Groovy Cheminformatics

with the Chemistry Development Kit

Ed. 1.4.1-0

Egon L. Willighagen PhD

Long time CDK developer

Further information: http://chem-bla-ics.blogspot.com
E.L. Willighagen <egon.willighagen+cdkbook@gmail.com>

Preface

This book is written to help people start developing cheminformatics software using the Chemistry Development Kit, also known as the CDK. But unlike past CDK documentation, I wanted to do something new. This book is that something new.

The book you now have in your hands is written for CDK 1.4.1. That means that all code snippets are compiled and tested against this particular version. That does not mean the scripts are useless for other CDK version. In fact, the CDK API does not change that often, and it is likely that most of these scripts work out of the box with other versions too, or at least from the same series. API changes do happen, of course, and the book has a chapter on migrating code from older CDK versions to the current version. But, the importance of all this is that all code compiles against this particular CDK version. It also means that all output shown in this book is actually automatically generated with the code shown. This is because each snippet in this book is actually part of a small Groovy script, BeanShell script, or Java Application, and when the PDF for this book was generated, all 94 scripts in the book have been compiled and executed. You will also observe that the book prepends each code snippet with a small orange bar, pointing to a file in the source code distribution which will be soon available from the book homepage. For now, please send me an email to get a copy of this distribution.

Now, the importance of this is that I want to be able to update the book with every CDK release. Not just every minor release, but also every bug fix release. The last should not really require me to change the code snippets, but it might change the output of the code here and there.

Another aspect of this book is also something new to me: the book is not Open. You are not allowed to copy it or parts of it; you are also not allowed to make derivatives. That is fairly new to me. The idea here is that this book provides support. None of the code examples in this book are hidden knowledge. Most of it is part of the public documentation already. Otherwise, most of the answers in this book show up in some form or another on the mailing list, in some blog, or otherwise. But this compilation will be closed. At least for now.

Finally, I think that the frequent editions that I plan to make is fairly novel as well. As I understood, my *publisher* Lulu.com makes it possible to make

an updated release whenever I want; one of the aspects of print-on-demand. As said, the book will see updates with each CDK release, but I may also make intermediate releases, adding new chapters with new content. I'll have to see how things go.
Therefore, each book will be better than the last. This first edition was not too extensive, but covered all the basics. This second edition had several new chapters, in a continued effort to make this book really informative. In fact, so much had changed, that a new cover seemed a appropriate. The drawing is made by Josefien Willighagen. This third edition again adds more content, making it almost twice as thick as the first one.
Have fun, and do let me know if you like to see something added!

Egon Willighagen
egon.willighagen+cdkbook@gmail.com

Contents

1. Introduction

Readers of this book will probably know what the Chemistry Development Kit (CDK) is: *An Open-Source Java Library for Chemo- and Bioinformatics* [1]. While the CDK project was founded in 2000, the code base originates from the groundbreaking open source cheminformatics work Christoph Steinbeck started in 1997 with the JChemPaint [2] and CompChem projects. This book is not about those past projects, however; it is about the CDK as it is now. It has evolved enormously over the past 10 years, and got more and more functionality [3], thanx to the many contributors (see page 107). Moreover, by now, the CDK has shown its role in many cheminformatics and bioinformatics fields, and you will find that this book cites many scientific papers that use the CDK.

The goal of this book is to introduce the reader, you, to the wide variety of functionality available in the library. It will discuss parts of the data model, basic cheminformatics algorithms, chemical file formats, as well as some of the applications of the CDK.

During the discussion of the various features, we will also discuss some of the important cheminformatics concepts. We will discuss bits about chemical graph theory, computer representation, etc. But the goal of this book is not to provide an introduction into cheminformatics. For that, various other books are available [4–7].

As such, this book does require a basic chemical education. It assumes that you know what atoms are, how they are connected by chemical bonds, and it assumes some basic computer knowledge. This book is about learning how to perform cheminformatics tasks using the CDK. But to keep the required knowledge to a minimum, the examples will be pretty verbose.

Moreover, at the end of the book you can find an appendix containing a keyword list, where each keyword reflects some cheminformatics concept, linking to to matching CDK class or method that provides related functionality. As such, a secondary goal of this book is to serve as reference material for more experienced CDK developers.

References

[1] C. Steinbeck, Y. Han, S. Kuhn, O. Horlacher, E. Luttmann, E. Willighagen, The Chemistry Development Kit (CDK): An Open-Source Java Library for Chemo- and Bioinformatics, *Journal of Chemical Information and Computer Sciences* **2003**, *43*, 493–500.

[2] S. Krause, E. L. Willighagen, C. Steinbeck, JChemPaint - Using the Collaborative Forces of the Internet to Develop a Free Editor for 2D Chemical Structures, *Molecules* **2000**, *5*, 93–98.

[3] C. Steinbeck, C. Hoppe, S. Kuhn, M. Floris, R. Guha, E. L. Willighagen, Recent developments of the chemistry development kit (CDK) - an open-source java library for chemo- and bioinformatics. *Current pharmaceutical design* **2006**, *12*, 2111–2120.

[4] J. Gasteiger, T. Engel, *Chemoinformatics : a textbook* (Ed.: J. Gasteiger), Wiley-VCH, 1st ed., **2003**.

[5] A. R. Leach, V. J. Gillet, *An introduction to chemoinformatics*, Springer, Rev. Ed, **2007**.

[6] J.-L. Faulon, A. Bender, *Handbook of Chemoinformatics Algorithms (Chapman & Hall/CRC Mathematical & Computational Biology)*, Chapman and Hall/CRC, 1st ed., **2010**.

[7] J. Wikberg, M. Eklund, E. Willighagen, O. Spjuth, M. Lapins, O. Engkvist, J. Alvarsson, *Introduction to Pharmaceutical Bioinformatics*, Oakleaf Academic, Stockholm, Sweden, **2010**.

2. Atoms, Bonds and Molecules

The basic objects in the CDK are the IAtom, IBond and IAtomContainer [1]. The name of the latter is somewhat misleading, as it contains not just IAtoms but also IBonds. The primary use of the model is the graph-based representation of molecules, where bonds are edges between two atoms being the nodes [2].
Before we start, it is important to note that CDK 1.4.1 has an important convention around object properties: when a property is unset, the object's field is set to *null*. This brings in sources for NullPointerExceptions, but also allows us to distinguish between, for example, zero and unset formal charge.

2.1. Atoms

The CDK interface IAtom is the underlying data model of atoms. Creating a new atom is fairly easy. For example, we can create an atom of element type carbon, as defined by the element's symbol that we pass as parameter in the constructor:

Script 2-1: code/CreateAtom1.java

```
IAtom atom = new Atom("C");
```

Alternatively, we can also construct a new carbon atom, by passing a carbon IElement, conveniently provided by the Elements class:

Script 2-2: code/CreateAtom2.java

```
IAtom atom = new Atom(Elements.CARBON);
```

A CDK atom has many properties, many of them inherited from the IElement, IIsotope and IAtomType interfaces. Figure 2.1 shows the interface inheritance specified by the CDK data model.

