....	35
How to organize your Jet files.....	41
AppUtils.....	42
TypeUtils.....	42
NameUtils.....	50
InterfaceUtils.....	51
PostProcessorUtils.....	53
Other Utils.....	53
Part III, Create your own cartridge project.....	54
Create jet enabled project.....	54
Create your api, to support the new jet-templates [optionally].....	58
Create your jet-templates.....	59
Register your jet templates.....	59
How is the final filename generated.....	63
Keep your templates manageable.....	64
Deploy this project into the easyMda plugin.....	65
Additional 3-party lib(s).....	65

Debugging your jet template(s) and api.....	66
Logging.....	68

## ***Introduction***

This developers guide is divided in five parts:

1. The first part describes how to create your own cartridge (code-generator), the fast and easy way.
2. This describes how to setup the workspace and what is the role of all the different project.
3. This describes more in detail how to enhance the current cartridge (or module)
4. This describes how to enhance the current api and also to create new api's
5. The last part describes the generator itself.

### **Note**

In the remainder of this document the term 'cartridge' will be used to describe an Eclipse java project that contains the Jet engine templates. Such a project may contain several template trees. Each template tree may contain one or more template-branches, and each branch may contain one or more (Jet engine) template files. Hence the cartridge is more or less synonym to the template-trees.

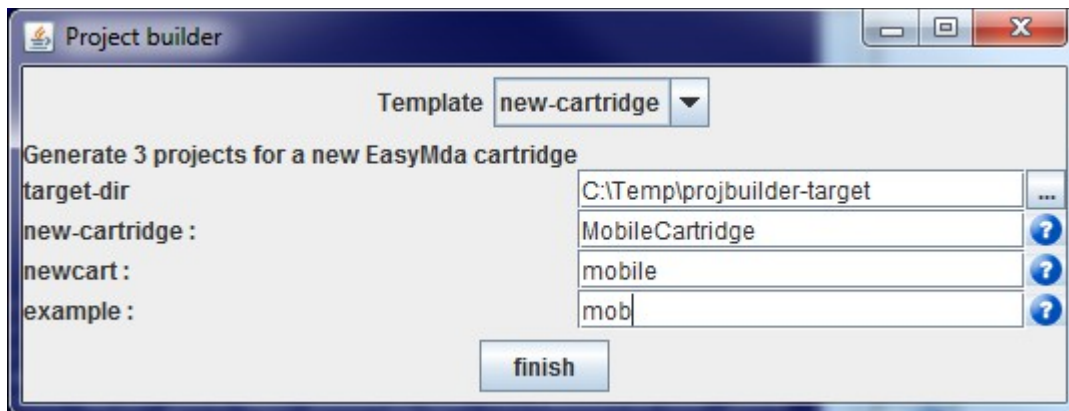
## Part 1, Create your own cartridge the easy way.

In the developer's guide I described how to use an existing cartridge. If you want to enhance an existing cartridge, like the webapp cartridge, you should first clone the source and prepare a workspace, in order to do so. This is described in the next chapter. But you may also want to create a new code-generator cartridge from scratch yourself. For that, it is not required to clone the source-code. The plug-in is required though, and you want to download the following zip-file from EayMda download page:

projectbuilder-1.7.zip

This contains a small but useful program and some projects templates, that can be used to kick-start your new cartridge.

- Download this zip-file, unzip into whatever location.
- Goto this target location
- Depending on your OS, run the bat file or bash (.sh) script
- In the combobox, select “new-cartridge” , this will result in:



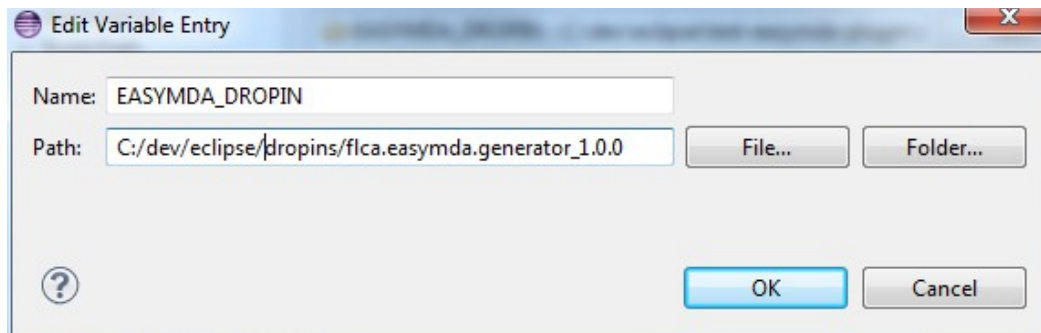
- Fill in the values. When you hover over a field it will give some explanation what is the purpose of this field, and finally hit [finish]

After this step three new Eclipse project are created in target-dir that you selected.

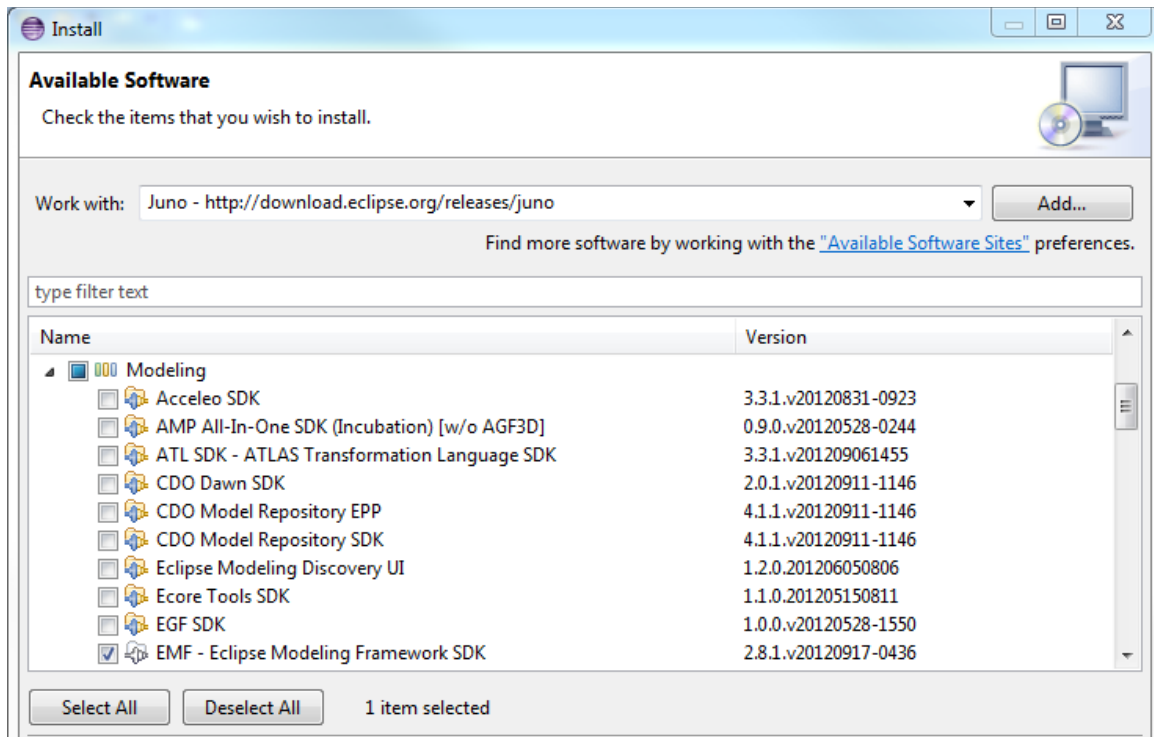
- Start an Eclipse that contains the EasyMda plug-in.  
Hence an Eclipse version where you copied the `flca.easymda.codegenerator.1.0.0` folder from the `EasyMda-1.8.zip` under the `eclipse/dropins` folder.
- Import the thee new project's in your Eclipse workspace.

Now you have all the projects that you need initially to create your own code generator. But unfortunately, after importing these three projects you will have a number of errors, that we need to fix first, before we can run our first test.

- 1) Setup a Java build path variable: **EASYMDA\_DROPIN** that points to the dropins folder of Eclipse from above, where you installed the EasyMda plug-in, for example:



- 2) Depending on your Eclipse version, you may be okay now, meaning that you got rid of all errors. But not all Eclipse version (like Eclipse classic) contain the modules needed for the Jet engine. So if you see errors like: `missing org.eclipse.emf.codegen ..` you should install one extra Eclipse module with:  
Help 'install new software'  
Add from the standard Eclipse site the :  
EMF Eclipse modeling framework SDK, see:



- 3) And finally you have to fix the Java error that I put there deliberately, because you have to setup the correct path.

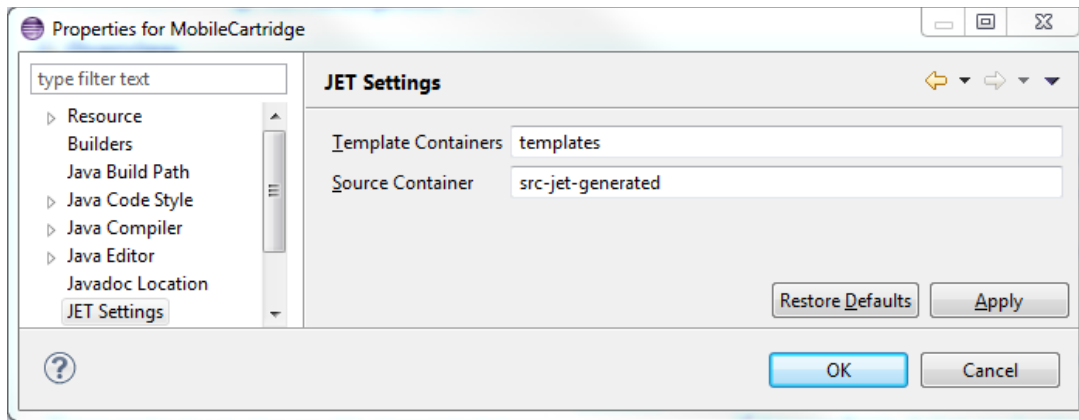
In the file `XxxxData` you have to fix the error so that this variable point to dropins folder (from above) where you installed the plug-in (using the Linux delimiter) for example:

```
String ECLIPSE_DROPIN_DIR = "c:/dev/eclipse/test-easymda-plugin/dropins" ;
```

## NOTE

I noticed that Eclipse sometimes 'forgets' a specific project setting. In particular the Jet Settings is sometimes gone, So you should check this setting on the `XxxCartridge` project (with Project properties Jet Settings that will show the form below.





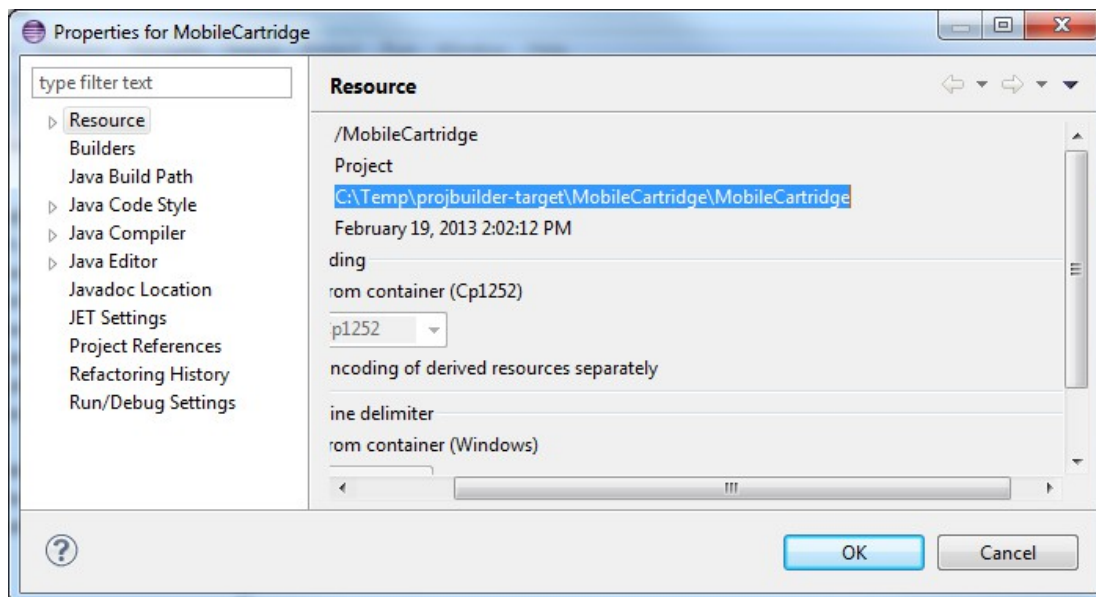
If the "Source Container" field is, empty you should fill in: **src-jet-generated**<sup>1</sup>

Final step:

The XxxCartridge project has all the minimum that are needed to become a valid cartridge. In fact you are almost ready to execute a first test-run. The only thing left is to 'register' this cartridge at the EasyMda plug-in.

