Maven Tips

This page collects various things learned about Maven and tips on how we use it.

Parent Poms

These can serve as a common spot to put shared things inherited by child poms.
These can serve as a place to specify (in the <modules> element) children to recursively process.

- These two functions need not be the same POM, but are in our current implementation
- You can install the shared, parent POM without running the install on the children by using the parameter -N, e.g.
    - mvn -N install
      This will install the parent to your local repository (which you need to do if you change it) without running the install on the children.

Our use of parent poms

Version 2.3.0

Pom parent hierarchy:

```
maven-super-pom
  uimaj
    uimaj/eclipse-plugin-superpom
      (all the eclipse projects in the uima base)
    SandboxDistr
      SandboxDist/annotator-package
      (all the sandbox projects)
    uima-as
      (all the uima-as projects)
```

The parts common to a release are factored toward the top.

The factoring of resources and test resources in the build is such that if you specify any resource, you have to specify the ones that are the standard ones (src/main/resources and src/test/resources) if you want them included.

Parents, if edited, should be installed into your local repo - otherwise, the changes may not be picked up. Currently, it appears maven picks up the changes only for the immediate parent, not for grandparents (unless the grandparent is installed).

We do not use (at the moment) the Apache Super Pom (http://repo1.maven.org/maven2/org/apache/apache/6/apache-6.pom. One reason is that it includes the standard license and notice in the jars; we include custom license and notice files depending on the subproject, along with a Disclaimer file (while we're in incubation).

The mvn command

The mvn command documention appears when you type mvn -help. Besides the options, it takes 0 or more goals to run, and/or 0 or more life-cycle phases to run. If given a life cycle phase, such as "install", it runs all the previous phases in the life cycle, up to that point. If given a goal, such as "eclipse:eclipse", it may run life-cycle phases up to a specified point, depending on the goal implementation. For eclipse:eclipse, the documentation says it runs the phase "generate-resources" - meaning it runs all the life cycle phases up to that point.

If given multiple arguments, it does each one in sequence. e.g. `mvn clean install` or `mvn eclipse:clean eclipse:eclipse`

Useful mvn commands

mvn help:effective-pom - shows the fully inherited pom

mvn -N xxx - skip calling maven on submodules - useful when changing a parent and needing to install it.

Coding dependency scopes: compile vs. provided

compile:

- the dependency mechanism operates transitively, and dependencies of the dependent artifact are located and included, too.
- the eclipse:eclipse for Eclipse plug-in projects puts <link>s to the jars in the local repository into the .project file (if run locally; if eclipse:eclipse run from the parent POM, it acts like "provided" below – it puts the other plugin in as a "project" plugin dependency.

provided:

- the dependency mechanism doesn't fetch the dependents of the dependent artifact - the chain stops here
- the eclipse:eclipse for Eclipse plugin projects doesn't put any <link>s in the .project file. The .classpath file has an entry for the required plugins container, and the plugin container is set up to reference the other project directly

Other ways eclipse:eclipse generates project references versus jars-in-the-local-repo refs

You can run eclipse:eclipse on the parent of multiple projects. If you do this, dependencies between modules will be configured as direct project dependencies in Eclipse (unless useProjectReferences is set to false), instead of as refs to the jars in the local maven repository.

If you do this for plugin projects, it works the same way. To enable this, you have to use maven to build the uimaj-ep-runtime plugin. This build actually populates the .class(es) files for this project from the other projects, as it adds those other project's jars to its jar.

Other plugins that depend on the ep-runtime should specify a compile dependency on the ep-runtime.

The maven-eclipse-plugin is configured for pde(s) with <manifest>.ignore</manifest>

This gives the message when executing mvn eclipse:eclipse
`WARNING The references manifest file doesn't exist, plugin dependencies will not be updated: C:\a\Eclipse\3.3\apache\uimaj-ep-runtime\.ignore`
If you don't have this, eclipse:eclipse overwrites the manifest for the plugin, which loses all the plugin configuration information.

Passing parameters to Ant scripts

This can be done. In the <tasks> part, you must include an extra statement: `<property name="name-inside-ant" value="${name-inside-maven}"/>`

For the many cases where these names are the same, this looks kind of stupid, but seems to work.

For example, to set an ant value myValue from a Maven settings file, you write in the maven `antrun` configuration:

```
...  <tasks>
        ...  some ant code, perhaps
       <property name="myValue" value="${myValue}"/>
```

It seems the use of ${name} in a <property ...> statement as a value, if it isn't defined in ant, looks up the name in the maven environment. This is only done for the <property ... > element, though. So you need this dummy assignment, before first use.