2.1.1. IElement

The most common property of IElements are their *symbol* and *atomic number*. Because the IAtom extends the IElement, CDK atoms also have these properties. Therefore, we can set these properties for atoms manually too:

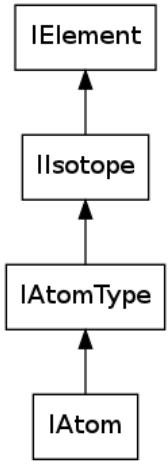

Figure 2.1.: The IAtom interface extends the IAtomType interface, which extends the IIsotope interface, which, in turn, extends the IElement interface.

Script 2-3: code/ElementProperties.groovy

```
atom.setSymbol("N")
atom.setAtomicNumber(7)
```

Of course, we can use the matching get methods to recover the properties:

Script 2-4: code/ElementGetProperties.groovy

```
IAtom atom = new Atom(Elements.CARBON);
println "Symbol: " + atom.getSymbol()
println "Atomic number: " + atom.getAtomicNumber()
```

which outputs:

```
Symbol: C
Atomic number: 6
```

2.1.2. IIsotope

The IIsotope information consists of the *mass number*, *exact mass* and *natural abundance*:

Script 2-5: code/IsotopeProperties.groovy

```
IAtom atom = new Atom("C");
atom.setMassNumber(13)
atom.setNaturalAbundance(1.07)
atom.setExactMass(13.00335484)
```

Here too, the complementary *get* methods are available:

Script 2-6: code/IsotopeGetProperties.groovy

```
println "Mass number: " + atom.getMassNumber()
println "Natural abundance: " + atom.getNaturalAbundance()
println "Exact mass: " + atom.getExactMass()
```

giving:

```
Mass number: 13
Natural abundance: 1.07
Exact mass: 13.00335484
```

Appendix B lists all isotopes defined in the CDK with a natural abundance of more then 0.1.

2.1.3. IAtomType

Atom types are an important concept in cheminformatics. They describe some basic facts about that particular atom in some particular configuration. These properties are used in many cheminformatics algorithms, including adding hydrogens to hydrogen-depleted chemical graphs (see Section 8.2.1) and force fields. Chapter 6 provides much more detail on the atom type infrastructure in the CDK library, and, for example, details how atom types can be perceived, and how atom type information is set for atoms.

The IAtomType interface contains fields that relate to atom types. These properties include *formal charge*, *neighbor count*, *maximum bond order* and *atom type name*:

Script 2-7: code/AtomTypeProperties.groovy

```
atom.setAtomTypeName("C.3")
atom.setFormalCharge(-1)
atom.setMaxBondOrder(IBond.Order.SINGLE)
atom.setFormalNeighbourCount(4)
```

2.2. Bonds

The IBond interface of the CDK is an interaction between two or more IAtoms, extending the IElectronContainer interface. While the most common application in the CDK originates from graph theory [2], it is not restricted to that. That said, many algorithms implemented in the CDK expect a graph theory based model, where each bond connects two, and not more, atoms.
For example, to create ethanol we write:

Script 2-8: code/Ethanol.groovy

```
IAtom atom1 = new Atom("C")
IAtom atom2 = new Atom("C")
IAtom atom3 = new Atom("O")
IBond bond1 = new Bond(atom1, atom2, IBond.Order.SINGLE);
IBond bond2 = new Bond(atom2, atom3, IBond.Order.SINGLE);
```

The CDK has a few bond orders, which we can list with this groovy code:

Script 2-9: code/BondOrders.groovy

```
IBond.Order.each {
  println it
}
```

which outputs:

```
SINGLE
DOUBLE
TRIPLE
QUADRUPLE
```

As you might notice, there is no AROMATIC bond defined. This is deliberate and the CDK allows to define single-double bond order patterns at the same time as aromaticity information. For example, a kekule structure of benzene with bonds marked as aromatic can be constructed with:

Script 2-10: code/AromaticBond.groovy

```
IAtom atom1 = new Atom("C")
IAtom atom2 = new Atom("C")
IAtom atom3 = new Atom("C")
IAtom atom4 = new Atom("C")
IAtom atom5 = new Atom("C")
```

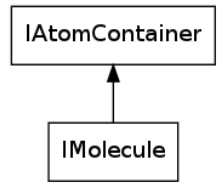

Figure 2.2.: The IMolecule interface extends the IAtomContainer interface. It does not add any new methods, but just the implied meaning that an IMolecule is full connected.

```
IAtom atom6 = new Atom("C")
IBond bond1 = new Bond(atom1, atom2, IBond.Order.SINGLE)
IBond bond2 = new Bond(atom2, atom3, IBond.Order.DOUBLE)
IBond bond3 = new Bond(atom3, atom4, IBond.Order.SINGLE)
IBond bond4 = new Bond(atom4, atom5, IBond.Order.DOUBLE)
IBond bond5 = new Bond(atom5, atom6, IBond.Order.SINGLE)
IBond bond6 = new Bond(atom6, atom1, IBond.Order.DOUBLE)
bond1.setFlag(CDKConstants.ISAROMATIC, true);
bond2.setFlag(CDKConstants.ISAROMATIC, true);
bond3.setFlag(CDKConstants.ISAROMATIC, true);
bond4.setFlag(CDKConstants.ISAROMATIC, true);
bond5.setFlag(CDKConstants.ISAROMATIC, true);
bond6.setFlag(CDKConstants.ISAROMATIC, true);
```

2.3. Molecules

We already saw in the previous pieces of code how the CDK can be used to create molecules, and while the above is strictly enough to find all atoms in the molecule starting with only one of the atoms in the molecule, it often is more convenient to store all atoms and bonds in a container.
The CDK has two main containers, which are identical in functionality, but which have different semantics: the IAtomContainer and the IMolecule (see Figure 2.2. The first is a general container to holds atoms an bonds, while the IMolecule has the added implication that it is meant that the container holds a single molecule, of which all atoms are connected to each other via one or more covalent bonds. It is important to note, however, that the latter is not enforced.
Adding atoms and bonds is done by the methods addAtom(IAtom) and

addBond(IBond):

Script 2-11: code/AtomContainerAddAtomsAndBonds.groovy

```
mol = new AtomContainer();
mol.addAtom(new Atom("C"));
mol.addAtom(new Atom("H"));
mol.addAtom(new Atom("H"));
mol.addAtom(new Atom("H"));
mol.addAtom(new Atom("H"));
mol.addBond(new Bond(mol.getAtom(0), mol.getAtom(1)));
mol.addBond(new Bond(mol.getAtom(0), mol.getAtom(2)));
mol.addBond(new Bond(mol.getAtom(0), mol.getAtom(3)));
mol.addBond(new Bond(mol.getAtom(0), mol.getAtom(4)));
```

The addBond() method has an alternative which takes three parameters: the first atom, the second atom, and the bond order. Note that atom indices follows programmers habits and starts at 0, as you can observe in the previous example too. This shortens the previous version a bit:

Script 2-12: code/AtomContainerAddAtomsAndBonds2.groovy

```
mol = new AtomContainer();
mol.addAtom(new Atom("C"));
mol.addAtom(new Atom("H"));
mol.addAtom(new Atom("H"));
mol.addAtom(new Atom("H"));
mol.addAtom(new Atom("H"));
mol.addBond(0,1,IBond.Order.SINGLE);
mol.addBond(0,2,IBond.Order.SINGLE);
mol.addBond(0,3,IBond.Order.SINGLE);
mol.addBond(0,4,IBond.Order.SINGLE);
```

2.3.1. Iterating over atoms and bonds

The IAtomContainer comes with convenience methods to iterate over atoms and bonds. Both methods use the Iterable interfaces, and for atoms we do:

Script 2-13: code/CountHydrogens.groovy

```
int hydrogenCount = 0
for (IAtom atom : mol.atoms()) {
    if ("H".equals(atom.getSymbol())) hydrogenCount++
```

```
}
println "Number of hydrogens: $hydrogenCount"
```

which returns

```
Number of hydrogens: 4
```

And for bonds the equivalent:

Script 2-14: code/CountDoubleBonds.groovy

```
int doubleBondCount = 0
for (IBond bond : mol.bonds()) {
  if (IBond.Order.DOUBLE == bond.getOrder())
    doubleBondCount++
}
println "Number of double bonds: $doubleBondCount"
```

giving

```
Number of double bonds: 1
```

2.3.2. Neighboring atoms and bonds

It is quite common that you like to see what atoms are connected to one particular atom. For example, you may wish to count how many bonds surround a particular atom. Or, you may want to list all atoms that are bound to this atom. The IAtomContainer class provides methods for these use cases. But it should be stressed that these methods do only take into account explicit hydrogens (see the next section).
Let's consider ethanol again, given in Script 2-8, and count the number of neighbors for each atom:

Script 2-15: code/NeighborCount.groovy

```
for (atom in ethanol.atoms()) {
  println atom.getSymbol() +
    " " + ethanol.getConnectedAtomsCount(atom)
}
```

which lists for the three heavy atoms:

```
C 1
C 2
O 1
```

Similarly, we can also list all connected atoms:

Script 2-16: code/ConnectedAtoms.groovy

```
for (atom in ethanol.atoms()) {
  print atom.getSymbol() +
    " is connected to "
  for (neighbor in ethanol.getConnectedAtomsList(atom)) {
    print neighbor.getSymbol() + " "
  }
  println ""
}
```

which outputs:

```
C is connected to C
C is connected to C O
O is connected to C
```

We can do the same thing for connected bonds:

Script 2-17: code/ConnectedBonds.groovy

```
for (atom in ethanol.atoms()) {
  print atom.getSymbol() +
    " has bond(s)"
  for (bond in ethanol.getConnectedBondsList(atom)) {
    print " " + bond.getOrder()
  }
  println ""
}
```

which outputs:

```
C has bond(s) SINGLE
C has bond(s) SINGLE SINGLE
O has bond(s) SINGLE
```

2.3.3. Molecular Formula

Getting the molecular formula of a molecule and returning that as a String is both done with the MolecularFormulaManipulator class:

Script 2-18: code/MFGeneration.groovy

```
molForm = MolecularFormulaManipulator.getMolecularFormula(
  azulene
```

```
)
mfString = MolecularFormulaManipulator.getString(molForm)
println "Azulene: $mfString"
```

giving:

```
Azulene: C10H8
```

The method assumes the atom container has explicit hydrogens (see Section 2.4). If your atom container does not have explicit hydrogens, you should add those first, as explained in Section 8.2.2.

2.4. Implicit and Explicit Hydrogens

The CDK has two concepts for hydrogens: *implicit hydrogens* and *explicit hydrogens*. Explicit hydrogens are hydrogens that are separate vertices on the chemical graph. Implicit hydrogens, however, are not, and are attributes of existing vertices.
For example, if we represent methane as a chemical graph, we can define either a hydrogen-depleted chemical graph with a single carbon atom and zero bonds, or a graph with one carbon and four hydrogen atoms, and four bonds connecting the hydrogens to the central carbon. In the latter case, the hydrogens are explicit, while in the former case we can add those four hydrogens as implicit hydrogens on these carbon.
The first option in CDK code looks like:

Script 2-19: code/HydrogenDepletedGraph.groovy

```
molecule = new Molecule();
carbon = new Atom(Elements.CARBON);
carbon.setImplicitHydrogenCount(4);
molecule.addAtom(carbon);
```

while the alternative look like:

Script 2-20: code/HydrogenExplicitGraph.groovy

```
molecule = new Molecule();
carbon = new Atom(Elements.CARBON);
molecule.addAtom(carbon);
for (int i=1; i<=4; i++) {
  hydrogen = new Atom(Elements.HYDROGEN);
  molecule.addAtom(hydrogen);
  molecule.addBond(0, i, IBond.Order.SINGLE);
}
```

Section 8.2 describes how hydrogens can be added programmatically.

2.5. Chemical Objects

Another interface that must be introduced is the IChemOject as it plays an key role in the CDK data model. Almost all interfaces used in the data model inherit from this interface. The IChemObject interface provides a bit of basic functionality, including support for object identifiers, properties, and flags.
For example. identifiers are set and retrieved with the setID() and getID() methods:

Script 2-21: code/ChemObjectIdentifiers.groovy

```
butane = new Molecule();
butane.setID("cdkbook000000001")
print "ID: " + butane.getID()
```

If you have more than one identifier, or other properties you like to associate with objects, you can use the setProperty() and getProperty() methods:

Script 2-22: code/ChemObjectProperties.groovy

```
butane = new Molecule();
butane.setProperty(
  "InChI", "InChI=1/C4H10/c1-3-4-2/h3-4H2,1-2H3"
)
print "InChI: " + butane.getProperty("InChI")
```

For example, we can use this approach to assign labels to atoms, such as in this example from substructure searching (see Chapter 10):

Script 2-23: code/AtomLabels.groovy

```
butane = MoleculeFactory.makeAlkane(4);
butane.atoms().each { atom ->
  atom.setProperty("Label", "Molecule")
}
ccc = MoleculeFactory.makeAlkane(3);
ccc.atoms().each { atom ->
  atom.setProperty("Label", "Substructure")
}
```

The CDKConstants class provides a few constants for common properties:

Script 2-24: code/CDKConstantsProperties.groovy

```
println "Title: " +
  aspirin.getProperty(CDKConstants.TITLE)
println "InChI: " +
  aspirin.getProperty(CDKConstants.INCHI)
println "SMILES: " +
  aspirin.getProperty(CDKConstants.SMILES)
println "CAS registry number: " +
  aspirin.getProperty(CDKConstants.CASRN)
println "COMMENT: " +
  aspirin.getProperty(CDKConstants.COMMENT)
println "NAMES: " +
  aspirin.getProperty(CDKConstants.NAMES)
```

outputting:

```
Title: aspirin
InChI: InChI=1/C9H8O4/c1-6(10)13-8-5-3-2-4-7(8)9(11)12/h...
  2-5H,1H3,(H,11,12)
SMILES: CC(=O)Oc1ccccc1C(=O)O
CAS registry number: 50-78-2
COMMENT: Against headaches.
NAMES: 2-(acetyloxy)benzoic acid
```

A third characteristic of the IChemObject interface is the concept of flags. Flags are used in the CDK to indicate, for example, if an atom or bond is aromatic (see Script 2-10) or if an atom is part of a ring:

Script 2-25: code/RingBond.groovy

```
benzene = MoleculeFactory.makeBenzene();
benzene.bonds().each { bond ->
  bond.setFlag(CDKConstants.ISINRING, true)
  println "Is ring bond: " +
    bond.getFlag(CDKConstants.ISINRING)
}
```

The next section talks about the CDK data class for rings.