You do that as follows:

- Select your new cartridge, right-click and ask for the properties :



---

<sup>1</sup> Or some folder if this is valid source-code folder.

In the Resource tab, copy the project name.

Then goto the `eclipse/dropins/flca.easymda.codegenerator.1.0.0/cartridges` folder

Open the `example.link` file and

replace this line:

```
#path=/media/c/mydocs/robin/Project/...etc
```

with

```
path=<paste the project-name that copied above>/bin
```

### Notes

Make sure to use the correct path delimiters, that means on both Linux and Windows, use the Linux convention, hence only use: `"/"` for example:

```
path=/media/c/mydocs/robin/flca/com.flca.mda/flca.mda.templates.webapp/bin
```

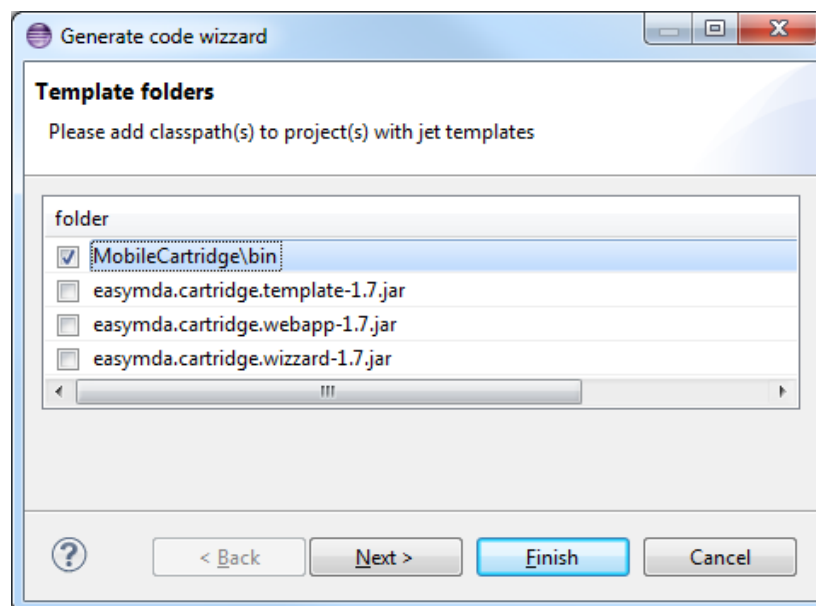
Because this path must contain the actual classpath, you should add the `"/bin"` after the project name. And don't forget to delete the `"#"` comment

Now you are ready to test !

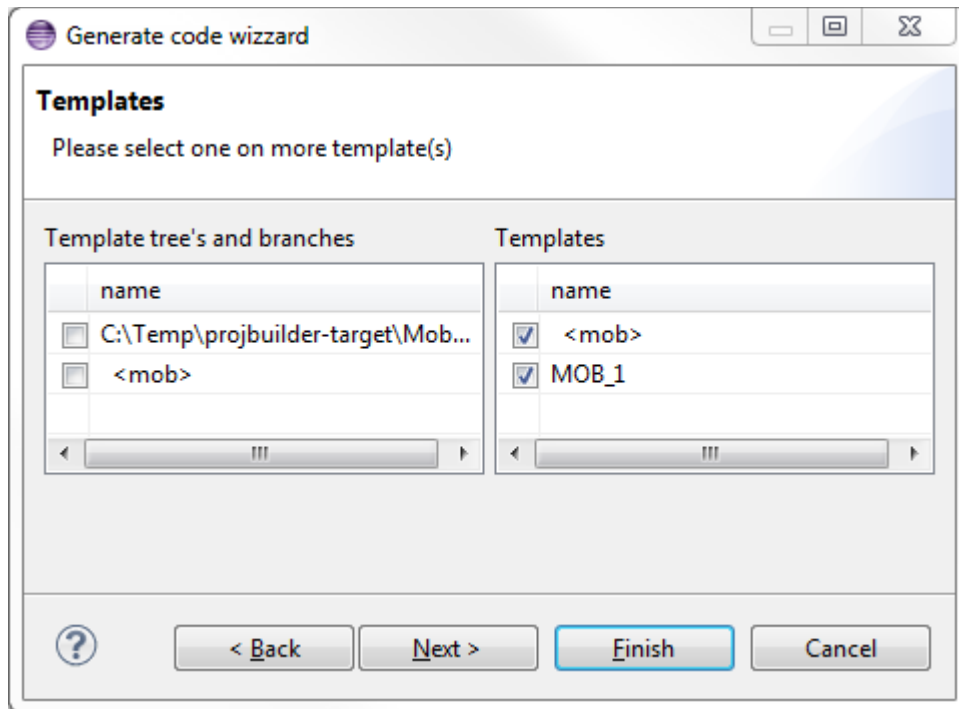
So lets first generate some code the standard way via the plug-in.

- Goto the XxxModel project, and select a class(es) or package(s) via the Package explorer in the Java perspective.
- Right-click and hit the "generate-code" item at the bottom.

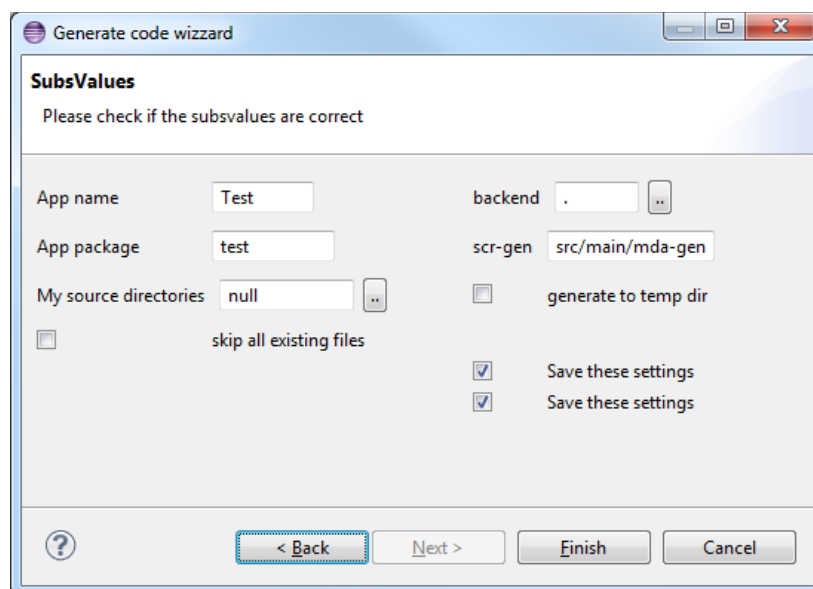
This brings up the following wizard:



If you setup the .link file correctly in the dropins/flca.easymda.codegenertor/cartrdiges folder (as described above), you should see your XxxCartridge project listed. Select this one and hit [Next].



Select the template and hit [Next] again



The important fields in this form are the :

- “backend” that indicates what will be the target project. Via the [...] button you can select a fixed path or you can manually fill in a relative path. The “.” indicates for now we will generate the code in the *XxxModel* project itself.
- “src-gen” indicates the target folder under the project above.
- “App-package” tells the generator that the base-package defined in the Model is substituted with this package name.

Hit [finish].

In the console window, the following message is listed:

```
generated C:\Temp\MobileCartridge\MobileModel\src\main\mda-gen\test\demo\Demo1.java
```

Congratulations, you just created your first own generated code!

However small this maybe, it is a solid foundation to build your actual generator. The 'only' thing left is to take your example source-code, copy/paste these into corresponding jet template files and replace all tokens that need to be parametrized, and register these template files accordingly. You may also want to develop your own interfaces, annotations and/or api if needed. The most important asset however is your example code! The better that code is in terms of solid coding standards like: high-cohesion, low-coupling, the easier it is to parametrize it!

While developing your new code-generator, do you always have to go through the wizards pages from above? No, we can shorten the develop-test-review cycle even more. Here the *XxxTestGenerate* project plays a key role! This project has a junit test class (*TestXxxCartridge*) that extends from *TestTemplatesBase* that looks like:

```
public class TestMobileCartridge extends TestTemplatesBase {
    @BeforeClass
    public static void beforeOnce() {
        TestTemplatesBase.beforeOnceBase(new MobileData());
    }

    @Test
    public void testMob() {
```

```
generate(Demo1.class, MobileConstants.getTemplate(TidMobile.MOB_1));  
}
```

In the `beforeOnce()` method, the generator basically feed-ed with all input that it would get otherwise via the Gui wizard. In the `testMob()` method you tell the generator explicitly to generate code for the `Demo1` class with the (compiled) Jet template class indicated via the enum constant name: `TidMobile.MOB_1`.

Rather than going through the wizard, you can now simply run this jUnit test. Because the last generated code will be on the clipboard, you able to review the generated code very quickly, so this gives you a very fast development cycle!

This ends this chapter. In the next chapter we going to explore the projects that you want to clone (via Git) from the EsayMda website.

So to summarize:

- Download the EasyMda-1.8.zip and projectbuilder-1.7.zip
- Install the EasyMda plug-in in the Eclipse dropins folder
- In the projectbuilder folder, run the .bat or .sh script
- In Eclipse (with the plug-in), open the three projects generated above.
- Fix the errors by creating new classpath varb EASYMDA\_DROPIN
- If needed: install new software: Modelling – EMF Modelling frameworks SDK
- Fix the java error at: `ECLIPSE_DROPIN_DIR = ...! .`
- Register the new cartridge, by changing the example.link (or creating a new xxx.link) file in: `<eclipse>/dropins/flca.easymda.codegenerator/cartrdiges` so that it point to the run-time folder (/bin) of the new cartridge project.

#### Note

Most of the steps above are a one-time setup to create a new 'EasyMda-enabled' Eclipse workspace.

## Part 1, Overall source-code project setup

In the previous chapter we saw how to kick-start a new code generator. There did not access the actual source. If you want to do more, or want to look inside the code, you should first grep the source code and do some preparations to get a working eclipse workspace. I will first explain these steps, and then I will tell more about all the different projects.

You should first clone the Git easymda project from Google project with:

```
git clone https://robin.bakkerus@code.google.com/p/easymda/
```

Because this plugin is an Eclipse plugin you should have a recent Eclipse version that supports Java-7, preferably a Juno release (Classic or JEE).

If you want to use the build tools, you should download Gradle (from [www.gradle.org](http://www.gradle.org)) but that is not required.

After you cloned the Git project, you can import all the projects into Eclipse. After this step probably have a large number of errors that we need to fix first. To fix these errors you have to the following.

- 4) Setup a Java build path variable: **M3\_REPO** that point to the m3\_repo folder containing all the external jar files. These jar files can be obtained from the download page via two zipfiles: `ext_m3_repo-1.7.jar` and `flca_m3_repo-1.8.jar`<sup>2</sup>

- 5) Import user libraries, using the file: `flca.easymda.plugin/eclipse.userlibraries`

**NOTE:** you should first search/replace the m3-repo, so that it points to the correct location!

The eclipse command is:

Window preferences Java User-libraries' Import ...

- 6) Depending on your Eclipse version, you may be okay now, meaning that you got rid of all errors. But it is also likely that you still have errors like:

`missing org.eclipse.emf.codegen ..`

In that case you should install one extra Eclipse module with:

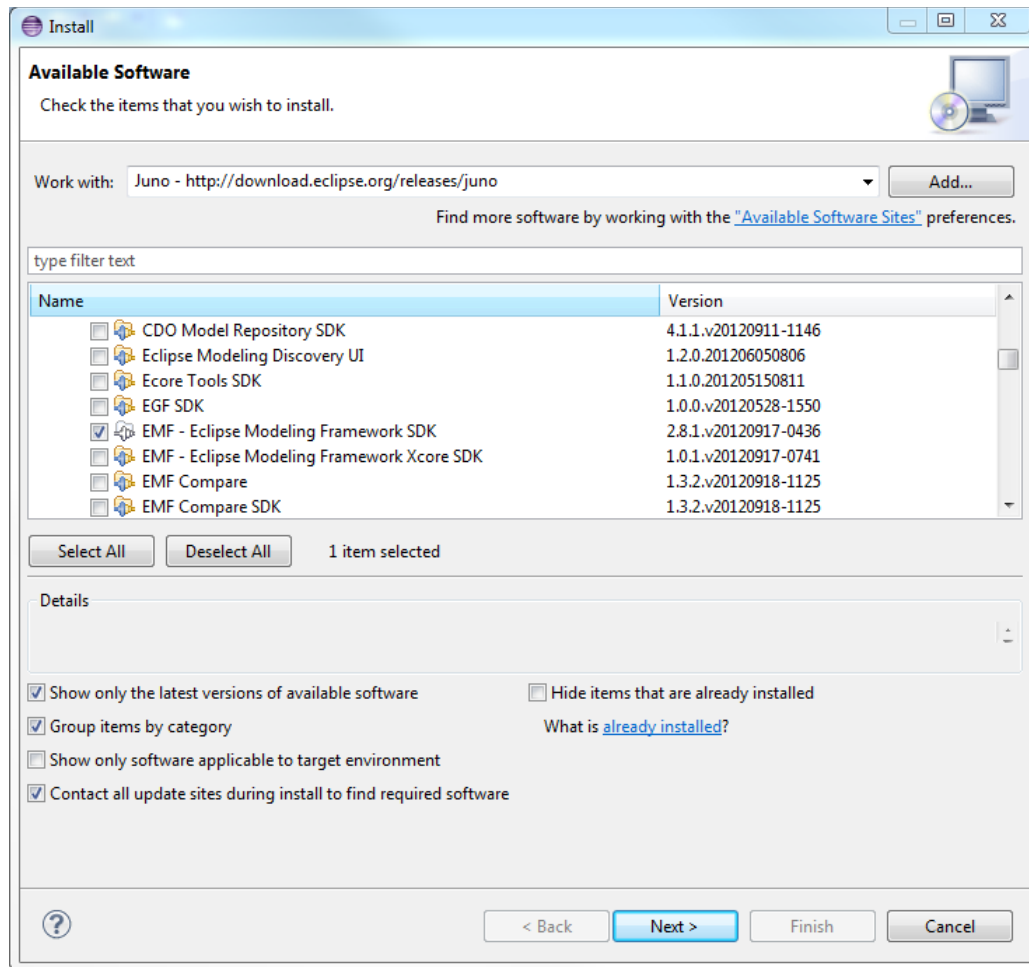
---

<sup>2</sup> I split these into 2 zip-files because the 'ext\_m3\_repo' is very static whereas the 'flca\_m3\_repo.zip may change frequently.

