Maven and Eclipse usage on META

PURPOSE

This document discuses our best practices for using Maven as the primary project build tool along with Eclipse as an Integrated Development Environment.

BACKGROUND

Maven is the command line software build tool we use for building the ARRoW application for deployment. Eclipse[1] provides a source code development environment for editing and debugging the META application. Maven utilizes pom.xml files to specify the configuration of the application for assembly, while Eclipse utilizes .project, .classpath, and other files for specifying the configuration of the application for debugging. The difficulty is in keeping the Maven pom.xml and the Eclipse project, classpath and settings files in sync with each other.

We have decided that the pom.xml file is the authoritative basis for the project and the Eclipse files need to be generated from the pom.xml file. The pom.xml file is checked into svn, the Eclipse files are not. Our solution is based on using the maven-eclipse-plugin (v2.8), which is a Maven plugin that takes a pom.xml file and generates the relevant Eclipse project configuration files.

The maven-eclipse-plugin is documented at the Maven site[2]; the examples below are intended to extend that documentation with specific usage on ARRoW, and to inform developers on how to maintain pom files accordingly.

General usage

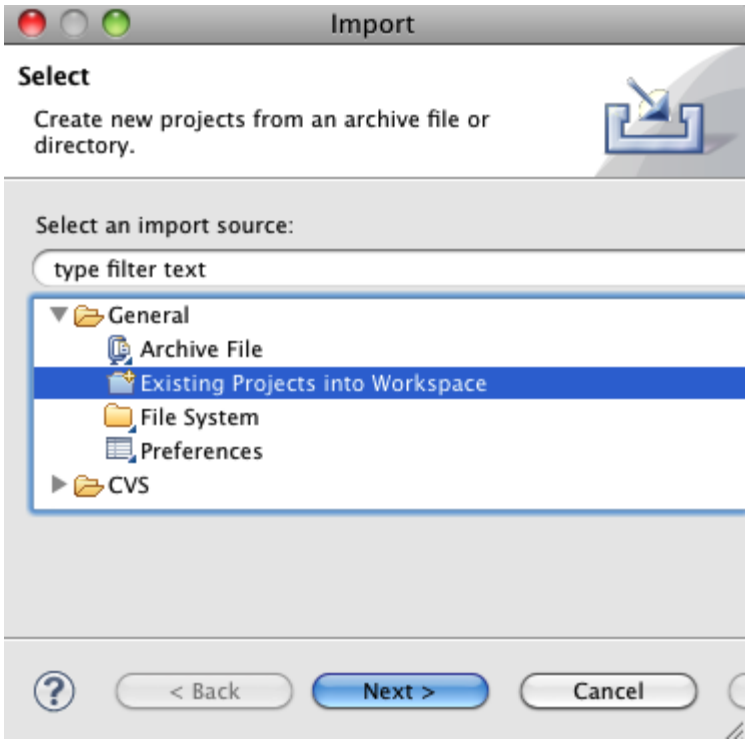
The simplest usage of the maven-eclipse-plugin is to go to the project directory of the project you would like to work on in Eclipse and run:

```
>mvn eclipse:clean eclipse:eclipse install
```

This command will remove previously generated Eclipse config files, generate new config files, and do a local install of the project. The project may now be imported to Eclipse using the Eclipse command: *Import>Existing Projects into Workspace*

[1] Spring Tool Suite , the recommended IDE for ARRoW work, is a customized version of Eclipse. http://www.springsource.com/developer/sts

