

# Customizing file IO

This article describes how file IO in CDK can be customized to do or do not certain things when files are read or written. This is very convenient for certain kinds of file formats. The possibilities will be exemplified using the command line utility `cdk-fileconvert` and Java source code.

by Egon Willighagen

An interesting feature of file IO in CDK is that it is customizable. Before I will give all the details, let's start with a simple example: creating a Gaussian input file for optimizing the structure of methane, and let's start with an XYZ file, that is, with 'methane.xyz':

```
5
methane
C 0.25700 -0.36300 0.00000
H 0.25700 0.72700 0.00000
H 0.77100 -0.72700 0.89000
H 0.77100 -0.72700 -0.89000
H -0.77100 -0.72700 0.00000
```

Later I'll explain how IO customization is done on a source code level, but I'll use the CDK **FileConverter** first now. In all command line options I will use the `cdk-fileconvert` shell wrapper which is equivalent to

```
java -cp cdk-all.jar \
  org.openscience.cdk.applications.FileConverter
```

The following command will convert the XYZ file into a Gaussian input file:

```
cdk-fileconvert -o gin methane.xyz
```

The three letter code `gin` indicates the **io.program.GaussianInputWriter** which writes the Gaussian input format. Use `cdk-fileconvert --help` to see what other output formats are supported. The output will look something like

```
# b3lyp/6-31g
```

Created with CDK (<http://cdk.sf.net/>)

```
0 1
C 0 0.257 -0.363 0.0
H 0 0.257 0.727 0.0
H 0 0.771 -0.727 0.89
H 0 0.771 -0.727 -0.89
H 0 -0.771 -0.727 0.0
```

The writer used the default IO options in the above example. So, the next step is to see which options the writer allows.

## IO options

To get a list of options for a certain IO class in CDK, use the `--listoptions`, or `-l`, argument. For the **GaussianInputWriter** this would be:

```
cdk-fileconvert -l \
  program.GaussianInputWriter
```

The output is a list of options supported by the IO class. At the time of writing, the class supported eight IO options: Basis, Method, Command, Comment, OpenShell, ProcessorCount, UseCheckPoint-File and Memory. All options have defaults, and the output given earlier is created using these.

There are two mechanism that allow overwriting the default IO settings. One uses a text interface and is started with:

```
cdk-fileconvert -q all -o gin methane.xyz
```

Alternatively, you can make a Java properties file, for example called 'custom.props', of which the content may look like:

```
Basis=6-31g*
Command=geometry optimization
Comment=Job started on Linux cluster \
  on 20041010.
ProcessorCount=5
```

You can use these setting with the command:

```
cdk-fileconvert -p custom.props -o gin \
  methane.xyz
```

The nice thing about the **FileConverter** is that it can convert a large set of files in one command. For example, try: `cdk-fileconvert -o gin *.xyz`.

## io.listener.ChemObjectIOListener

At the Java source code level the above is also possible. All CDK IO classes implement the **io.ChemObjectIO** interface, which has three methods of interest (in version 1.6 of the interface):

```
public IOSetting[] getIOSettings();
public void addChemObjectIOListener\
  (ChemObjectIOListener listener);
public void removeChemObjectIOListener\
  (ChemObjectIOListener listener);
```

The method `getIOSettings()` returns the settings available for the IO class. In the **GaussianInputWriter** there are eight settings, so it will return an array of length eight. Later I will discuss the **io.setting.IOSetting** class and will first discuss the **io.listener.ChemObjectIOListener**. Classes implementing this interface have to implement only one method, and how they do that it's up to the class:

```
public void processIOSettingQuestion\
  (IOSetting setting);
```

CDK contains three implementation of this interface: **TextGUIListener**, **SwingGUIListener** and **PropertiesListener**, all in the `io.listener` package. The first was used when we used the command `cdk-fileconvert -q`. The second is a Swing implementation with the same function and is used in JChemPaint (<http://jchempaint.sf.net>), see Fig. 1. The third implementation is the one which I will use in the following source code example.

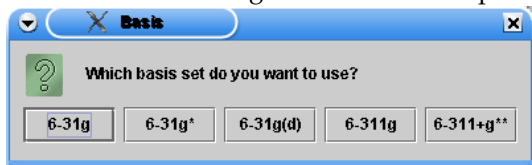


Figure 1: Customization question asked by the **SwingGUIListener**.

## PropertiesListener

If we want to automatize file conversion, it is not possible to use the interactive **TextGUIListener** and **SwingGUIListener** classes, but we need to use the **PropertiesListener** instead. This class is also used in the **FileConvertor** when the user uses a properties file with the `-p` option.

Consider the following source code:

```

1 import java.io.*;
2 import java.util.*;
3
4 import org.openscience.cdk.*;
5 import org.openscience.cdk.io.*;
6 import org.openscience.cdk.io.listener.*;
7 import org.openscience.cdk.io.program.*;
8 import org.openscience.cdk.io.setting.*;
9
10 public class Example {
11
12     public static void main(String[] args) {
13         // the custom settings
14         Properties customSettings =
15             new Properties();
16         customSettings.setProperty(
17             "Basis", "6-31g*");
18         customSettings.setProperty(
19             "Command",
20             "geometry optimization");
21         customSettings.setProperty(
22             "Comment",
23             "Job started on Linux cluster " +
24             "on 20041010.");
25         customSettings.setProperty(
26             "ProcessorCount", "5");
27
28         PropertiesListener listener =
29             new PropertiesListener(
30                 customSettings
31             );
32
33         try {
34             // create the writer