Help 'install new software'

Add from the standard Eclipse site the :

EMF Eclipse modeling framework SDK, see:



Note that EMF Modeling framework is located under the **Modeling** group.

Now all error should be gone!

You will see the following directory structure:

```
flca.easymda.plugin
flca.frw.backend
flca.frw.backend.appengine
flca.frw.backend.berkeley
```

flca.frw.backend.cache  
flca.frw.backend.jpaa  
flca.frw.backend.perst  
flca.frw.common  
flca.frw.frontend.flex  
flca.frw.frontend.zk  
flca.mda.cartridge.mobile  
flca.mda.cartridge.template  
flca.mda.cartridge.webapp  
flca.mda.cartridge.wizzard  
flca.mda.common.api  
flca.mda.generator  
flca.mda.test.api  
flca.mda.test.generate  
flca.mda.test.model.template  
flca.mda.test.model.webapp  
flca.mda.test.model.wizard  
flca.mda.test.target

This looks like an impressive number of Eclipse projects, that will take ages to digest, but in fact it is not that much at all, and you should be able to find your way easily.

Let's examine these projects more in detail:

#### **flca.easymda.plugin**

This is just a resource project, that contains some files like :

- Apache OpenOffice documentation files
- Javadoc files.
- Support file for the Gradle build tool.

#### **flca.frw.backend and flca.frw.common**

These are two projects that contain some api that is used by the generated webapp backend project. The flca.frw.backend depends on flca.frw.common.

When you are creating your own cartridge, you can ignore these 2 projects. The same is true for the following projects:

**flca.frw.backend.jpaa, flca.frw.backend.berkeley, flca.frw.backend.cache,  
flca.frw.backend.prest, flca.frw.backend.appengine**



These are tiny projects used in conjunction with `flca.frw.backend` for a specific persistence technology

**`flca.frw.frontend.flex`, `flca.frw.frontend.zk`**

These are project that contain some api used by respectively a flex based or zk based frontend.

**`flca.mda.test.model.template`, `flca.mda.test.model.webapp`, `flca.mda.test.model.wizard`**

These a model projects that I used to the corresponding cartridges.

**`flca.mda.test.api`**

This is project that contains a number of jUnit tests inside other projects. I have these jUnit tests in this dedicated project because I did not want to pollute the actual projects with references to model or target projects.

When you create your own project, you should think about to put your jUnit tests when these tests (to be valuable) need to reference classes 'outside' the cartridge project itself. Rather than copying such classes into the cartridge project itself (under `java/test`) or having a dependency to some other project, I opted for this dedicated test project.

**`flca.mda.test.generate`**

This project also contain jUnit tests, but for a different purpose. With these jUnit test you can generate one or more (or all) files, without the hassle to first start the plug-in, going through all the screens and than hit [finish]. This is possible because these jUnit test extend a base-class that (in the `@BeforeOnce` method) will execute the same tasks as done by the plug-in before hitting the [finish] button. So this is a great tool to test any new template that you create.

**`flca.mda.test.target`**

This is test target project that is referenced by the test project above.

**`flca.mda.generator`**

This is the Eclipse project that contains the actual plug-in!

Unless you want to do something very specific, you should be able to use this plug-in as-as,

when you create your own cartridge. But in order to use your new cartridge, you must put either the corresponding jar file, or xxx.link file pointing to the bin folder of this cartridge project, in the cartridges folder. That's all. This will be explained more in detail in a next chapter.

#### **flca.mda.common.api**

This is a project that contain general api that can be used by any (new) cartridge. The the plug-in project from above depends on this projects and therefor you will find the flca.mda.common.api-1.8.jar inside its \lib folder.

#### **flca.mda.cartridge.template, flca.mda.cartridge.webapp, flca.mda.cartridge.wizzard**

These are the Eclipse Jet-enabled projects that contain the actual cartridges. These are accessible, because the corresponding jar files can be found the \cartridges folder of the flca.mda.generator project.

### *Conclusion*

When you create you cartridge from scratch you only really need the following projects :

- flca.mda.generator
- flca.mda.common.api

You may want to make your own copy of :

- flca.mda.test.generate
- flca.mda.test.api

To test your own api and template(s).

And you should make your own :

- cartridge project
- corresponding test model project
- corresponding test target project if needed.

## Part 2, Enhance existing, and/or create new, jet-template.

I bet, that if you are an experienced developer, and you (and your team) gained good results with a particular framework, then probably the first thing you want to, once you played with easyMda, is to find out how you could alter the cartridge, so that it will generate precisely that code, that you are already comfortable with.

After looking into this developer's guide, you will find out how easy this can be accomplished. That is exactly the main objective of easyMda, to provide a tool that allows you to develop (new) templates fast and easy, to automate repetitive portions of the source code base. Not only think of the usual suspects, like crud apps, to generate, but any part that is repetitive, like workflows, pageflows, etc.

The wizard cartridge is not more than a demo project to show how easyMda works, nevertheless it generates perfectly valid java code. The webapp cartridge, is much more elaborate, and I have used it successfully in many projects. I deliberately made webapp so that it is not dependent on an existing framework, like Spring<sup>3</sup>, because this way it is easier to adapt it to your needs.

Ok, let's delve into easyMda.

The power of easyMda is based on three cornerstones:

- Eclipse Jet-engine  
In my opinion the most powerful and easy to use generator.
- Java reflection  
This gives the ability to examine in detail, a class, interface, properties, method argument etc, at run time.
- Inheritance  
This gives the possibility to define an interface with methods that should be implemented by user.
- Annotations  
This is similar to the power of interfaces from above, but this can be used in a much

---

<sup>3</sup> I want to publish a Spring bases webapp (springapp) in the near future though.

more fine-grained way.

One important feature that makes easyMda, so easy, is that it can make use of the power that's already inside Eclipse. The Jet-engine is one thing, **intellisense** and **auto-completion** are other very powerful features.

### ***The Jet engine***

I will not go into the details about the jet engine and jet nature itself, because there are many tutorials out there. If you are new to the Jet engine then take a look at:

[http://www.eclipse.org/articles/Article-JET/jet\\_tutorial1.html](http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html)

In short, the Jet engine is a templating tool very similar to **JSP**. Like a jsp file, a jet file is automatically translated into a java file, that will generate the final code, that can be anything you want: java, xml, txt etc. Like jsp, in which you embed java code, the entire java stack can be used which makes it so powerful. One very nice feature of the jet nature is that each time you save a jet file, it is immediately translated into a java file, and if automatic build is enabled, immediately compiled. If the compiled class does not have build errors, you know for sure that the generator is correct, at least from the jet-side point of view. At run-time you may still encounter errors because the api may throw an error, but at least the jet syntax is correct.

A jet file is created under the **templates** folder of Jet enabled java project. You may organize the jet files under sub-folders under the **templates** folder. In general a jet file has an extension that contains the word "jet". A typical layout may look like:

```
templates/backend/Entity.javajet
        EntityDao.javajet
:
templates/frontend/flex/Entity.asjet
        :
```

Eclipse comes with a default Jet editor which is not very powerful. I advise you install the Jet editor <todo>

This editor has features like intellisense and auto-completion.

### Note

Don't forget to associate the jet file with the new editor.

<todo>

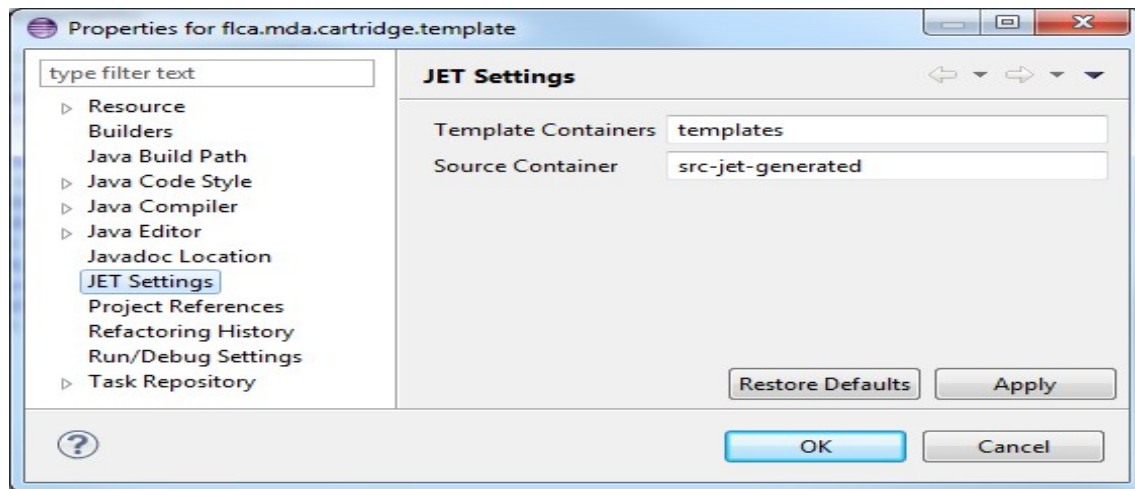
A minimal jet file looks like:

```
<%@ jet package="backend" class="EntityModel" [1]
imports="flca.mda.common.api.util.* com.flca.mda.codegen.helpers.* " %> [2]
```

[1] It start with a “<% jet” directive, followed by the package and class name.

This package and classname define where the output java file will be generated. Under the project jet settings, you define the root folder for the generated jet files like this:

right-click the project → Properties → Jet settings



If the “Source container” is called “src-jet-generated”, then the above jet file will result in a java file:

src-jet-generated\backend\EntityModel.java

[2] With the “imports” tag one can import as many packages as needed.

In the above example, the template does not do very much, let's fill it up.

```
<%@ jet package="backend" class="Demo"
imports="java.util.* " %>
<% Date today = new Date(); %>
Today <%=today%> I create my first jet template.
```

This will generate a java src-jet-generated\backend\Demo.java, that looks like:

```
package backend;
import java.util.*;
public class Demo {
    protected final String TEXT_1 = "Today ";
    protected final String TEXT_2 = " I create my first jet template.";

    public String generate(Object argument) {
        final StringBuffer stringBuffer = new StringBuffer();
        Date today = new Date();
        stringBuffer.append(TEXT_1);
        stringBuffer.append(today);
        stringBuffer.append(TEXT_2);
        return stringBuffer.toString();
    }
}
```

As you can see, the syntax inside the jet file, is similar to the jsp syntax. Between the "<% %>" you can put java command, and with "<%= %>" you output a variable.

Note instead of :

```
<% Date today = new Date(); %>
Today <%=today%> I create my first jet template.
```

You could also have used:

```
Today <%=new Date()%> I create my first jet template.
```

To output something conditional and/or repetitive the following constructs can be used:

```
<% if (acondition) { %>
Write this
<% } else { %?
Write that
<% } %>
```

```
<% for (String s : myapi.getList()) { %>
the value of s is <%=s%>
<% } %>
```

It is also possible to include another jet file like this:

```
<%@ include file="../Header.jetinc"%>
```

A Jet template is triggered via the generate method, that has the signature:  
`generate(Object argument)`

This argument often contains important information about what needs to be generated, and this is also true for the easyMda generator. I will come to that later.

This is about all you need to know about the Jet engine to create a jet template. What's much more important is what api you can use to generate any code you want.