[2] http://maven.apache.org/plugins/maven-eclipse-plugin/



arrow-mvn-all

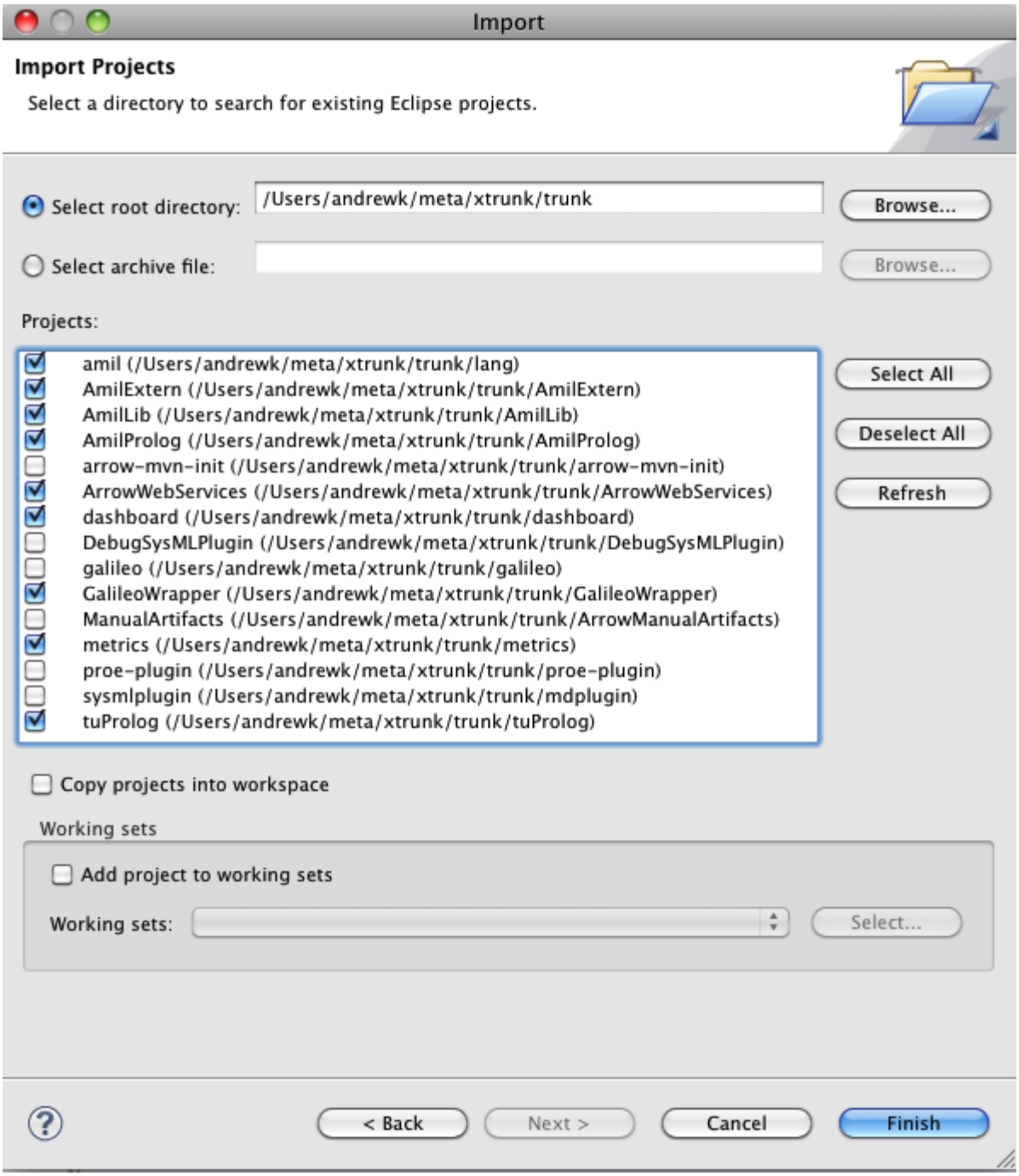
The arrow-mvn-all/pom.xml builds all of the modules needed to run ArrowWebServices. In order to work on all of these modules in Eclipse the configuration files need to be created from this top level directory.

From the arrow-mvn-all directory run,

```
>mvn eclipse:clean eclipse:eclipse install
```

A successful build will show that we think we are good as far as building from maven, and it will have installed the jars from any modules into your local Maven repository.

Import the project files into Eclipse. Use the *Import>General>Existing Projects into Workspace*. Select trunk as the root directory, then select all the modules listed in the arrow-mvn-all/pom.xml which at the time of writing are the following projects; ArrowWebServices; lang(amil); AmilExtern; AmilLib; GalileoWrapper; tuProlog; AmilProlog; metrics; dashboard.



