chem-bla-ics

Jumbo 5.0 and the CDK

Egon Willighagen

Published December 10, 2005

Citation
Willighagen, E. (2005). Jumbo 5.0 and the CDK. *Chem-bla-ics*. https://doi.org/10.59350/y0mte-4ns18

Keywords
Cdk, Cml, Java

I reported earlier that the CDK has been updated in CVS to use CML from the new Jumbo 5.0. The transition actually involved a lot of changes in the CDK, some I would like to address in the following comments. One thing is that CML write support (not reading!) uses the new Jumbo library which requires Java 1.5. Thus, if Java 1.5 is not available, then CML writing should not be compiled. This is how this is done.

The JavaDoc

The CDK makes extensive use of JavaDoc taglets. CDK uses tags of type `@cdk.SOMETAG`. And an important tag in this case, is the `@cdk.require` tag, becuase it allows us to make the CDK build system aware that the class requires Java 5.0 to be compiled. Thus, we have for example this code in CVS, of which bits are:

```
/**
 * Serializes a SetOfMolecules or a Molecule object to CML 2 code.
 * Chemical Markup Language is an XML based file format {@cdk.cite PMR99}.
 * Output can be redirected to other Writer objects like StringWriter
 * and FileWriter.
 *
 * @cdk.module       libio-cml
 * @cdk.builddepends xom-1.0.jar
 * @cdk.depends      jumbo50.jar
 * @cdk.require      java1.5
 */
public class CMLWriter extends DefaultChemObjectWriter {
}
```

As probably is clear compiling this jars requires a two jars to be present, of which the `jumbo50.jar` itself is not required for compiling the class source code. It also shows the use of the `@cdk.require` tag.

The build.xml

Because the CDK still does not require Java 1.5, the CDK is supposed to be buildable with Java 1.4 (the oldest supported Java release). The Ant build.xml script is quite able to conditionally leave out compiling parts of the CDK, if configured correctly using proper JavaDoc tags, as explained earlier.

First, the build.xml checks what libraries are available for compiling certain parts of the CDK. For example, the build.xml code to check for Java 1.5 looks like:

```
 property="isJava15">
   string="${java.version}" substring="1.5"/>
```

chem-bla-ics

Run `ant info` to see what is being checked for, or look at the `build.xml` source code for the check target.

All compiling is done by the compile-module target, and there it in- and excludes bits of the CDK depending on the checked conditions:

```
srcdir="${build.src}" destdir="${build}" optimize="${optimization}"
      debug="${debug}" deprecation="${deprecation}">

  name="${src}/java1.4+.javafiles" if="isJava13"/>
  name="${src}/java1.4.javafiles" unless="isJava14"/>
  name="${src}/java1.5.javafiles" unless="isJava15"/>
  name="${src}/ant1.6.javafiles" unless="hasAnt16"/>
  name="${src}/r-project.javafiles" unless="rispresent"/>

  name="${src}/${module}.javafiles"/>
```

Keep in mind that the `*.javafiles` are created with JavaDoc based on the CDK JavaDoc tags mentioned earlier.

The build.xml 2

While the above mechanism has been present since for some time now, having jumbo50.jar in CVS made the situation a bit trickier: the `jumbo50.jar` uses the 49.0 class format used in Java 1.5, and cannot be processed by Java 1.4 systems. Since the classpath used when compiling CDK source code, is defined in configuration files for those modules in src/META-INF, the problem did not occur when compiling the modules. However, it did show an error in the `reallyRunDoclet` target today, when I was creating the `*.javafiles` with JavaDoc. The solution was trivial:

```
name="reallyRunDoclet" id="reallyRunDoclet"
 depends="compileDoclet" unless="dotjavafiles.uptodate">
  private="true"  maxmemory="128m">

      dir="${lib}">
        name="*.jar" />

        name="jumbo50.jar" unless="isJava15"/>

      dir="${lib}/libio">
        name="*.jar" />

      dir="${devellib}">
        name="*.jar" />
```

```
     name="net.sf.cdk.tools.MakeJavaFilesFilesDoclet"
      path="${doc}/javadoc"/>

     dir="${src}">
       name="org/openscience/cdk/**"/>
```

cdk.applications.FileConvertor

There is another area of interest: the `FileConvertor`, which is, sort of, CDK's OpenBabel's `babel` variant. The FileConvertor must be compiled in all cases, so we need to conditionally instantiate the `CMLWriter`, which is not really a problem. However, compiling the source code is more troublesome: the `CMLWriter` class must be loaded on runtime, and not occur hardcoded in the source code.

In the past I have solved this by using `.getInstance()` constructs, but the ChemObjectWriter interface does not define this functionality, so I decided to use the `java.lang.reflect` mechanism:

```
} else if (format.equalsIgnoreCase("CML")) {
  Class cmlWriterClass = this.getClass().getClassLoader().
    loadClass("org.opscience.cdk.io.CMLWriter");
  if (cmlWriterClass != null) {
    writer = (ChemObjectWriter)cmlWriterClass.newInstance();
  }
  Constructor constructor = writer.getClass().getConstructor(new Class[]
{Writer.class});
  writer = (ChemObjectWriter)constructor.newInstance(new Object[]
{fileWriter});
} else {
```

Now, this has been, by far, the longest blog item I have written so far. I hope it gave you good insight in some techniques CDK uses to deal with situations where functionality might, or might not, be present at build and at run time.

References