### Note

But before you start creating templates, the by far most important task should already be finished by yourself! That is that you have a good working example(s) in place, of the code that you want to generate. This part is not only the most important but also the most time consuming. Because probably, if you want to generate source code, you want to have high quality code with logging documentation, well organized etc. One nice thing I discovered, is that the better the code is re-factored, the easier it is to generate! For example it is much easier to generate code of loosely coupled classes that only interact via messages, than to generate code for spaghetti like coupled classes. Because with the latter you will see that inside the template you will (also) end up with many if-then-else constructs. Even if the good re-factored code-base contains (many) more (small) classes, it is much faster to create corresponding templates for these, than to create less, but much more complex template classes for spaghetti like code base.

### ***Jet argument***

All jet enabled files, are translated into a java file that contains a single method **generate** that has the following signature:

`String generate(Object argument);`

The easyMda generator makes use of this by always passing a `JetArgument` as the required argument for all templates.

The `JetArgument` has a number of valuable properties:

```
Object getElement();
Class<?> getElementClass();
ITemplate getTemplate();
String getSourceCode();
```

- `getElement()`  
returns the instance of the current model class. We can use this to find out what the classname should be etc, but we can also use this object cast this instance to an interface that it implements and use that interface. This feature is used for example in `IWizard.jet` template.
- `getElementClass()`  
This returns the `Class` of the current model. This method is especially important, when the current model is an interface. In that case the generator is obviously not able to create an instance, and `getElement()` will return null.  
In general you don't have to worry about when to use `getElement` or `getElementClass`, because in most if not all cases, you know upfront in the template if you are dealing with a class or interface, because that is something you specify when you set-up the templates.
- `getTemplate()`  
This returns the current `ITemplate` being used.
- `getSourceCode()`  
This will return the original source code of the current model class or interface. Currently the only two use-cases that make use of this data are
  - to copy the original java doc
  - to copy the original method argument names, instead of generating argument names like `arg0`, `arg1` etc.



In addition, the `JetArgument` has a static method:

```
public static JetArgument getCurrent()
```

that returns the current active `ITemplate` (the generator runs in a single thread). This is particular handy inside some template api, that need to know about the current active template for example. You will see some examples in the chapters below.

## ***Common Api***

Theoretically you have enough tools in hand with the Jet engine. You can import the entire java stack, and hence you can do anything you want. But that is similar to building a modern application without any framework or standard library. Although many of you love that idea, unfortunately it is not realistic. To be productive, you have to rely on (many) frameworks and api. The same is true for easyMda. To be productive and create new templates fast and easy, you should use the the api that is provided with easyMda.

But before I explain the api more in detail, first a few words about the overall architecture of easyMda.

In order to generate a set of java file(s) from input java model file(s), a number of components are used each with its own role. At the start you have the jet engine, that generated a java file from a jet file, but strictly spoken this is not part of the plug-in. Then there is the actual generator. This component takes care of selecting the model class(es) and/or packages, displaying and handling the wizard, instantiating the jet java file and triggering the generate method etc.

Once this generate method is triggered the third component plays a role, and that is the api. This can be the common api, or cartridge specific api. The generator does depends on the common api, but not the other way around. The generator should not depend in any way on the specific cartridge api and the cartridge itself. The specific cartridge api may or may not rely on the common api. The cartridges that are currently available all have some specific

api that depends on the common api.

See:



From this it is clear, that for the jet template developer, the common api should be your best friend. So that is the one we explore at first more in detail.

### ***The Common api***

As stated before, the most important asset you need before you can build a successful template, is a good working example. Once you have a good working example, the 'only' thing you have to do, is to replace a number of tokens by dynamic counterparts. As said before, you can do a lot with the Jet and java language itself, but the common api makes a lot of tasks much easier. The common api has a number of utility classes, more or less in order of importance:

- **TypeUtils**  
Contains methods about a Java type in the broadest sense
- **NameUtils**  
This has methods that primarily deal with name and string handling.

- `TemplateUtils`  
Has methods that deal with an `ITemplate`
- `ImportUtils`  
Helps with organizing the import statements
- `InterfaceUtils`  
Similar to `TypeUtils` but this time dedicated to interfaces and methods
- `AppUtils`  
Contains methods dealing with overall application set-up
- `BufferUtils`  
To help with inserting blocks of text at a specific location
- `PostprocessorUtils`  
To trigger some special postprocessing

While building easyMda, I was much inspired by Taylor MDA, and those who know Taylor, recognize many method names. An important difference however, is that most utilities from above, have regular method instead of static methods. So you have to instantiate these before you can use it, example:

```
<% TypeUtils tu = new TypeUtils()% %>
:
<% for (Class c : tu.getAllFields()) { %>
```

I opted for this approach instead of using static method for two reasons:

- Once the utility is instantiated, the code becomes more compact
- You can build your own api on top of the common api by extending the common utilities.

In the following chapters the most important api routines will be described in detail. For a complete overview, you should examine the javadoc.

## ***How to replace that classname and package***

Some of the most obvious tokens that should be replaced, are the classname and package. In the simplest case, replacing the class can be as simple as this:

```
<% JetArgument jetatg = (JetArgument) argument; %>
<% Class thisclass = jetarg.getElementClass(); %>
:
public class <%=thisclass.getSimpleName()%> {
```

This works well if this class is for example the entity class itself. But it already becomes more tricky if we want to generate the corresponding Dao class. In that case you could use:

```
public class <%=thisclass.getSimpleName()%>Dao {
```

But both examples did not reveal yet how the corresponding package should be described. Also, the latter example is perfectly valid if the Dao class looks like this: <Entity>Dao. But how do you know? And what about the package that you should import if you want to use for example a mock object. How do you know what the package for this mock object will be? Ultimately you know, because you look at how the classname and package are defined inside the class that defines the IRegisterTemplates. This class will be described much more in detail later, but for now it is enough to know that that this 'register-templates' class defines in detail what the final classname and package will be.

A few simplified pseudo-code examples explain how this is done. Suppose that register-templates class has the following two definitions:

```
Entity class:
class = ${CLASSNAME}
package = ${PACKAGE}
```

```
Entity Mock object
class = ${CLASSNAME}Mock
package = ${PACKAGE}.mock
```

then for the model entity: org.abc.custinfo.Customer  
the resulting entity resp mock object will become:

```
org.abc.custinfo.Customer  
org.abc.custinfo.mock.CustomerMock
```

So returning to the question, how do know you what classname or package you should use? The answer is: you should ask the template, via the `TemplateUtils`. So instead of writing template code that looks like:

```
public class <%=currClass.getSimpleName()%>Mock
```

you should write something like this:

```
<% TemplateUtils tplu = new TemplateUtils(); %>  
:  
<% String mock = tplu.getClassName(currClass, Tid.ENTITY MOCK.name()); %>  
public class <%=mock%>
```

The template utilities method `getClassName()` returns the output classname defined by the given class and the name of a template definition. There is a similar method to obtain the package.

```
<% String pck = tplu.getPackage(currClass, Tid.ENTITY MOCK.name()); %>
```

In the examples above the current class is being used by the generator. This is often the case. Many api methods therefore have overloaded methods that don't require a class as parameter, instead the current active model class is being used. This is possible because (as said before), the input `JetArgument`, also has a static method to get the current active `JetArgument` and via that the current active class can be obtained. So the shorter variants look like:

```
<% String mock = tplu.getClassName(Tid.ENTITY MOCK.name()); %>  
<% String pck = tplu.getPackage(Tid.ENTITY MOCK.name()); %>
```

Both variants of the template utilities method: `getClassName()` are very heavily used. Instead the `getPackage()` methods are hardly used. That is because inside the `getClassName()` methods it will take care of the necessary import automatically. More on this topic in the next

chapter.

Both the `getClassName()` and `getPackage()` are overloaded even further. They accept as the last parameter, either an `ITemplate` object, or a `String`.

An `ITemplate` object can be obtained with: `getTemplate(String)` for example:

```
<% ITemplate t1 = tplu.getTemplate(Tid.ENTITY MOCK.name()); %>
```

and this can be used instead of the above examples like this:

```
<% String mock = tplu.getClassName(t1); %>
```

In the examples above the **name** of the template is an important argument, but instead of using hard-coded names, you should use similar code like:

```
Tid.ENTITY MOCK.name()
```

The `Tid` (mnemonic for Template identifier) is an Enum inside the webapp cartridge. But because the common api is not aware any cartridge template that makes use of the common api, it can not the `Tid` as input parameter, so therefore not the perfect but almost perfect, strong typing interface, is used.

#### Note:

In almost all jet templates, you need the name of the current class and current package. The easiest to obtain these two, is via two methods inside `NameUtils`:

```
<% String classname = nu.getCurrentClassname(); %>  
<% String pck = nu.getCurrentPackage(); %>
```

### ***How to manage those import statements***

In code generation, one of the more difficult tasks is to generate all the correct import statements. Not only you should take care that all imports are generated at the top, but also while for example generating a service method, you suddenly have to take care also of the import statement for the corresponding return type and parameter argument(s). One

simple approach to deal with this problem, is to simply import all possible packages using wild cards. This may generate valid code, but you then end up with ugly code (like for example code generated by JAXB), and violates the principle that the generated code should be same (or better) than if you would have written it yourself.

The `common-api` is designed as such that generating the correct import statements is as simple as possible. This is primarily achieved taking care of the import statements inside the utility methods when that is applicable. We saw already one example of this automatic import handling in the previous paragraph.

In all Jet templates that should generate a Java class, you should use something like this at the top of the Jet template:

```
package <%=pck%>;

<%ImportUtils impu = new ImportUtils();
  StringBuffer importStringBuffer = stringBuffer;
  int importInsertionPoint = stringBuffer.length();
  impu.addCompilationUnitImports(stringBuffer.toString());
%>
```

And this at the very bottom of the Jet template:

```
<% importStringBuffer.insert(importInsertionPoint, impu.computeSortedImports());%>
```

What happens is the following. Just below the package statement, all the import statements should be generated. This is achieved by the `ImportUtils addCompilationUnitImports()` method that adds some kind of bookmark at this point. At the end, the import statements that are accumulated are written at this bookmark.

One can force 'hard-coded' import statements like this:

```
<% impu.addImport("com.google.inject.Inject");%>
```

Many methods in `common api` will take care of the imports automatically. For example:

```
<% String typ = tu.getTypeName(clazz); %>
<% String returntyp = iu.getReturn(method); %>
```

Both commands will return the simple classname, and it will also import the given class!

A special method that can be used to automatically import the corresponding class is the `use()` method in `NameUtils`. This method has the following signatures:

- `public String use(Class<?> aClass)`
- `public String use(String aFullyQualifiedClassname)`

This will return the simple classname and take care of the corresponding import.  
The following 2 examples will result in identical generated code:

code fragment 1

```
<% impu.addImport("java.util.List");%>
<% impu.addImport("java.util.ArrayList");%>
:
List<String> mylist =
    new ArrayList<String>();
```

code fragment 2

```
<%=nu.use(List.class)%><String> mylist =
    new <%=nu.use("java.lang.ArrayList")%>();
```

Note that in the latter code fragment, I used `use()` with both signatures.

Obviously the first code fragment is easier to read. The biggest advantage of the latter approach however, is that this can be used inside an if-then-else construct or a potentially empty for-loop.

## ***Generate lines of code at specific location with BufferUtils***

Normally inside the jet template, the order of the output lines will correspond with the jet template itself. In the previous chapter we saw an exception to this rule, where all import statement are grouped together and placed just below the 'package' statement.



Because managing imports is so common, a special utilities class `ImportUtils` exists, that was explained in the previous chapter.

In addition there is another utilities class `BufferUtils` that is a more generic implementation of `ImportUtils`.

It has the following static methods:

- `String initBookmark(String aName)`
- `void append(String aName, String aText)`
- `void write(String aName, StringBuffer aTargetBuffer)`

Let's look at an example, how to generate a number of google guice `'@Inject'` statement that should be grouped just below the class definition.

```
Public Class <%=classname%> {  
<% String INJECT="inject"; %> (1)  
<%=BufferUtils.initBookmark(INJECT)%> (2)  
:  
<% BufferUtils.append(INJECT,"\t@Inject " + srv + " " + srvprop + ";");%> (3)  
:  
<% BufferUtils.write("inject", stringBuffer); %> (4)
```

- (1) In order to avoid typo's we create a String constant first.
- (2) The first `BufferUtils` command should be this one. Here we define the insertion point for this group of output lines, and give this bookmark a name.
- (3) After we created a bookmark, we can add as many lines we want. Note duplicate lines will be skipped.
- (4) At the very end of the jet-template, we must instruct `BufferUtils` to output the collection lines at the specified bookmark location.

Note the last parameter in this method is the `StringBuffer`, where to insert the lines. In general you should use the `StringBuffer` used by the jet engine itself. And this is defined by the commands that we saw before:

```
<%ImportUtils impu = new ImportUtils();  
StringBuffer importStringBuffer = stringBuffer;  
int importInsertionPoint = stringBuffer.length();
```

```
impu.addCompilationUnitImports(stringBuffer.toString());%>
```

So in this case it is: "stringbuffer"

## ***How to generate JavaDoc***

Java is nothing special, so you can simply generate documentation like this:

fragment from jet template-branch

```
/* Method : <%=methodName%>  
   Return type : <%=returnType%>  
   :  
*/
```

To document service methods, there is a special api inside `InterfaceUtils` called `getDocumentation(Method)`. This api will take into account the literal documentation that has been added in the original model file.