2.6. Rings

One important aspect of molecules is rings, partly because rings can show interesting chemical phenomena. For example, if the number of π electrons

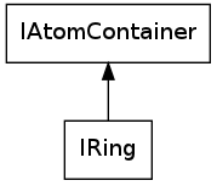

Figure 2.3.: The IRing interface extends the IAtomContainer interface and is used to hold information about rings.

is right, then the ring will become aromatic, as we commonly observer in phenyl rings, such as in benzene. But, cheminformatics has many other aspects where one like to know about those rings. For example, 2D coordinate generator (see Section 8.3) requires algorithms to know what the rings are in a molecule.
Section 7.2 explains what functionality the CDK has to determine a bond takes part in a ring system. Here, we just introduce the IRing interface, which extends the more general IAtomContainer as shown in Figure 2.3. Practically, there is nothing much to say about the IRing interface. One method it adds, is to get the size of the ring:

Script 2-26: code/Ring.groovy

```
IRing ring = new Ring(5, "C")
println "Ring size: " + ring.getRingSize()
println "Ring atoms: " + ring.getAtomCount()
println "Ring bonds: " + ring.getBondCount()
```

But this should be by definition the same as the number as atoms and bonds:

```
Ring size: 5
Ring atoms: 5
Ring bonds: 5
```

References

[1] C. Steinbeck, Y. Han, S. Kuhn, O. Horlacher, E. Luttmann, E. Willighagen, The Chemistry Development Kit (CDK): An Open-Source Java Library for Chemo- and Bioinformatics, *Journal of Chemical Information and Computer Sciences* **2003**, *43*, 493–500.

[2] A. T. Balaban, Applications of graph theory in chemistry, *Journal of Chemical Information and Computer Sciences* **1985**, *25*, 334–343.

3. Salts and other disconnected structures

In Section 2.3 we saw how atoms and bonds are contained in the IMolecule data model. It was mentioned that the molecule object is aimed at connected structures, while IAtomContainer is meant as a general container. And this is exactly what we need for disconnected structures like salts and molecular crystal structures.

Functionality to determine if the content of an IAtomContainer is connected, you can use the ConnectivityChecker, as explained in Section 7.1.

3.1. Salts

Salts are one of the most common disconnected structures found in compound databases: a salt is a combination of two or more connected molecules bound to each other by coulombic interactions. These may be solids.

A common kitchen example is the table salt sodium chloride. We can represent this using the following model:

Script 3-1: code/Salt.groovy

```
salt = new AtomContainer();
sodium = new Atom("Na");
sodium.setFormalCharge(+1);
chloride = new Atom("Cl");
chloride.setFormalCharge(-1);
salt.addAtom(sodium);
salt.addAtom(chloride);
```

If you prefer a single IAtomContainer to only contain connected atoms, instead of unbound atoms as in this salt example, you can partition them into two or more new containers, as explained in Section 7.1.

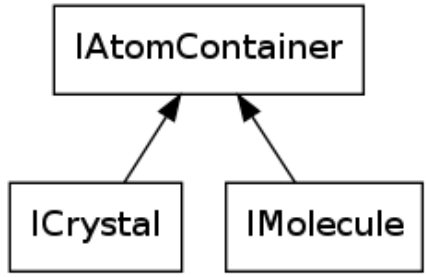

Figure 3.1.: The ICrystal and IMolecule interfaces both extends the IAtomContainer interface.

3.2. Crystals

Of course, the representation given in the previous section is a very basic model for sodium chloride. A crystal structure would perhaps be a more accurate description of what you like to represent. In this case, the ICrystal subclass of the IAtomContainer can be used (see Figure 3.1):

Script 3-2: code/SaltCrystal.groovy

```
salt = new Crystal();
sodium = new Atom("Na");
sodium.setFormalCharge(+1);
chloride = new Atom("Cl");
chloride.setFormalCharge(-1);
salt.addAtom(sodium);
salt.addAtom(chloride);
```

If we want to add the crystal structure parameters and crystal structure coordinates of the atoms, we add can add them too (data taken from [1]):

Script 3-3: code/SaltCrystalParam.groovy

```
salt = new Crystal();
salt.setA(new Vector3d(5.6402, 0, 0));
salt.setB(new Vector3d(0, 5.6402, 0));
salt.setC(new Vector3d(0, 0, 5.6402));
salt.setZ(4);
sodium = new Atom("Na");
sodium.setFormalCharge(+1);
```

```
sodium.setFractionalPoint3d(
  new Point3d(0, 0, 0)
);
chloride = new Atom("Cl");
chloride.setFormalCharge(-1);
chloride.setFractionalPoint3d(
  new Point3d(0.5, 0.5, 0.5)
);
salt.addAtom(sodium);
salt.addAtom(chloride);
```

References

[1] http://www.ilpi.com/inorganic/structures/nacl/.

4. Paired and unpaired electrons

The CDK data model supports more than just the chemical graph. We have seen atoms and bonds earlier, the bonds being atoms sharing electrons. Atoms, however, can also have electrons in the valence shell not involved in bond: free electon (lone) pairs, and unpaired electrons as found in radicals.

But before we look at how we add paired and unpaired electrons, we should first look at the two principle classes involved in representing these concepts. Like bonds, a free electron pair and a unpaired electron are bound to the atom. Depending on the theory used, the exact environment which holds the electrons can be named differently. For example, they might be referred to as orbitals, atomic or molecular. The CDK simply refers to the holder as IElectronContainers, and has several subinterfaces for bonds (IBond), lone pairs (ILonePair), and unpaired electrons (ISingleElectron), as shown in Figure 4.1.

4.1. Lone Pairs

Oxygens are atoms with lone pairs: the free electrons that do not take part in a bond. These lone pairs can be explicitly modeled in the CDK. For example, this is how we can represent water:

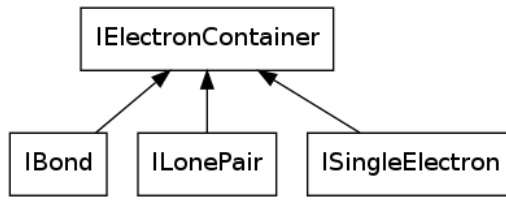

Figure 4.1.: The IBond, ILonePair, and ISingleElectron interfaces all extend the IElectronContainer interface.