ArrowWebServices

If the goal is to work only in ArrowWebServices, and not to also work in all of the additional modules on which ArrowWebServices depends, then run

```
>mvn eclipse:clean eclipse:eclipse install
```

From the ArrowWebServices directory, and import into Eclipse only that project.

Changes to pom.xml files

Any time a pom.xml file is changed, to add a dependency, to add a module, to change a version, or really any change at all it is best to re-run

```
>mvn eclipse:clean eclipse:eclipse install
```

Then you must refresh the affected Eclipse projects, or if in doubt just refresh them all.

Changes to Eclipse configuration

At some point you will want to make a change in your Eclipse configuration and then reflect that change in the pom.xml file. This is not an easy task. The Eclipse configuration files are not well documented, and each Eclipse plugin that has some feature you wish to use will store its configurations in its own files. But here is the general outline:

- Make changes to the Eclipse config
- Identify the configuration files that changed
- Modify the maven-eclipse-plugin build settings to generate the config files

Prior to making any changes to the Eclipse configuration, archive the files in the project directory

Annotations Example

The AmilExtern module needed to use some features of http://code.google.com/p/spi/ which in turn needed Eclipse annotation processing to be configure to use spi. The Eclipse configuration was done by:

[G]o to properties for the project, and navigate to Java Compiler/Annotation Processing. Check Enable project specific settings, Enable annotation processing, and Enable processing in editor. Then select the Factory Path dialog (under Annotation Processing), once again Enable project\ specific settings, and use Add JARs... to add a reference to the ManualArtifacts project jars/spi-full-2.4.jar. This will allow compilation of the @ProviderFor annotation.[…]

Once this was done, we were able to see new files: .factorypath; .settings/org.eclipse.jdt.apt.core.prefs; .settings/org.eclipse.jdt.core.prefs had been created.

The content of these files were used to configure the maven-eclipse-plugin in the pom.xml file.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-eclipse-plugin</artifactId>
    <version>2.8</version>
    <configuration>
        <downloadSources>true</downloadSources>
        <downloadJavadocs>false</downloadJavadocs>
        <wtpversion>2.0</wtpversion>
        <additionalBuildcommands>
            <buildCommand>
```

```
      <name>org.springframework.ide.eclipse.core.springbuilder</name>
                              </buildCommand>
                        </additionalBuildcommands>
                        <additionalProjectnatures>

      <projectnature>org.springframework.ide.eclipse.core.springnature
</projectnature>
                        </additionalProjectnatures>
                        <additionalConfig> <!--  Enable annotation
processing in Eclipse -->
                              <file>
                                    <name>.factorypath</name>
                                    <content><![CDATA[<factorypath>
                                      <factorypathentry
kind="VARJAR" id="M2_REPO/com/googlecode/spi/spi/2.4/spi-2.4.jar"
enabled="true" runInBatchMode="false"/>
                                                      </factorypath>

                                          ]]>
                                    </content>
                              </file>
                              <file>

      <name>.settings/org.eclipse.jdt.apt.core.prefs</name>
                                    <content><![CDATA[
eclipse.preferences.version=1
org.eclipse.jdt.apt.aptEnabled=true
org.eclipse.jdt.apt.genSrcDir=${jsr269.generated.dir}
org.eclipse.jdt.apt.reconcileEnabled=true
                                      ]]>
                                    </content>
                              </file>
                              <file>

      <name>.settings/org.eclipse.jdt.core.prefs</name>
                                    <content><![CDATA[
eclipse.preferences.version=1
org.eclipse.jdt.core.compiler.codegen.targetPlatform=1.6
org.eclipse.jdt.core.compiler.compliance=1.6
org.eclipse.jdt.core.compiler.processAnnotations=enabled
org.eclipse.jdt.core.compiler.source=1.6
                                          ]]>
                                    </content>
                              </file>
                        </additionalConfig>
                  </configuration>
            </plugin>
```

WTP, Spring Nature, etc

Not all configuration needs to be done as in the Annotations example above. The maven-eclipse-plugin has settings and facilities for managing Eclipse configurations. We use WTP and we enable Spring functionality in Eclipse as follows:

```
      <wtpversion>2.0</wtpversion>
         <additionalBuildcommands>
         <buildCommand>
      <name>org.springframework.ide.eclipse.core.springbuilder</name>
                                 </buildCommand>
      </additionalBuildcommands>
      <additionalProjectnatures>
      <projectnature>org.springframework.ide.eclipse.core.springnature
</projectnature></additionalProjectnatures>
```

Therefore, we recommend reviewing the capabilities of the maven-eclipse-plugin before relying on CDATA and creation of preference files.

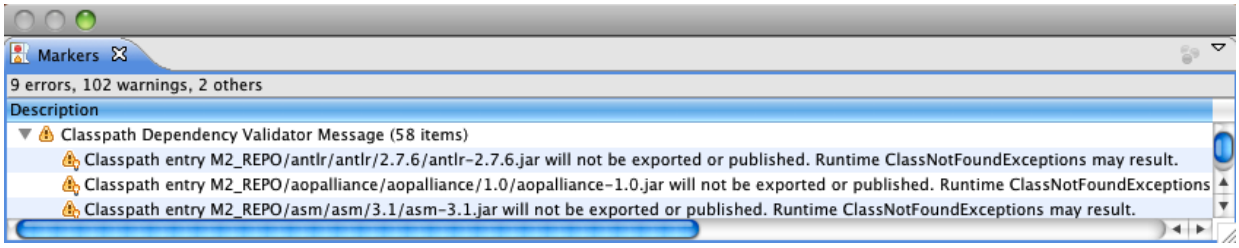
Issues to Consider

Inaccurate warning messages

There is a bug that causes the reporting of Classpath Dependency Validator Messages. These can be ignored.

http://maven.40175.n5.nabble.com/Created-MECLIPSE-565-Classpath-entries-to-be-marked-as-NOT-exported-td267021.html

"As a workaround these warnings can be filtered out by removing selection from "Classpath Dependency Validator Message" in Problems view filter configuration."



History

On previous BAE projects we had used both maven-eclipse-plugin and m2eclipse[3] (aka m2e), but as of maven-eclipse-plugin v2.8, m2e developers are explicit in saying the two are not compatible.[4]

[3] m2e is the same as m2eclipse originally developed by Sonatype, now given to Eclipse foundation.

[4] http://maven.40175.n5.nabble.com/Releasing-maven-eclipse-plugin-td211330.html

Alternatives considered

1) Use maven-eclipse-plugin

This is the method we have chosen, mainly because it allowed us to configure the use of Annotations in the AmilExtern project correctly in Eclipse.

2) Use the Eclipse plugin m2e

This is the method we used to use when it was compatible with the maven-eclipse-plugin. However, we frequently had odd behaviors when a project was more than a few modules as on DeepGreen. In tracking down the solution to the AmilExtern issue, we discovered that m2e is now explicit about being incompatible with maven-eclipse-plugin. There are many discussions about this we found via Google. Some large projects like Apache CXF give various advice;

- http://cxf.apache.org/setting-up-eclipse.html
- http://cxf.apache.org/connecting-maven-eclipse-checkstyle-and-pmd.html ,
- http://cxf.apache.org/cxf-m2eclipse.html

Finally, this page had what we considered to be a well reasoned discussion that simply put m2eclipse out of the running as a solution; http://groups.google.com/group/scala-ide-user/browse_thread/thread/33da4ca59183eb6e/c9a17c80dbe75cab?pli=1

3) Manage the pom.xml and Eclipse settings files separately and check in both to version control.

This could still be a reasonable option, but we would have to deal with checking in the files, and ideally have to add some test on checking that ensures the pom.xml files are always more recent than the Eclipse config files to demonstrate that no change could have been made in an Eclipse file and not reflected by the update to the pom file.

Working Notes

```
// none
```

NO_M2ECLIPSE_SUPPORT: Project files created with the maven-eclipse-plugin are not supported in M2Eclipse