## ***How to keep manual code, after regenerating an output file***

A general issue with code generation tools, is that it is sometimes troublesome to align regenerated code with existing file(s) that meanwhile have been modified by the developer(s). In the user's guide, there is an entire chapter about this topic.

One point mentioned in this chapter is that easyMda has the ability to generate so-called "free-block". These free-block appear as comments inside the generated file, and the developer has the option to add his/her own code between the start and finish tags of these free blocks.

Depending on the target language the output will be in a different format. Currently the following two are supported:

- `<!-- xxx -->`  
for xml
- `/* xxx */`  
For all other languages

The following example show how this 'free-blocks' api is used:

The easiest way to create a is like this:

```
<%= SimpleMerge.freeBlock("Add_your_own_code") %>
```

This will result in :

```
/* freeblock_start: Add_your_own_code */  
/* freeblock_end: Add_your_own_code */
```

Note that the string 'Add\_your\_own\_code' should be unique if you intent to more then one free-blocks inside a template.

You can explicitly add a 'start' and 'end' freeblock like this. This has the advantage that in between you can put (default) code. For example:

```
<%= SimpleMerge.freeBlockStart("Add_your_own_code") %>  
    some generated code that may be replaced by the developer.  
<%= SimpleMerge.freeBlockEnd("Add_your_own_code") %>
```

For java files, the generator can also use the excellent JavaMerge component that is part Eclipse EMF tools-suite. The cartridge developer must specify for each template what 'merge-strategy' should be used.

## ***How to retrieve data from other resources***

In most templates, all the code to be generated can be deducted from the current model. But there are use-cases where the model project alone can not provide all the necessary information. For example inside the java models you don't explicitly define what should be the context-root of a webapp, or what should the the Google app-engine key etc.

Another use-case is that as a jet developer you may want to alter the generate-code, based on some criteria that the user should provide.

All these use-cases could be solved with the available easyMda machinery that we encountered so far: inheritance and annotations, but this is overkill for the the above use-cases. Instead easyMda provides an additional api that can be used here.

### **Interactive properties & ini files**

An important feature while developing a templates cartridge, is that the cartridge developer can register any number of so-called `SubsValue` item(s). A `SubsValue` is basically of 'substitute-from → substitute-to' key/value pair, with some additional properties to control its presentation on the wizard.

Two very important `SubsValue` items are created implicitly by the generator engine for each file it generates, these are:

- `CLASSNAME`
- `PACKAGE`

As you may expect, these values correspond with respectively the classname and package of the class being processed. These names are frequently used to define what will be the target class. See for example the following code snippet to register a specific template:

```
r.setPackage("<%=PACKAGE%>.util");  
r.setClassname("<%=CLASSNAME%>Mock");
```

So given a model class :	<code>flca.demo.crm.customer.Customer.class</code>
And Base-package :	<code>flca.demo.crm</code> <sup>4</sup>
And App-package :	<code>sitepen.crm</code> <sup>5</sup>
Then the result target will :	<code>sitepen.crm.customer.util.CustomerMock.class</code>

The `SubsValue` items are created at two places:

- Implicitly by the engine, See explanation above.
- Explicitly inside the class that implements `IRegisterTemplates` (in the next chapter more on this topic), in the method `getSubstitutes()`

The 'toValue' is resolved in a hierarchical way at two places:

1. From the ini-file `<model-project>/easymda.ini`

Note: Initially this ini file will not exist (you are the one that created the model-

---

4 This is the package in the Model project that is the base for all model clases.

5 This is the value that you filli-in in the last Gui page.

project!) but it will be created after the first run of the easyMda engine.

2. Interactively on the last (of three) wizard pages.

After this page these values are stored in the ini-file from above:

<model-project>/easymda.ini.

Properties found at the top (say 1) may be overwritten by `SubsValue` items found below.

To use these `SubsValue` items inside the jet-template, there is `api` inside `Nameutils`:

- `String getSubsValue(String aKey)`

The `SubsValue` class, is similar to a `Map<String, String>` that means that you should cast the 'subsTo' `String` value to another format like `int` or `boolean`, if that is what you want. Note it is possible to present a `SubsValue` item as a checkbox, depending on the select state, the value "true" or "false" will be set.

Example:

```
<appengine-web-app xmlns="http://appengine.google.com/ns/1.0">
  <application><%=nu.getSubsValue(app.engine.name)%></application>
  <version><%=nu.getSubsValue(app.engine.version)%></version>
```

```
<% if ("true".equals(nu.getSubsValue("use.jdo")) { %>
  write jdo code
<% } else { %>
  write hibernate code
<% } %>
```

## Ini-files

Another way to obtain other resources, is via ini-files. But as we will see, the implementation and therefore the applicable use-cases differ from using `SubsValue` items.

One of the (many) things the generator does after the user hits the [Generate code ...] from the context menu, is to scan a number of locations for ini-files, in an hierarchical way. For each ini-file it finds a (singleton) `Map<String, IniFileSection>` will be populated. The `String` in this map is the section-name (hence this should be unique over all inifiles). The `IniFileSection` is another map in this case a simple `Map<String, String>` like a `Properties`. All the files will be

scanned in the following order. Ini-files found more to the bottom may overwrite previously defined values.

The following locations will be scanned:

1. Eclipse easyMda plugin directory
2. Cartridge(s)
3. Model project

The engine will use brute-force to look for ini-files, it will simply scan all directories under the location mentioned above.

The ini-file format is slightly different from a regular ini-file, in order to allow multi-line values. This is best explained with a few examples:

First a 'normal' ini-file

```
[ AAA ]
key = avalue
another.key = another value
[BBB]
#not.now = this line uncommented
```

Now ini-files with multi-lines. These are depicted with “{{” and “}}” tag.

First a valid multi-line ini-file:

```
[ CCC ]
lines = {{
  This is line 1
  And line 2
}}
another.key = single line value
```

And now some examples of bad multi-line ini-file

```
[ DDD ]
bad.lines = {{ wrong, a start of multi-lines should end with only "= {{"

[ EEE ]
bad.lines = {{
```

ok the start was good  
but the end should be indicated with a newline with only the "}}" chars  
So this is not good }}

To use key/value from an ini-file is easy, with the following method inside NameUtils:

```
<%= nu.substitute("aSectionName", "aKey") %>
```

Note that you don't have to give the name of the ini-file! Like the key/values inside `SubsValue` items described above, the key/value pairs inside the ini-files are all string, so you need to cast the value if you need an int or boolean value.

Both the `SubsValue` as the ini-files, have the same purpose, so what are the pro's and con's and what should be used:

What	Pro's	Con's
SubsValue	Are interactive, via the wizard.	Because these are interactive, you should limit its number !
ini-files	The values can prepared, in an hierarchical way. No practical limit	These are not interactive, so the end-user has no control.

With this in mind, you should limit the number of `SubsValue` items to only those that you typically want to change just before you generate the code. Note that at the next run the last values being used, are now presented as default values.

Because there is no practical limit to the number of ini-files, and because it has sections, ini-files may play an important role in simplifying jet template code, or your api-code by replacing otherwise lengthy switch or if-then-else statement with a single call to an ini-file.

A good example can be found in several service jet templates, that deal with generating the typical crud code (retrieve, findAll, save, remove). See:

```
<% for (CrudMethod cm : iu.getAllCrudMedhods()) { %>
```

```
<% Properties props = iu.getSubsWithsForCrudOper(cm); %>
<%= tu.getCrudSnippet(cm, "java.crud.serviceImpl", props) %>
<% } %>
```

In this example the pattern used, consist of three parts:

1: In the jet-template itself, all crud code is generated via just one (!) command:

```
<%= tu.getCrudSnippet(cm, "java.crud.serviceImpl", props) %>
```

2 : The special utility: `Properties props = iu.getSubsWithsForCrudOper(cm);`

Generates Properties object with a number from-to pairs that will be used to substitute values inside ini-file section items.

3: The sections mentioned above, for this example consult the file: `crud.ini` that contain sections like:

```
[ java.crud.extServiceIntf ]
save = {{
    public <%=entity%> save<%=entity%>(<%=entity%> aValue <%=c%><%=contextArg%>);
}}

retrieve = {{
    public <%=entity%> retrieve<%=entity%>(<%=pkType%> aPk <%=c%><%=contextArg%>) ;
}}

find_all = {{
    public Collection<<%=entity%>> findAll<%=entity%>(<%=contextArg%>);
}}

remove = {{
    public void remove<%=entity%>(<%=pkType%> aPk <%=c%><%=contextArg%>);
}}
```

The idea is that the “divide-and-conquer pattern” is exploited as much as possible. The `iu.getSubsWithsForCrudOper()` should generate properties like:

```
entity = Customer
pkType = Long
:
```

And the `tu.getSubsWithsForCrudOper(Method, String)` is responsible for getting the



corresponding section based on the current (crud) method.

### ***How to organize your Jet files.***

In Java it is standard practise to organize the Java classes in such a way that these are easy to maintain. This can be achieved by putting repetitive code in a separate file(s) in order to avoid duplicate code and to split up large files into smaller files.

Although it is little bit more difficult with Jet files, the same principals described above can be achieved with Jet files as well. Therefore two techniques are available:

- using the Jet include statement.
- Using the TypeUtils include method.

The Jet include statement is used in almost all examples at the top of the file to file to set-up a number of common variables, for example:

```
<%@ include file="dojo-intf-init.jetinc" %>
```

This include is performed at 'compile-time', hence the effect is identical as if the entire include file is pasted into to master file.

The TypeUtils include statement is almost identical to the include statement from above, but now the include is only executed at run time.

For example see file: dojo/Services.jet :

```
<% for (CrudMethod cm : iu.getAllCrudMethods()) { %>  
<%= tu.include(DojoCrudServices.class, cm) %>  
<% } %>
```

The include method has the following signature:

```
public String include(Class<?> aSnippetClass, Object ... aArguments )
```

Hence the first argument is Jet class, and a additional number of Object arguments.

The advantage of this runtime approach instead of the dynamic approach is that runtime Jet class is more 'self-contained' and can also be jUnit tested separately.

## Major common api Utils

In this chapter the more important api will be explained per utilities class. Where necessary I will provide some feedback plus examples. For a complete set of examples, see the cartridge templates projects under the samples folder.

## AppUtils

AppUtils is a small utility that has the following methods:

- `String getApplicationName()`
- `String getApplicationPackage()`

## TypeUtils

The TestUtils api class contains methods that can be used primarily to obtain information about a class, or its properties. It is therefore heavily used.

In this paragraph the important and frequently used methods will be described.

In order to generate for example an entity class (an entity class, is a class that can be persisted) you must first obtain information about all the fields in the model class, and depending on the type different code should be generated. A collection for example inside an entity class, differs in a number of ways from a java primitive property.

The following methods are available to retrieve the properties of a class.

- `List<Field> getAllFields(Class aSource)`  
This will return all fields.
- `List<Field> getAllSimpleFields(Class<?> aSourceClass)`  
A simple field, is primitive a java type like Integer, Date or Decimal, but not an Enum, a collection or another entity class (the latter is an association or relation)
- `List<Field> getAllRelations(Class<?> aSourceClass)`  
returns all one2many and many2one relations
- `List<Field> getAllRelations(Class<?> aSourceClass)`
- `List<Field> getAllOneToManyFields(Class<?> aSourceClass)`
- `List<Field> getAllEnums(Class<?> aSourceClass)`

### Note

All the lists from can be empty, but never null.

### Note

In a future version, all of the above methods may be overloaded with an empty argument list, so that the same method can be used without an input source class. In that case the current model class will be used.

Some examples how these methods can be used:

```
<% for (Field fld : tu.getAllFields(element.getClass())) { //forA %>
<% String fldname = nu.uncapSafeName(fld); %>
    protected <%=tu.getImportedType(fld)%> <%=fldname%>;
<% } //forA %>
```

In this example we simple generate properties for all the fields in the class.

All the getXxx() methods from above, return a list of Field(s). Not surprisingly because under the covers the following standard java reflection method is used:

```
Field fields[] = aSourceClass.getDeclaredFields();
```

Once you have the corresponding Field object, the TypeUtils contains a number of methods that can provide more detailed info about this Field object:

- `String getTypeName(Field aField)`  
This return the class or or primitive of this Field as a String. It will also take care of the corresponding import statement if needed. If the return type is a generic type, the correct corresponding string will be returned.