Script 4-1: code/LonePairOxygen.groovy

```
IAtom atom1 = new Atom("H")
IAtom atom2 = new Atom("H")
IAtom atom3 = new Atom("O")
IBond bond1 = new Bond(atom1, atom2, IBond.Order.SINGLE)
IBond bond2 = new Bond(atom2, atom3, IBond.Order.SINGLE)
IMolecule water = new Molecule()
water.addAtom(atom1)
water.addAtom(atom2)
water.addAtom(atom3)
water.addBond(bond1)
water.addBond(bond2)
water.addLonePair(new LonePair(atom3))
water.addLonePair(new LonePair(atom3))
```

And we can the count the number of lone pair on each atom with, for example, this code:

Script 4-2: code/LonePairCount.groovy

```
for (atom in water.atoms()) {
  println atom.getSymbol() + " has " +
    water.getConnectedLonePairsCount(atom) +
    " lone pairs"
}
```

which gives us:

```
H has 0 lone pairs
H has 0 lone pairs
O has 2 lone pairs
```

4.2. Unpaired electrons

An unpaired electron on an atom makes that atom a radical. Radicals are common mass spectroscopy and as the latter is an important use case of the CDK, unpaired electrons are well-supported in the data model.
We can add an unpaired electron with the addSingleElectron method of the IAtomContainer class.

Script 4-3: code/HydrogenRadical.groovy

```
hydrogen = new Atom("H")
radicalElectron =
  new SingleElectron(hydrogen)
hydrogenRadical = new Molecule()
hydrogenRadical.addAtom(hydrogen)
hydrogenRadical.addSingleElectron(radicalElectron)
```

5. Input/Output

The CDK has functionality for extracting information from files in many different file formats. Unfortunately, hardly ever the full format specification is supported, but generally the chemical graph and 2D or 3D coordinates are extracted, not uncommonly complemented with formal or partial charge.

5.1. File Format Detection

Typically, a human is fairly aware about the format of a file. Computer programs require a fairly accurate method for detecting the chemical format of a file. To detect the format of a file, the FormatFactory can be used:

Script 5-1: code/GuessFormat.groovy

```
Reader stringReader = new StringReader(
  "<molecule xmlns='http://www.xml-cml.org/schema'/>"
);
FormatFactory factory = new FormatFactory();
IChemFormat format = factory.guessFormat(stringReader);
System.out.println("Format: " + format.getFormatName());
```

For example, this script recognizes that a file has the Chemical Markup Language [1, 2] format:

```
Format: Chemical Markup Language
```

To learn if the CDK has a IChemObjectReader or IChemObjectWriter one can use the methods getReaderClassName() and getWriterClassName() respectively:

Script 5-2: code/HasReaderOrWriter.groovy

```
Reader stringReader = new StringReader(
  "<molecule xmlns='http://www.xml-cml.org/schema'/>"
);
IChemFormat format = factory.guessFormat(stringReader);
```

Table 5.1.: A few of the formats and their readers and writers.

Format	Description	Reader/Writer
Chemical Markup Language	XML-based file format that can store connection tables, atomic properties, 2D and 3D coordinates, crystal structures, and much more.	CMLReader, CMLWriter
MDL molfile	Text-based format that can stored connection tables, atomic properties like formal charge, and 2D or 3D coordinates.	MDLV2000Reader, MDLV2000Writer
MDL SD file	Extension of the *MDL molfile* format to store a database of molecules and associated properties. See Section 5.5.1.	MDLV2000Reader, SDFWriter
PDB	Format used by the Protein Database which stored 3D geometries or proteins and protein-ligand complexes. It can also be used for small molecules.	PDBReader
XYZ	Simple format that can store 3D coordinates and optionally partial atomic charges. It cannot store connectivity information.	XYZReader, XYZWriter

```
String readerClass = format.getReaderClassName();
String writerClass = format.getWriterClassName();
System.out.println("Reader: " + readerClass);
System.out.println("Writer: " + writerClass);
```

It reports:

```
Reader: org.openscience.cdk.io.CMLReader
Writer: org.openscience.cdk.io.CMLWriter
```

5.2. Reading from Readers and InputStreams

Many input readers in the CDK allow reading from a Java Reader class, but all are required to also read from an InputStream. The difference between these two Java classes is that the Reader is based on a character stream, while an InputStream is based on an byte stream. For some readers this difference is crucial: processing an XML based format, such as CML and XML formats used by PubChem should be read from an InputStream, not a Reader.

For other formats, it does not matter. This allows, for example, to read a file easily from a string with a StringReader (mind the newlines indicated by '\n'):

Script 5-3: code/InputFromStringReader.groovy

```
String bf3 = "4\n" +
"Bortrifluorid\n" +
"B    0.0000    0.0000    0.0000\n" +
"F    1.0000    0.0000    0.0000\n" +
"F   -0.5000   -0.8660    0.0000\n" +
"F   -0.5000    0.8660    0.0000\n";
reader = new XYZReader(
  new StringReader(bf3)
)
chemfile = reader.read(new NNChemFile())
mol = ChemFileManipulator.getAllAtomContainers(chemfile)
  .get(0)
println "Atom count: $mol.atomCount"
```

But besides reading XML files correctly, the support for InputStream also allows reading files directly from the internet and from gziped files (see Section 5.4).

5.2.1. Example: Downloading Domoic Acid from PubChem

As an example, below will follow a small script that takes a PubChem compound identifier (CID) and downloads the corresponding ASN.1 XML file, parses it and counts the number of atoms:

Script 5-4: code/PubChemDownload.groovy

```
cid = 5282253
reader = new PCCompoundXMLReader(
  new URL(
    "http://pubchem.ncbi.nlm.nih.gov/summary/" +
    "summary.cgi?cid=$cid&disopt=SaveXML"
  ).newInputStream()
)
mol = reader.read(new NNMolecule())
println "CID: " + mol.getProperty("PubChem CID")
println "Atom count: $mol.atomCount"
```

It reports:

```
CID: 5282253
Atom count: 43
```

PubChem ASN.1 files come with an extensive list of molecular properties. These are stored as properties on the molecule object and can be retrieved using the getProperties() method, or, using the Groovy bean formalism:

Script 5-5: code/PubChemDownloadProperties.groovy

```
mol.properties.each {
  line = "" + it
  println line
}
```

which lists the properties for the earlier downloaded domoic acid:

```
PubChem CID=5282253
Compound Complexity=510
Fingerprint (SubStructure Keys)=00000371E072380000000000...
  00000000000000000001600000000000000000000000000000000...
  1E00100800000D28C18004020802C00200880220D2080000000020...
  000008088188000800A00120081200440000 4D000988003BC7F020E...
  80000000000000000000000000000000000000000000000
IUPAC Name (Allowed)=(2S,3S,4S)-3-(carboxymethyl)-4-[(1Z...
  ,3E,5R)-5-carboxy-1-methyl-hexa-1,3-dienyl]pyrrolidine...
  -2-carboxylic acid
IUPAC Name (CAS-like Style)=(2S,3S,4S)-4-[(2Z,4E,6R)-6-c...
  arboxyhepta-2,4-dien-2-yl]-3-(carboxymethyl)-2-pyrroli...
  dinecarboxylic acid
IUPAC Name (Preferred)=(2S,3S,4S)-4-[(2Z,4E,6R)-6-carbox...
  yhepta-2,4-dien-2-yl]-3-(carboxymethyl)pyrrolidine-2-c...
  arboxylic acid
IUPAC Name (Systematic)=(2S,3S,4S)-3-(2-hydroxy-2-oxoeth...
  yl)-4-[(2Z,4E,6R)-6-methyl-7-oxidanyl-7-oxidanylidene-...
  hepta-2,4-dien-2-yl]pyrrolidine-2-carboxylic acid
IUPAC Name (Traditional)=(2S,3S,4S)-3-(carboxymethyl)-4-...
  [(1Z,3E,5R)-5-carboxy-1-methyl-hexa-1,3-dienyl]proline
InChI (Standard)=InChI=1S/C15H21NO6/c1-8(4-3-5-9(2)14(19...
  )20)11-7-16-13(15(21)22)10(11)6-12(17)18/h3-5,9-11,13,...
  16H,6-7H2,1-2H3,(H,17,18)(H,19,20)(H,21,22)/b5-3+,8-4-...
  /t9-,10+,11-,13+/m1/s1
InChIKey (Standard)=VZFRNCSOCOPNDB-AOKDLOFSSA-N
Log P (XLogP3-AA)=-1.3
```

```
Mass (Exact)=311.136887
Molecular Formula=C15H21NO6
Molecular Weight=311.33034
SMILES (Canonical)=CC(C=CC=C(C)C1CNC(C1CC(=O)O)C(=O)O)C(...
  =O)O
SMILES (Isomeric)=C[C@H](/C=C/C=C(/C)\textbackslash[C@H]...
  1CN[C@@H]([C@H]1CC(=O)O)C(=O)O)C(=O)O
Topological (Polar Surface Area)=124
Weight (MonoIsotopic)=311.136887
```

5.3. Input Validation

The history of the CDK project has seen many bug reports about problems which in fact turned out to be problems with in the input file. While the general perception seems to be that because files could be written, the content must be consistent.
However, this is a strong misconception. There are several problems found in chemical files in the wild. A first common problem is that the file is not conform the syntax of the specification. An example here can be that at places where a number is expected, something else is given; not uncommonly, this is caused by incorrect use of whitespace.
A second problem is that the file looks perfectly reasonable, but that the software that wrote the file used conventions and extensions that are not supported by the reading software. A common example is the use of the D and T symbols, for deuterium and tritium in MDL molfiles, where the specification does not allow that.
A third problem is that most chemical file formats do not disallow incorrect chemical graphs. For example, formats often allow to bind an atom to itself, which will cause problems when analyzing this graph. These problems are much more rare, though.

5.3.1. Reading modes

The IChemObjectReader has a feature that allows setting a validating mode, which has two values:

Script 5-6: code/ReadingModes.groovy

```
IChemObjectReader.Mode.each {
  println it
}
```

returning:

```
RELAXED
STRICT
```

The STRICT mode follows the exact format specification. There RELAXED mode allows for a few common extensions, such as the support for the T and D element types. For example, let's consider this file:

```
 CDK

  3  2 0 0 0 0 0 0 0  0999 V2000
    2.5369   -0.1550    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
    3.0739    0.1550    0.0000 D   1  0  0  0  0  0  0  0  0  0  0  0
    2.0000    0.1550    0.0000 T   1  0  0  0  0  0  0  0  0  0  0  0
  1  2  1  0  0  0  0
  1  3  1  0  0  0  0
M  ISO   2   2   2   3   3
M  END
```

If we read this file with:

Script 5-7: code/ReadStrict.groovy

```
reader = new MDLV2000Reader(
  new File("data/t.mol").newReader(),
  Mode.STRICT
);
water = reader.read(new Molecule());
println "atom count: $water.atomCount"
```

we get this exception:

```
Invalid element type. Must be an existing element, or on...
  e in: A, Q, L, LP, *.
```

However, if we read the file in RELAXED mode with this code:

Script 5-8: code/ReadRelaxed.groovy

```
reader = new MDLV2000Reader(
  new File("data/t.mol").newReader(),
  Mode.RELAXED
);
water = reader.read(new Molecule());
println "atom count: $water.atomCount"
```

the files will be read as desired:

```
atom count: 3
```

5.3.2. Validation

When a file is being read in RELAXED mode, it is possible to get error messages. This functionality is provided by the IChemObjectReaderErrorHandler support in IChemObjectReader. For example, we can define this custom error handler:

Script 5-9: code/CustomErrorHandler.groovy

```
class ErrorHandler
implements IChemObjectReaderErrorHandler {
  public void handleError(String message) {
    println message;
  };
  public void handleError(String message,
    Exception exception)
  {
    println message + "\n  -> " +
            exception.getMessage();
  };
  public void handleError(String message,
    int row, int colStart, int colEnd)
  {
    print "location: " + row + ", " +
          colStart + "-" + colEnd + ": ";
    println message;
  };
  public void handleError(String message,
    int row, int colStart, int colEnd,
    Exception exception)
  {
    print "location: " + row + ", " +
          colStart + "-" + colEnd + ": "
    println message + "\n  -> " +
            exception.getMessage()
  };
}
```

and use that when reading a file:

Script 5-10: code/ReadErrorHandler.groovy

```
reader = new MDLV2000Reader(
  new File("data/t.mol").newReader(),
```

```
    Mode.RELAXED
);
reader.setErrorHandler(new ErrorHandler());
water = reader.read(new Molecule());
```

we get these warnings via the handler interface:

```
location: 6, 32-35: Invalid element type. Must be an exi...
  sting element, or one in: A, Q, L, LP, *.
location: 7, 32-35: Invalid element type. Must be an exi...
  sting element, or one in: A, Q, L, LP, *.
```

5.4. Gzipped files

Some remote databases gzip their data files to reduce download sized. The Protein Brookhaven Database (PDB) is such a database. Fortunately, Java has a simple API to work with gziped files, using the GZIPInputStream:

Script 5-11: code/PDBCoordinateExtraction.groovy

```
reader = new PDBReader(
  new GZIPInputStream(
    new URL(
      "http://www.pdb.org/pdb/files/1CRN.pdb.gz"
    ).openStream()
  )
);
crambin = reader.read(new ChemFile());
for (container in
     ChemFileManipulator.getAllAtomContainers(
       crambin
     )) {
  for (atom in container.atoms()) {
    println atom.point3d;
  }
}
```

5.5. Iterating Readers

By default, the CDK readers read structures into memory. This is fine when it is a relatively small model. It no longer works for large files, such as 1GB

MDLSDfiles [3]. To allow processing of such large files, the CDK can take advantage from the fact that these SD files are basically a concatenation of MDL molfiles. Therefore, one can use an iterating reader to process each individual molecule one by one.