```

```

35         GaussianInputWriter writer =
36             new GaussianInputWriter(
37                 new FileWriter(
38                     new File("methane.gin")
39                 )
40             );
41         writer.addChemObjectIOListener(
42             listener
43         );
44
45         XYZReader reader = new XYZReader(
46             new FileReader(
47                 new File("methane.xyz")
48             )
49         );
50
51         // convert the file
52         ChemFile content =
53             (ChemFile)reader.read(
54                 new ChemFile()
55             );
56         Molecule molecule =
57             content.getChemSequence(0).
58             getChemModel(0).
59             getSetOfMolecules().
60             getMolecule(0);
61         writer.write(molecule);
62         writer.close();
63     } catch (Exception e) {
64         e.printStackTrace();
65     }
66 }
67
68 }

```

The first eight lines import the packages required for this example: some classes from the default Java libraries; the core CDK classes in the `org.openscience.cdk` package; and, of course, the classes in CDK's IO packages.

The **PropertiesListener** takes a **Properties** class as parameter in its constructor. Therefore, the properties for writing the Gaussian input file are defined first on lines 14 to 26. Then, on lines 28 to 31, the **PropertiesListener** can be instantiated.

On lines 33 to 65 the file conversion happens. First the output writer is created on lines 35 to 40, specified to write to the 'methane.gin' file.

The most important lines in this example are lines 41 to 43: on those lines the **PropertiesListener** is registered with the output writer. Only then will the output be customized with the settings given earlier in the method.

On lines 45 to 49 the input reader is constructed, and the lines 51 to 62 perform the format conversion. On line 63 all **Exceptions** are caught that might be thrown during the conversion process.

## Customizable IO classes

I promised to get back on the **IOSetting** class. If you check again the **ChemObjectIO** and **ChemOb-**

jectIOListener interfaces, this class has the function of containing information about the IO setting, and store the active value.

CDK currently has four **IOSetting** implementations: **BooleanIOSetting**, **IntegerIOSetting**, **StringIOSetting** and **OptionIOSetting**. The first three classes can only take specific settings: yes or no, integers, and **Strings** respectively.

The **OptionIOSetting** is a special class; besides one default, it contains a list of suggestions. The **ChemObjectIOListeners**, i.e. all current implemen-

tations, will offer this list of options to the user, as depicted in Fig. 1.

If you plan to add **IOSettings** to existing or new IO classes, please have a look at the source code of the **GaussianInputWriter** or the **CMLWriter**.

I hope you appreciate the simplicity of the whole architecture.

*Egon Willighagen*

*Radboud University Nijmegen, The Netherlands*

*e.willighagen@science.ru.nl*

# First steps in the implementation of a force field for the CDK package

Development of a 3D-structure model builder tool.

by *Christian Hoppe*

## Introduction

As mentioned in the CDK News 1.1[1] we recently started to work on a CDK implementation of a force field. Force fields are widely used in the area of chemoinformatics, e.g. to optimize the 3D geometry of a molecule, in conformational search or studies in molecular dynamics[2]. A force field of a molecule describes with a set of equations the potential energy surface of that molecule. Since those equations are not based on first principles empirically or quantenmechanically precalculated data is needed. This parameterisation limits the application of force fields and over the years several different sets of equations with corresponding data have been developed[2, 3]. In general one can differentiate these force fields in two classes, one for macromolecules and one for small organic molecules (usually less than 200 heavy atoms)[2]. We are aiming at the latter and plan to provide at least a couple of minimization algorithms which can be used in combination with a force field to optimize the 3D-structure of a molecule.

## 3D-Model builder

In the process of implementing a force field method for the CDK package, we started with the development of a tool for the generation of reasonable 3D starting structures. One of the most severe problems in the generation of 3D coordinates is the layout of rings and ring systems. Therefore we followed a knowledge-based approach in collecting and storing unique ring systems (ignoring different confor-

mations) to use them as templates in the 3D structure generation process[4]. This procedure differs from real 3D structure generating programs as e.g. CORINA[5], which stores for small ring systems conformational templates. We did not consider conformations at this step, because we only want to generate a reasonable starting structure for the force field. We downloaded a collection of small molecules as molfile from the nci databank (<http://cactus.nci.nih.gov/ncidb2/download.html>)[6]. To extract the molecule data stored in this file (249,071 3D-structures) the **IteratingMDLReader** from the CDK software package was used.

```
import org.openscience.cdk.io.iterator
    .IteratingMDLReader;

iteratingMdlReader = new IteratingMDLReader(
    new BufferedReader(new FileReader(nci_molecules))
);

Molecule molecule = null;
while (iteratingMdlReader.hasNext()) {
    molecule = (Molecule)imdl.next();
    // ...
}
```

To identify ring systems in a molecule one can use e.g. the **SSSRFinder** class. This class identifies the smallest set of smallest rings of a molecule. The **RingPartitioner** class can partition this sssRingSet into RingSets of connected rings which share at least an atom, a bond or three or more atoms with another ring in the RingSet. The container class **RingSet** is able to store, handle and manipulate rings and ring systems.

```
import org.openscience.cdk.RingSet;
import org.openscience.cdk.ringsearch.SSSRFinder;
import org.openscience.cdk.ringsearch
    .RingPartitioner;

SSSRFinder sssrf = new SSSRFinder();
```