For example the following model class, with the properties:

```
String name;
int count;
BigDecimal salary;
org.abc.MaritalStatusEnum maritalStatus;
```

```
org.abc.Address address;  
List<org.abc.Telecom> telecoms;
```

The following jet template fragment :

```
<% for (Field fld : tu.getAllFields(element.getClass())) { //forA %>  
    type is <%=tu.getTypeName(fld)%>;  
<% } //forA %>
```

will result in:

```
import java.math.BigDecimal;  
import org.abc.Address;  
import org.abc.MaritalStatusEnum;  
import org.abc.Telecom;
```

```
type is String  
type is int  
type is BigDecimal  
type is MaritalStatusEnum  
type is Address  
type is List<Telecom>
```

If you want to generate for getters and setters, than you need to format the name like in this example:

```
//--- getters and setters  
<% for (Field fld : tu.getAllFields(element.getClass())) { %>  
<% String fldname = nu.uncapName(fld); %>  
  
    public <%=tu.getTypeName(fld)%> get<%=nu.capName(fldname)%>() {  
        return this.<%=fldname%>;  
    }  
    public void set<%=nu.capName(fldname)%>(<%=tu.getTypeName(fld)%> aValue) {  
        this.<%=fldname%> = aValue;  
    }  
<% } //for loop %>
```

The TypeUtils also has a number of methods, that have a field as input argument, that tell what kind of field this is:

- **boolean** `isSimpleField(Field aField)`  
All except composite field
- **boolean** `isComposite(Field aField)`  
A composite is one2many or many2one relation
- **boolean** `isManyToOneField(Field aField)`
- **boolean** `isOneToManyField(Field aField)`
- **boolean** `isCollection(Field aField)`
- **boolean** `isPrimitive(Field aField)`  
true if it is an: int, long, boolean etc type.  
The `isXxx()` methods below return false if it is a primitive() !
- **boolean** `isEnum(Field aField)`
- **boolean** `isBooleanType(Class<?> type)`
- **boolean** `isStringType(Class<?> type)`
- **boolean** `isDate(Class<?> type)`
- **boolean** `isTimestamp(Class<?> type)`
- **boolean** `isTime(Class<?> type)`
- **boolean** `isBoolean(Class<?> type)`
- **boolean** `isInteger(Class<?> type)`
- **boolean** `isDouble(Class<?> type)`
- **boolean** `isLong(Class<?> type)`

We already looked at `getTypeName(Field)` the method is overloaded and comes in three flavours:

- `String getTypeName(Field aField)`
- `String getTypeName(Class<?> aClass)`
- `String getTypeName(Class<?> aClass, Class<?> aGenericType)`
- `Class getType(Field aField)`
- `Class getType(Field aField)`

The purpose of the second method: `getTypeName(Class<?> aClass)` why returning the type of the class, when we already know the class? For two reasons:

- This method also takes care of the import
- If the class is a generic one, the return string will be something like:  
`Set<Type>`

The third method offers the same functionality see:

```
<%=tu.getTypeName(Integer.class)%>
<%=tu.getTypeName(List.class)%>
<%=tu.getTypeName(List.class, String.class)%>
```

results in:

```
import java.util.List;
```

```
Integer
```

```
List<??
```

```
List<String>
```

More or less similar methods are:

- `Class<?> getGenericType(Field aField)`
- `String getGenericTypeName(Field aField)`

The first method returns a `Class` instead of a `String` like the similar methods.

### Methods that deal with inheritance

To find out if a class extends a type or implements an interface:

- `Class<?>[] getAllSuperTypes(Class<?> aClass)`

This is not null array the given class extends plus all the interfaces it implements.

To find out if the class extends or implements a particular class use:

- **boolean** `hasType(Class<?> aSourceClass, Class<?> aTestForClass)`

### Note

Use the standard java reflection methods to retrieve the supertype or interfaces:

- `Class<?> aClass.getSupertype()`
- `Class<?>[] aClass.getInterfaces()`

Special variations on the same theme are the following two methods:<sup>6</sup>

- `String getExtendsAndImplements(Class aClass, Class aDefaultImplements[])`
- `String getExtendsAndImplements(Class aClass, String aDefaultImplements[])`

---

<sup>6</sup> In a future version these methods will be moved to the dedicated webapp api, because these are not generic methods.

This returns a formatted ready to use string, see:

```
public class <%=classname%> <%=tu.getExtendsAndImplements(element.getClass(), new String[]
{"com.flca.frw.shared.IEntity", "java.io.Serializable"})%>
```

gives:

```
public class Customer implements IEntity, Serializable
```

### Methods to handle the entity's primary key

All entity classes require a primary key. For jpa entities, these are defined by the `@Id` annotation. The user may or may not provide an explicit primary key. When the user did **not** provide an explicit primary key, the template may generate one. This can be done with the following methods:

- `boolean hasExplicitId(Class<?>)`
- `boolean hasExplicitId()`  
Overloaded to use the current active class, the same applies for the methods below.
- `Class<?> getPkType(Class<?> aClass)`
- `Class<?> getPkType()`
- `Class<?> getPkName(Class<?> aClass)`
- `Class<?> getPkName()`
- `Field getPkField(Class<?> aClass)`
- `Field getPkField()`
- `String generateIdField()`
- `String generateIdGetterSetter()`

With the `hasExplicitId()` method you can find out if the user provided an explicit primary key or not. If so, then the `getPkType()` and `getPkName()` can be used to retrieve the class and name respectively. The last three methods can be used to generate a primary when the user did not provide one, and to generate the corresponding getters and setters.

Examples:

Model-1, with an explicit primary key

```
public class A1 implements IEntity {
    String name;
```

```
@Id
String id;
:
```

Model-2, without an explicit primary key

```
public class A2 implement IEntityType {
String name;
:
```

Jet template:

```
<% if (!tu.hasExplicitId()) {%>
<%= tu.generateIdField() %>
<% } %>
:
<%= tu.generateIdGetterSetter() %>
```

results in:

```
public class A1 {
String name;
@Id;
Long id;

public String getId() {
return id;
}
public void setId(String value) {
id = value;
}
:
respectively:
```

```
public class A2 {
String name;
@Id;
Long pk;

public Long getPk() {
return pk;
}
public void setPk(Long value) {
```



```
    pk = value;
  }
:
```

## Annotations

A very important cornerstone of the easyMda generator, is the ability to use annotations, in a type safe way, with fine grained control, to alter the behaviour of a class, interface, property and/or method.

TypeUtils has the following method related with annotation.

- **Annotation** `getAnnotation(Field aField, Class<?> aAnnotation)`  
This returns an Annotation give its corresponding class.
- **boolean** `hasAnnotation(Field aField, Class<?> aAnnotation)`  
To find if this field has a particular annotation
- **String** `getAnnotations(Field aField)`  
This method returns a formatted String that can be used as-is like this:  

```
<%= tu.getAnnotations(field) %>
private <%=fldtype%> <%=field.getName()%>;
```

### Note

To get all annotations that belong to a field, use:  
`Annotations annots[] = field.getAnnotations();`

## IAnnotationsParser

In the jet templates in the webapp cartridge, the `tu.getAnnotation()` is frequently used. The generated code is often very specific to the webapp cartridge, yet the generic (and not overridden) `tu.getAnnotation()` is used, how is this possible?

The answer is that the `tu.getAnnotation()` will delegate this call to `AnnotationsHelper` and this class has the ability to register so-called annotation hooks. Via this method:

```
public static void registerParser(IAnnotationParser aParser)
```

IAnnotationParser has the following interface:

```
void parseAnnotation(Field aField, StringBuffer aBuffer);
void parseAnnotation(Method aMethod, StringBuffer aBuffer);
void parseAnnotation(Class<?> aClass, StringBuffer aBuffer);
```

## NameUtils

The second utilities class that will be described is the NameUtils. This is an utility class, that with name handling plus some additional methods.

The following method's all return a formatted String:

- String capName(String name)
- String capName(Field field)
- String capName(Object element)
- String uncapName(String name) resp (Field) (Object)
- String nonPlural(String)  
replaces "ies" a the with "y" and "s" at the end with ""

### Examples

```
<%= nu.capName("abc")%>
<%= nu.uncapName("abc")%>
<%= nu.nonPlural("abc")%>
<%= nu.nonPlural("companies")%>
<%= nu.nonPlural("orders")%>
```

results in

```
Abc
abc
abc
company
order
```

We already looked at:

- String use(Class)
- String use(String)

These are short-cut's to return the simple name, and also take care of the corresponding import. Example:

```
<%=nu.use(List.class)%><String> mylist =
    new <%=nu.use("java.lang.ArrayList")%>();
```

Results in:

```
import java.util.ArrayList;
import java.util.List;
```

```
List myList =
    new ArrayList();
```

The following method:

- `String getSimpleName(String aFullQualifiedClassname)`

is similar to `use(String aFullQualifiedClassname)` expect the `getSimpleName()` will **not** add the corresponding import.

And we have seen the following methods:

- `String getCurrentClassname()`
- `String getCurrentPackage()`

These are very often used short-cut's to retrieve the classname or respectively the package of the current active class or interface.

Example:

```
<% String classname = nu.getCurrentClassname(); %>
<% String pck = nu.getCurrentPackage(); %>
:
package <%=pck%>;
:
public class <%=classname%> {
:
}
```

## InterfaceUtils

What `TypeUtils` is for `Classes`, is `InterfaceUtils` for interfaces.

It contains the following method to retrieve the methods of the interface:

- `List<Method> getMethods()`  
This return all relevant method of the current active interface. Trivial methods like `toString()`, `equals()` are skipped. The return may be empty but not null.

Example:

```
<% for (Method m : iu.getMethods()) {
    :
<% } %>
```

The following method all have in common that they act on the given method:

- `int` `getParameterCount(Method aMethod)`
- `String` `getParameterName(Method aMethod, int aIndex)`
- `Class<?>` `getParameterType(Method aMethod, int aIndex)`
- `String` `getReturn(Method aMethod)`
- `String` `getThrows(Method aMethod)`
- `Class<?>[]` `getExceptions(Method aMethod)`
- `String` `getArguments(Method aMethod)`  
This returns a formatted string with argument-types plus argument names.
- `String` `getDocumentation(Method aMethod)`

Examples:

```
<%for (Method m : iu.getMethods()) { //forA %>
<%String args = iu.getArguments(m);%>
<%String params = iu.getParameters(m);%>
<%String returnType = iu.getReturn(m);%>
<%=iu.getDocumentation(m)%>
    public <%=iu.getReturn(m)%> <%=m.getName()%>(<%=params%>) <%=iu.getThrows(m)%>
    {
        try {
            ctx = makeFlcaCtx(context, "<%=m.getName()%>");
<%if ("void".equals(returnType)) { //ifA %>
            service.<%=m.getName()%>(<%=args%>);
<%} else { %>
            <%=returnType%> result = service.<%=m.getName()%>(<%=args%>);
<%} //ifA %>
<%if ("void".equals(returnType)) { //ifB %>
            return;
<%} else { %>
            return result;
<%} //ifB %>
<% if (iu.hasThrowsException(m)) { //ifC %>
<% for (Class<?> excClz : iu.getExceptions(m)) { %>
            } catch (<%=excClz.getSimpleName()%> e) {
                handleError(e, ctx);
                throw e;
<% } %>
<% } else { %>
            } catch (Throwable t) {
                handleError(t, ctx);
                throw new RuntimeException(t);
```

```
<% } //ifC %>
    } finally {
        cleanupCtx(ctx);
    }
}
<%} // forA %>
```

## PostProcessorUtils

With `PostProcessorUtils` you have the possibility to run a post-processor after the code has been generated. To use this facility you should add something like this in your jet-template.

```
<% PostProcessorUtils.add(MySpecialPostprocess.class); %>
```

The `MySpecialPostprocess` is a class that must implement the `IflcaPostProcess` interface. This interface has one method:

```
String parse(String aInput, ITemplate aTemplate);
```

You can add as many post-processors if you want. The order though is important. Each subsequent post-processor will get output of the previous post-processor.

## Other Utils

In previous paragraphs, we already walked through:

- `ImportUtils`
- `TemplateUtils`
- `BufferUtils`

## Part III, Create your own cartridge project

In this chapter I will explain the process of creating and deploying your own templates cartridge. I wanted to make this process as simple as possible.

In general the entire process is not more than these steps:

- Create a standard (jet enabled) Java project.
- Optionally create your own api, to support your new jet templates.
- Create your jet templates, as explained in the previous chapter.
- Register your jet templates.
- Deploy this project into the easyMda plugin

For a simple cartridge like the demo 'wizard' cartridge, the entire process can be accomplished within a few hours.

The steps will now be explained more in detail.

### ***Create jet enabled project***

The first task to accomplish is to create a new project that will host the new cartridge templates.

This new project is a standard Eclipse Java project, that should be Jet enabled. First create a normal project with the following commands:

File → New → Java Project

And give a name, and either use the default location or pick your own location in the usual way.