5.5.1. MDL SD files

MDL SD files can be processed using the IteratingMDLReader, for example, to generate a SMILES for each structure:

Script 5-12: code/IteratingMDLReaderDemo.groovy

```
iterator = new IteratingMDLReader(
  new File("data/test6.sdf").newReader(),
  DefaultChemObjectBuilder.getInstance()
)
while (iterator.hasNext()) {
  IMolecule mol = iterator.next()
  formula = MolecularFormulaManipulator.getMolecularFormula(mol)
  println MolecularFormulaManipulator.getString(formula)
}
```

Which outputs the molecular formula for the three entries in the file:

```
C19Br2N2O6
C20N2O5S
C17N2O6S
```

5.5.2. PubChem Compounds XML files

Similarly, PubChem Compounds XML files can be processed taking advantage of a XML pull library, which is nicely hidden behind the same iterator interface as used for parsing MDL SD files. Iterating over a set of compounds is fairly straightforward with the IteratingPCCompoundXMLReader class:

Script 5-13: code/PubChemCompoundsXMLDemo.groovy

```
iterator = new IteratingPCCompoundXMLReader(
  new File("data/aceticAcids38.xml").newReader(),
  DefaultChemObjectBuilder.getInstance()
)
while (iterator.hasNext()) {
  IMolecule mol = iterator.next()
```

```
    formula = MolecularFormulaManipulator
      .getMolecularFormula(mol)
    println MolecularFormulaManipulator.getString(formula)
  }
```

Which outputs the molecular formula for the three entries in the aceticAcids38.xml file:

```
C2H4O2
C2H3O2
C2H3HgO2
```

5.6. Customizing the Output

An interesting feature of file IO in the 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
```

The output will look something like:

```
%nprocl=5
# b3lyp/6-31g* opt

Job started on Linux cluster on 20041010.

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. To get a list of options for a certain IO class in one does something along the lines:

Script 5-14: code/ListIOOptions.java

```
IChemObjectWriter writer = new GaussianInputWriter();
for (IOSetting setting : writer.getIOSettings()) {
  System.out.println("[" + setting.getName() + "]");
  System.out.println("Option: " + setting.getQuestion());
  System.out.println(
    "Current value: " + setting.getSetting()
  );
}
```

which results in the following output:

```
[Basis]
Option: Which basis set do you want to use?
Current value: 6-31g
[Method]
Option: Which method do you want to use?
Current value: b3lyp
[Command]
Option: What kind of job do you want to perform?
Current value: energy calculation
[Comment]
Option: What comment should be put in the file?
Current value: Created with CDK (http://cdk.sf.net/)
[OpenShell]
Option: Should the calculation be open shell?
Current value: false
[ProcessorCount]
Option: How many processors should be used by Gaussian?
Current value: 1
[UseCheckPointFile]
Option: Should a check point file be saved?
Current value: false
[Memory]
Option: How much memory do you want to use?
Current value: unset
```

5.6.1. Setting Properties

The IO settings system allows interactive setting of these options, but a perfectly fine alternative is to use a Java Properties object.
Consider the following source code:

Script 5-15: code/PropertiesSettings.java

```
// the custom settings
Properties customSettings = new Properties();
customSettings.setProperty("Basis",   "6-31g*");
customSettings.setProperty("Command",
  "geometry optimization");
customSettings.setProperty("Comment",
  "Job started on Linux cluster on 20041010.");
customSettings.setProperty("ProcessorCount", "5");
PropertiesListener listener = new PropertiesListener(
  customSettings
);
// create the writer
GaussianInputWriter writer = new GaussianInputWriter(
  new FileWriter(new File("methane.gin"))
);
writer.addChemObjectIOListener(listener);
XYZReader reader = new XYZReader(
  new FileReader(new File("data/methane.xyz"))
);
// convert the file
ChemFile content = (ChemFile)reader.read(new ChemFile());
IMolecule molecule = content.getChemSequence(0).
  getChemModel(0).getMoleculeSet().getMolecule(0);
writer.write(molecule);
writer.close();
```

The PropertiesListener takes a Properties class as parameter in its constructor. Therefore, the properties are defined by the customSettings variable in the first few lines. The PropertiesListener listener is the instantiated with the customizations as constructor parameter.
The output writer, specified to write to the 'methane.gin' file, is created after which the ChemObjectIOListener is set. Only by setting this listener, the output will be customized with the earlier defined properties. The rest of the code reads a molecule from an XYZ file and writes the content to the created Gaussian Input file.

5.6.2. Example: creating unit test for atom type perception

We saw earlier an example for reading files directly from PubChem (see Section 5.2.1). This can be conveniently used to create CDK source code,

for example, for use in unit tests for the atom type perception code (see Section 6.2). But because we do not want 2D and 3D coordinates being set in the source code, we disable those options:

Script 5-16: code/AtomTypeUnitTest.groovy

```
cid = 3396560
mol = reader.read(new Molecule())
stringWriter = new StringWriter();
CDKSourceCodeWriter writer =
  new CDKSourceCodeWriter(stringWriter);
customSettings = new Properties();
customSettings.setProperty("write2DCoordinates", "false");
customSettings.setProperty("write3DCoordinates", "false");
writer.addChemObjectIOListener(
  new PropertiesListener(
    customSettings
  )
)
writer.write(mol);
writer.close();
println stringWriter.toString();
```

This results in this source code:

```
{
  IChemObjectBuilder builder = DefaultChemObjectBuilder....
  getInstance();
  IMolecule mol = builder.newInstance(IMolecule.class);
  IAtom a1 = builder.newInstance(IAtom.class,"P");
  a1.setFormalCharge(0);
  mol.addAtom(a1);
  IAtom a2 = builder.newInstance(IAtom.class,"O");
  a2.setFormalCharge(0);
  mol.addAtom(a2);
  IAtom a3 = builder.newInstance(IAtom.class,"O");
  a3.setFormalCharge(0);
  mol.addAtom(a3);
  IAtom a4 = builder.newInstance(IAtom.class,"C");
  a4.setFormalCharge(0);
  mol.addAtom(a4);
  IAtom a5 = builder.newInstance(IAtom.class,"H");
  a5.setFormalCharge(0);
  mol.addAtom(a5);
  IAtom a6 = builder.newInstance(IAtom.class,"H");
  a6.setFormalCharge(0);
  mol.addAtom(a6);
  IAtom a7 = builder.newInstance(IAtom.class,"H");
  a7.setFormalCharge(0);
```

```
    mol.addAtom(a7);
    IAtom a8 = builder.newInstance(IAtom.class,"H");
    a8.setFormalCharge(0);
    mol.addAtom(a8);
    IAtom a9 = builder.newInstance(IAtom.class,"H");
    a9.setFormalCharge(0);
    mol.addAtom(a9);
    IBond b1 = builder.newInstance(IBond.class,a1, a2, IBo...
    nd.Order.SINGLE);
    mol.addBond(b1);
    IBond b2 = builder.newInstance(IBond.class,a1, a3, IBo...
    nd.Order.DOUBLE);
    mol.addBond(b2);
    IBond b3 = builder.newInstance(IBond.class,a1, a4, IBo...
    nd.Order.SINGLE);
    mol.addBond(b3);
    IBond b4 = builder.newInstance(IBond.class,a1, a5, IBo...
    nd.Order.SINGLE);
    mol.addBond(b4);
    IBond b5 = builder.newInstance(IBond.class,a2, a9, IBo...
    nd.Order.SINGLE);
    mol.addBond(b5);
    IBond b6 = builder.newInstance(IBond.class,a4, a6, IBo...
    nd.Order.SINGLE);
    mol.addBond(b6);
    IBond b7 = builder.newInstance(IBond.class,a4, a7, IBo...
    nd.Order.SINGLE);
    mol.addBond(b7);
    IBond b8 = builder.newInstance(IBond.class,a4, a8, IBo...
    nd.Order.SINGLE);
    mol.addBond(b8);
}
```