This project depends on a few jar file that can be added in the usual way via:

Right click project → Properties → Build path → Libraries

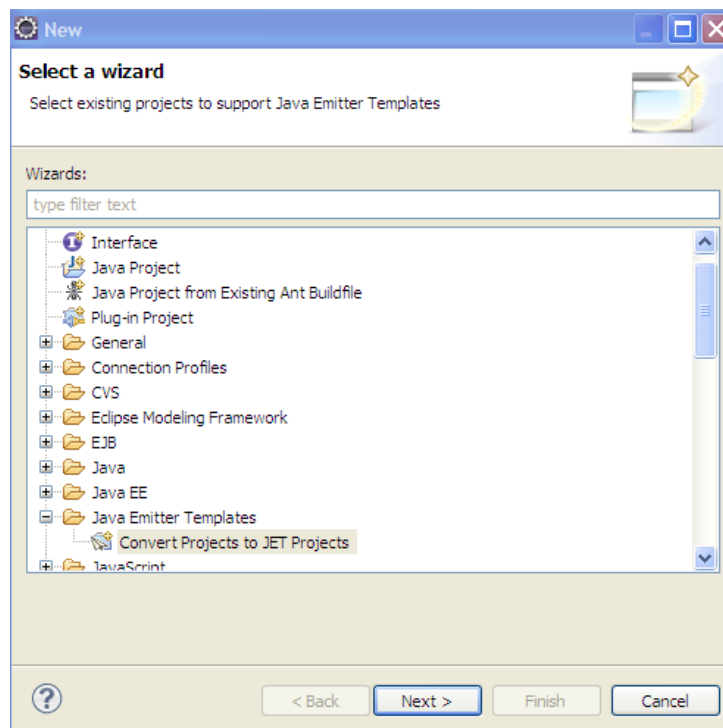
And now add the following two jar file : `easymda.commonapi-1.8.jar`

That can be found in the `<easymda-dir>/lib` directory.

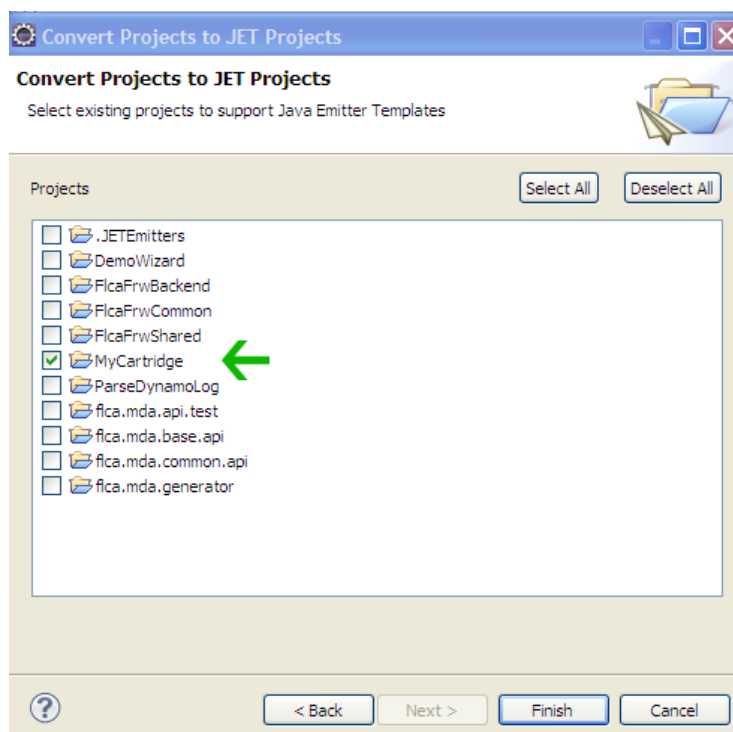
And finally make this project Jet enabled with the following command:

File → New → Other → Java Emitter Templates

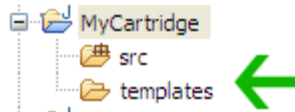
This brings up the following wizard:



After this select your new project you just created, and hit [Finish]



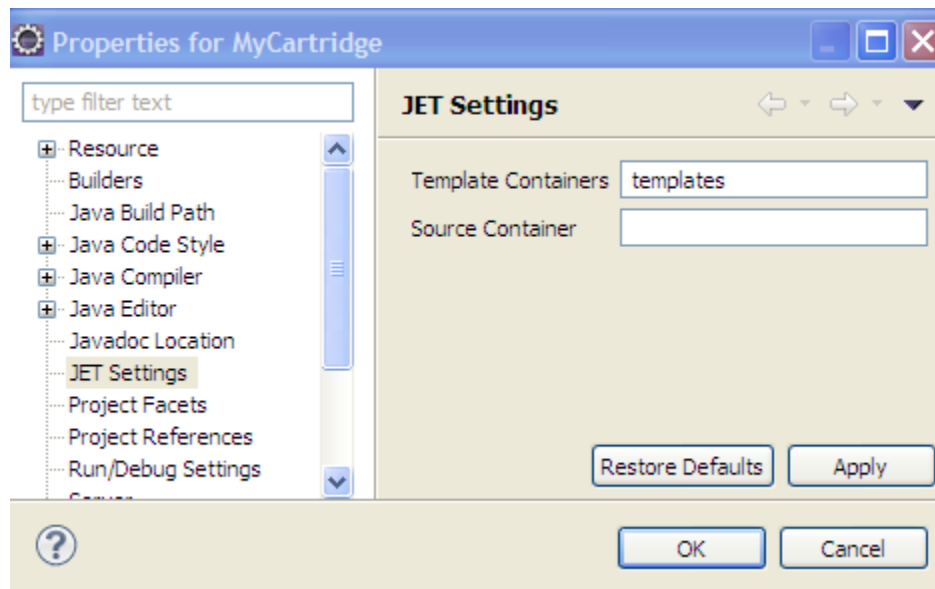
After this command, a new folder is created in the project that you just selected, named: templates.



Under this folder, you should create your new jet templates.

Although not required, I advise you to modify the jet-settings so that the compiled jet files are put in a dedicated java source folder. This can be achieved like this:

- Create a new source folder, with a name like **src-jet-generated** (or something similar) with:  
right-click → Properties → Jet settings  
this pop's up :





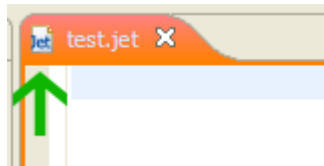
And now enter the new folder name from above in the 'Source container' input box.

To see if the set-up from above works as expected, create a new file under the **templates** folder with a name like: test.jet with:

new → file → test.jet

**Note:**

Apparently there is a bug in the Jet editor (org.elcipse.jet.editor v1.1.0) . This Jet editor puts the compiled java file under all source folders, instead of the src-jet-generated folder only. You can see that the test.jet file from above is opened with special jet editor if it shows the following icon:



If that is the case, close the file, and open it again with a text editor like this:  
right-click file → open with → Text editor.

The dedicated Jet editor has some important advantages compared to the text editor (like intellisense + auto-completion), you could as an alternative modify the setting of the 'other' source folder(s) like src to exclude the packages that you intend to use with the jet templates. Via Properties → Java build path → Source (tab) → select src folder → Edit .. → [Next] → Add Exclusion pattern like 'mypackage/\*\*'

<TODO url + instructions for better jet editor>

Paste the following content into this jet file:

```
<%@ jet package="mypackage"
```

```
imports="java.util.* flca.mda.common.api.util.* com.flca.mda.codegen.helpers.*
mda.type.* "
class="MyClass" %>
Hello new Jet template.
The time is now: <%=new Date()%>
```

When you save this file, you may notice that a new java under the src-jet-generated folder is generated named: src-jet-generated/mypackage/MyClass.java

Now make a typo in jet file like changing

```
The time is now: <%=new Date()%>
```

into:

```
The time is now: <%=new Date %>
```

Save the file, again, and now you will see 2 errors in the Problems tab.

This behavior that the java file is automatically generated when you save a jet-file, and this java file is compiled immediately afterwards (if build-automatically is enabled), is a very powerful feature. Once you fixed all the typo's and removed all the errors, you know for sure that from the jet syntax point everything is ok. This in contrast to many other generators out there, where syntax error are only revealed at run time.

### *Create your api, to support the new jet-templates [optionally]*

Depending on the requirements for your new cartridge you may want to create your own api because the utilities inside the common-api and/or other existing cartridges are not sufficient.

There are two kinds of api :

1. One that is targeted for the end-user that creates the model project(s)
2. And one to support the jet developer (this will be often the same developer)

#### **Api for model-project end user**

The api targeted for model project end user, consists of two types:

- interfaces
- annotations

You may look at [com.flca.cartridge.wizzard](#) project for some examples. Two examples:

```
mda/type/IWizzard.java:
package mda.type;
public interface IWizzard extends IBaseType
{
    String getTitle();
    String getImage();
    List<IWizzardPage> getPages();
}
```

```
mda/annotation.wizzard/PageField.java:
package mda.annotation.wizzard;
import omitted
@Target({METHOD, FIELD})
@Retention(RUNTIME)

public @interface PageField {
    public String label();
}
```

### Api to support jet developer.

This is a regular java file(s), like any other java api. You may consult the project: [flca.mda.common.api](#) for many examples.

## Create your jet-templates

Now that your project is in place, you can start creating your jet templates. This part is already explained extensively in the previous chapter.

## Register your jet templates

The last piece of the puzzle before you can deploy and use your new cartridge is to register all your newly created jet templates.

In the same spirit of easyMda where Java is the model, this registration is done via a java class as well.

All easyMda template cartridges must have exactly one class file that implements `flca.mda.templates.IRegisterTemplates`

Before I delve into `IRegisterTemplates`, more in detail, I first need to explain the concept of: templates-tree, template-branches and templates.

Each jet-enabled java project with easyMda templates, is called a cartridge. Such a cartridge consists of a so-called templates-tree. A templates-tree has a name, and contains a `List<TemplatesBranch>`. Each `TemplatesBranch` has a name, and contains a `List<ITemplate>`. A cartridge has a nested structure, in order to make it possible to combine a group of related templates, that logically belong to each other, but are otherwise independent. This concept can best be explained with an example.

The `flca.mda.templates.webapp` is a cartridge that eventually will consists of the following branches:

- jee backend
- google app-engine backend
- flex frontend
- ZK frontend
- GWT frontend
- HTML5 frontend
- ?more?

The idea is that the same model project can be generated with each template-branches from above.

On the other hand there is also the `flca.mda.templates.wizzard` cartridge, that only has 1 cartridge and just 4 templates.

The interface `IregisterTemplates` has the following operations:

Operation	description
<code>String getName()</code>	this return the name of this tree of template branches
<code>String getDescription()</code>	this returns the description
<code>Collection&lt;TemplatesBranch&gt; getTemplateBranches()</code>	This should return all the templates that you want to publish for this cartridge

<code>Collection&lt;SubsValue&gt; getSubstituteValues()</code>	This should return values that can be used by <a href="#">the codegenerator to substitute the corr values.</a>
<code>void beforeOnce()</code>	This is an optional hook that can be used to initialize the api once before the generator starts
<code>void beforeEach()</code>	This is an optional hook that can be used to initialize the api before each generation

The first two operations are self explanatory. Note that `getName()` should be an unique over all available template branches.

Via the method `Collection<SubsValue> getSubstituteValues()`, the cartridge developer can define a number of `SubsValue` (s) that can used to provide some additional information to the generator. A nice feature of a `SubsValue` is that can appear on the wizard, so that the end user make a last minute decision here. See the paragraph: "interactive properties & ini-files" for much info on this topic.

The methods: `beforeOnce()` and `beforeEach()` give the cartridge developer the possibility to hook into the engine before engine start, resp before each class/template combination is generated.

A typical use-case is to register your specific helper class that implements:

`IannotationParser` like this:

```
@Override
public void beforeOnce() {
    AnnotationsHelper.registerParser(new AnnotationParser());
}
```

This dynamic registration, makes it possible to use inside the jet-file the standard method inside the common-api `TypeUtils`:

```
<%= tu.getAnnotations(field) %>
```

I kept the most important operation, and the one should return non-empty collection:

`Collection<TemplatesBranch> getTemplateBranches()` for now, because it requires more attention.

The most import property of a `TemplatesBranch` is the `List<ITemplate>`. Hence in the end

the end comes down to creating instances of `ITemplate`.

`ITemplate` has the following operation:

Operation	description
<code>String getName()</code>	Name (or Id) of this template. This name should be unique for all templates!
<code>String getGeneratorFqn()</code>	Fully qualified classname of the corresponding jet generator. This class will be instantiated by the engine, and then its <code>generate(JetArgument)</code> will be called.
<code>String getJetPath()</code>	File of the corresponding jetFile. This is for debugging purposes only. <sup>7</sup>
<code>String getPackage()</code>	The generated package (and hence output directory) that corresponds with this template. This will often have a signature like: <code>\${PACKAGE}</code> of <code>\${PACKAGE}.xxx</code> Where <code>\${PACKAGE}</code> will be substituted automatically with package of the model class or interface being processed.
<code>String getClassName()</code>	The generated classname (and hence the output filename) that corresponds with this template. This will often have a signature like: <code>\${CLASSNAME}</code> of <code>\${CLASSNAME}.xxx</code> Where <code>\${CLASSNAME}</code> will be substituted automatically with classname of the model class or interface being processed. (see also note below)
<code>String getFileExtension()</code>	The extension of the file, default = <code>".java"</code>
<code>String getTargetDir()</code>	The targetdir. This is also often defined with an implicit substitute value like: <code>\${Backend}</code> or <code>\${Frontend}</code> etc.
<b>boolean</b> <code>appliesTo(Class&lt;?&gt; aSourceClass)</code>	Indicates on what implemented (or extended) interface, this template will be applied. A null or empty array indicates that this template can be applied any class. The input is an array, so several types can be given.
<code>TemplateMergeStrategy getMergeStrategy()</code>	Indicates merge strategy should be applied.
<code>Collection&lt;SunsValue&gt; getAskForSubstitutes()</code>	Explicitly defined SubsValue(s). This may return null or empty collection
<code>int getRank()</code>	The order in which the templates should be applied 1 first, 2 second etc. Default 5 ?.

<sup>7</sup> I contract to for example TaylorMDA that relies on the source of the jet template.

	Note in the current version this value is not used yet, but will be in later version, because in some rare cases, the order in which files are generated may be important.
--	--

**Note:**

the constants `CLASSNAME` and `PACKAGE` are defined in `com.flca.mda.codegen.CodegenConstants`.

**Note:**

In general each template will generate a corresponding output file which is defined by a combination of :

`getTargetDir()+getPackage()+getClassname()+getFileExtension()`.

In some rare cases though, you may want the output the content to a particular outputfile.

An example is the generated Guice module file. This can be achieved with the “#” sign inside the `getClassname()` definition.

For example the Guice module is defined like this:

```
${APP_NAME}Module#INSERT_DAO_BINDS
```

In this definition, `APP_NAME` will be substituted with the value coming from the corresponding `SubsValue` and the generated content will be output into the (generated) module file, after the (comment) line that contains “`INSERT_DAO_BINDS`”

## How is the final filename generated

The final output filename is based on a number of variables. The important rule to remember is that the “base package” is replaced with the so called “application package”, if the latter one is not empty.

The base package is determined by all the model classes, by finding the package that is common to all project classes (and/or interfaces). Consider the following two model projects, that each contain the following classes:

**Project-1 :**

```
order.Order + order.OrderDetail
customer.Customer + customer.Address ...
```

## Project-2 :

org.abc.order.Order + org.abc.order.OrderDetail  
 org.abc.customer.Customer + org.abc.customer.Address ...

The base package for project-1 is null (because no corresponding package exists), and for project-2 it is "org.abc"

This results in:

class	base-pck	app-pck	result
order.OrderDetail	"	org.abc	org.abc.order.OrderDetail
org.abc.order.OrderDetail	org.abc	org.abc	org.abc.order.OrderDetail
order.OrderDetail	"	"	order.OrderDetail
org.abc.order.OrderDetail	org.abc	"	org.abc.order.OrderDetail
org.abc.order.OrderDetail	org.abc	com.zyz	com.xyz.order.OrderDetail

## Keep your templates manageable

Although not required whatsoever, I advise you to keep your code manageable by using techniques similar to ones uses in the supplied cartridges `com.flca.templates.webapp` and `com.flca.templates.wizzard`

In these projects are created classes "Tid", `JavaConstants`, `FlexConstants` and `ZkConstants`.

Note: Tid is a short-cut for `TemplateIdentifier`.

The Tid is an Enum that corresponds exactly with all the templates within this project. The trick is that the Tid Enum values are also used to set the template name. This has the big advantage one can refer to a template in an almost 'type-safe' manner, without otherwise possible typo's. For example:

```
<%String srvbase = tplu.getAppClassName(Tid.APP_EXT_SERVICE_BASE.name());%>
```

The `JavaConstants` (and `Flex/ZkConstants`) classes are there to organize the templates, and make it easy to create a new definition.

With these to helper classes, the class that implements `IRegisterTemplates` can now use a method like:



```
private TemplatesBranch makeBackendBranch() {
    TemplatesBranch result = new TemplatesBranch();
    result.setName("backend.jpa");
    result.setDescription("java backend based on JPA persistence");

    for (Tid tid : Tid.values()) {
        ITemplate template = makeTemplate(tid);
        result.addTemplate(template);
    }

    return result;
}
```

To return one or more template-branches.

### ***Deploy this project into the easyMda plugin***

The last task before you can use your new cartridge, is to deploy it into the easyMda plugin. This is easy as well. The only thing you have to do, is to add a classpath in the

```
<eclipse>/dropins/flca.easymda.generator_1.0.0/cartridges
```

folder. This classpath can either be:

- A jar file with the cartridge project.  
Such file can be created like any other jar file from a Java project, either via Eclipse with: right-click → Export → Jar file etc.  
Or via an Ant or Maven script.
- A xxx.link file that contains a path property that point to the bin directory of the cartridge project. For example:  
wizard.link that contains:  
path=/media/c/mydocs/robin/projects/flca.mda/flca.mda.templates.webapp/bin

Additional 3-party lib(s)

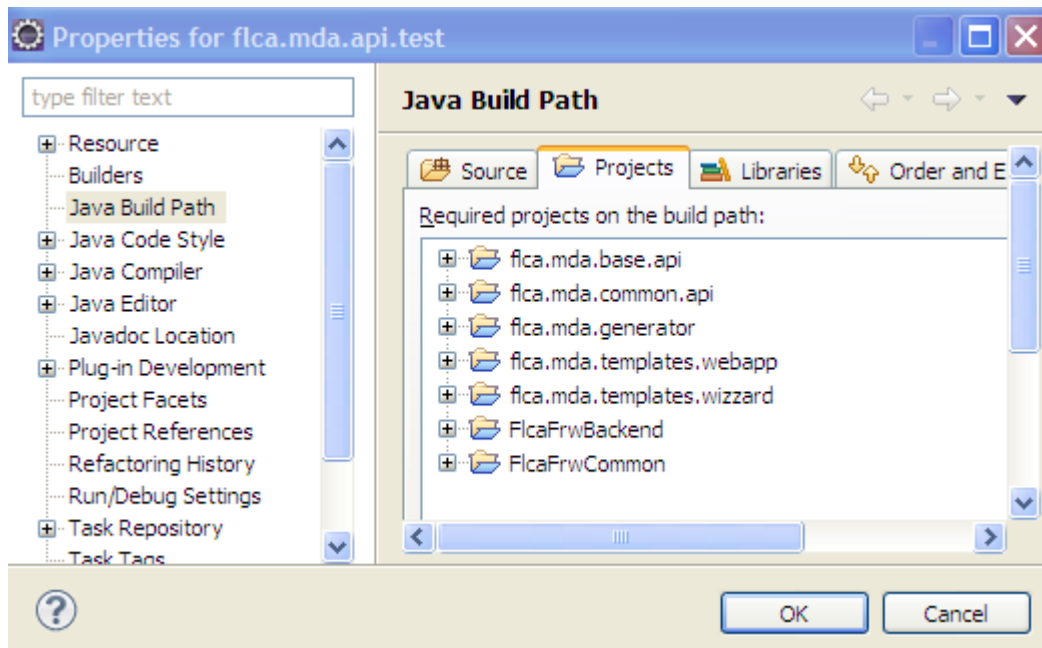
<TODO>

## Debugging your jet template(s) and api

A (IMO) important and very valuable feature that is not often found in other MDA tools, is the ability to generate-code for a particular model-project class / template combination via a junit tests. This is achieved by mocking the runtime 'environment' that is otherwise set-up by the plugin. This mocking up process is quite complex but hidden inside a junit base-class that can be simply extended. This junit base-class can be found in the the project:

`fica.mda.test.generate`

This test project is set-up a little bit different from other projects. Instead of putting a number jar files inside it java build-path, it has a number of project in its build-path. Like this:



This set-up has the advantage that when you set a breakpoint, you end up the original java file instead of the class file, and in most cases you can edit the core there and then.

To run a junit-test is easy: <TODO>

- Create a junit case that extends the base-class:  
`test.TestTemplatesBase`

- Provide the 'environment' settings by filling the names of the projects being used (these projects are normally filled by the plugin) something like this:

```
@BeforeClass
public static void beforeOnce()
{
    if (ShellUtils.isWindows()) {
        model_rootdir = "C:/mydocs/robin/com.flca.mda/flca.mda.templates.webapp";
        template_rootdir = "C:/mydocs/robin/com.flca.mda/flca.mda.templates.webapp";
        plugin_rootdir = "C:/mydocs/robin/com.flca.mda/FlcaMdaCodegen";
    } else {
        model_rootdir = "/media/c/mydocs/robin/com.flca.mda/flca.mda.templates.webapp";
        template_rootdir = "/media/c/mydocs/robin/com.flca.mda/flca.mda.templates.webapp";
        plugin_rootdir = "/media/c/mydocs/robin/com.flca.mda/flca.mda.generator";
    }

    TestTemplatesBase.beforeOnceBase();
}
```

- create the units, using the helper methods from the base class:  
testGenerateObject() and testGenerateInterface()  
examples:

```
@Test
public void testEntityTemplates()
{
    testGenerateObject(new A(), Tid.makeTemplate(Tid.ENTITY));
    :

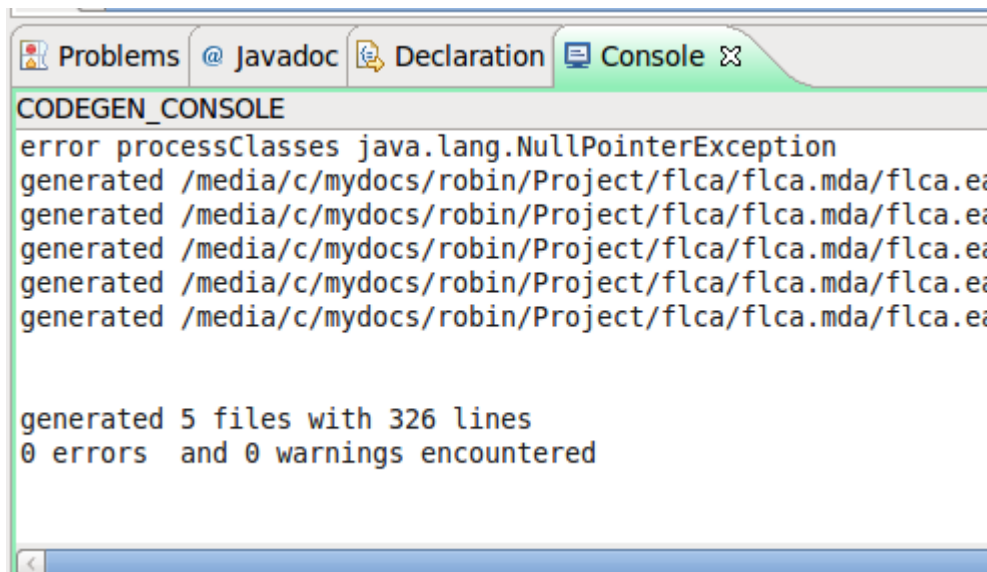
@Test @Ignore
public void testServiceTemplates()
{
    testGenerateInterface(TstService.class, Tid.makeTemplate(Tid.SERVICE));
    :
}
```

## Logging

EasyMda has a very simple logging framework<sup>8</sup>. Inside the engine itself `LogHelper` is being used, and within the api, the api developer can use `Logger` class that has the following static methods:

- `void info(String aMsg)`
- `void console(String aMsg)`
- `void warning(String aMsg)`
- `void debug(String aMsg)`
- `void error(String aMsg)`
- `void error(String aMsg, Throwable ex)`
- `boolean isDebugEnabled()`

All errors and messages sent to `Logger.console(msg)` will appear in the Eclipse console. Typical output may look like:



The screenshot shows the Eclipse IDE's Console window. The 'Console' tab is selected, showing output from the 'CODEGEN\_CONSOLE' plugin. The output consists of an error message followed by several 'generated' status messages and a summary line.

```
error processClasses java.lang.NullPointerException
generated /media/c/mydocs/robin/Project/flca/flca.mda/flca.ea
generated /media/c/mydocs/robin/Project/flca/flca.mda/flca.ea
generated /media/c/mydocs/robin/Project/flca/flca.mda/flca.ea
generated /media/c/mydocs/robin/Project/flca/flca.mda/flca.ea
generated /media/c/mydocs/robin/Project/flca/flca.mda/flca.ea

generated 5 files with 326 lines
0 errors and 0 warnings encountered
```

---

<sup>8</sup> I am still in doubt to use `log4j` or `slf4j` instead. It add's extra dependencies to the plugin and makes it larger, is that worthwhile?

All messages, except debug messages, unless debug messages are enabled (which is currently always true), will appear in the logfile:

`<current-model-project>/generate.log`

Prepare Eclipse Juno

help install-new-software

work with: Juno - <http://download.eclipse.org/releases/juno>

select: EMF - Eclipse Modeling Framework SDK