5.7. Line Notations

Another common input mechanism in cheminformatics is the line notation. Several line notations have been proposed, including the WiswesserLineNotation (WLN) [4] and the SybylLineNotation (SLN) [5], but the most popular is SMILES [6]. There is a Open Standard around this format called OpenSMILES, available at `http://www.opensmiles.org/`.

5.7.1. SMILES

The CDK can both read and write SMILES, or at least a significant subset of the line notation. You can parse a SMILES into a IAtomContainer with the SmilesParser. The constructor of the parser takes an IChemObjectBuilder because it needs to know what CDK interface implementation it must use

to create classes. This example uses the DefaultChemObjectBuilder:

Script 5-17: code/ReadSMILES.groovy

```
sp = new SmilesParser(
  DefaultChemObjectBuilder.getInstance()
)
mol = sp.parseSmiles("CC(=O)OC1=CC=CC=C1C(=O)O")
println "Aspirin has ${mol.atomCount} atoms."
```

Telling us the number of (non-hydrogen) atoms in aspirin:

```
Aspirin has 13 atoms.
```

Writing of SMILES goes in a similar way. But I do like to point out that by default the SMILESGenerator does not use the convention to use lower case element symbols for aromatic atoms. To trigger that, you should use the setUseAromaticityFlag method:

Script 5-18: code/WriteSMILES.groovy

```
generator = new SmilesGenerator()
mol = MoleculeFactory.makePhenylAmine()
smiles = generator.createSMILES(mol)
println "Ph-NH2 -> $smiles"
generator.setUseAromaticityFlag(true);
smiles = generator.createSMILES(mol)
println "Ph-NH2 -> $smiles"
```

showing the different output without and with that option set:

```
Ph-NH2 -> NC1=CC=CC=C1
Ph-NH2 -> Nc1ccccc1
```

Of course, this does require that aromaticity has been perceived, as explained in Section 11.4.

References

[1] P. Murray-Rust, H. S. Rzepa, Chemical Markup, XML, and the Worldwide Web. 1. Basic Principles, *J. Chem. Inf. Model.* **1999**, *39*, 928–942.

[2] E. L. Willighagen, Processing CML conventions in Java, *Internet Journal of Chemistry* **2001**, *4*, 4+.

[3] A. Dalby, J. G. Nourse, W. D. Hounshell, A. K. I. Gushurst, D. L. Grier, B. A. Leland, J. Laufer, Description of several chemical structure file formats used by computer programs developed at Molecular Design Limited, *Journal of Chemical Information and Computer Sciences* **1992**, *32*, 244–255.

[4] W. J. Wiswesser, How the WLN began in 1949 and how it might be in 1999, *Journal of Chemical Information and Computer Sciences* **1982**, *22*, 88–93.

[5] W. R. Homer, J. Swanson, R. J. Jilek, T. Hurst, R. D. Clark, SYBYL Line Notation (SLN): A Single Notation To Represent Chemical Structures, Queries, Reactions, and Virtual Libraries, *Journal of Chemical Information and Modeling* **2008**, *48*, 2294–2307.

[6] D. Weininger, SMILES, a chemical language and information system. 1. introduction to methodology and encoding rules, *J. Chem. Inf. Comput. Sci.* **1988**, *28*, 31–36.

6. Atom types

Graph theory is nice, but we are, of course, interested in chemistry. While graph theory has its limitations, we can do a lot of interesting things with just the vertex-edge formalism. Particularly, if we combine it with the concept of atom types.
An atom type is a concept to describe certain properties of the atom. For example, force fields use atom types to describe geometrical and interaction properties of the atoms in a molecule. Within such formalism, a sp^3 carbon is a carbon with four neighbors organized in a tetrahedral coordination, as depicted in Figure 6.1.

6.1. The CDK atom type model

A complete description for the atom types of the following atomic properties is needed by the various algorithms in the CDK:

- element
- formal charge
- number of bonded neighbors
- hybridization (sp^3, sp^2, sp, etc)
- number of lone pairs
- number of π bonds

For example, the carbon in methane, we can list these properties with this code:

Script 6-1: code/CDKAtomTypeProperties.groovy

```
factory = AtomTypeFactory.getInstance(
  "org/openscience/cdk/dict/data/cdk-atom-types.owl",
  NoNotificationChemObjectBuilder.getInstance()
);
IAtomType type = factory.getAtomType("C.sp3");
```

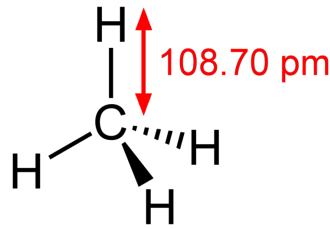

Figure 6.1.: 3D structure of methane, showing a sp^3 carbon surrounded by four hydrogens. Image from Wikipedia: `http://en.wikipedia.org/wiki/File:Methane-CRC-MW-dimensions-2D.png` (public domain).

```
println "element       : $type.symbol"
println "formal change : $type.formalCharge"
println "hybridization : $type.hybridization"
println "neighbors     : $type.formalNeighbourCount"
println "lone pairs    : " +
  type.getProperty(CDKConstants.LONE_PAIR_COUNT)
println "pi bonds      : " +
  type.getProperty(CDKConstants.PI_BOND_COUNT)
```

We will see the carbon has these properties:

```
element       : C
formal change : 0
hybridization : SP3
neighbors     : 4
lone pairs    : 0
pi bonds      : 0
```

For a carbon in benzene (*C.sp2*), it would list:

```
element       : C
formal change : 0
hybridization : SP2
neighbors     : 3
lone pairs    : 0
pi bonds      : 1
```

And for the oxygen in hydroxide (*C.minus*), it would give:

```
element       : O
formal change : -1
hybridization : SP3
neighbors     : 1
lone pairs    : 3
pi bonds      : 0
```

A full list of CDK atom types is given in a table in Appendix A.

6.1.1. Hybridization Types

The CDK knows about various hybridization types. Hybridizations are linear combinations of atomic orbitals and typically used to explain the orientation of atoms attached to the central atom. For example, Figure 6.1 showed one possible hybridization, sp^3.
The list of supported hybridization types can be listed with:

Script 6-2: code/HybridizationTypes.groovy

```
IAtomType.Hybridization.each {
  println it
}
```

listing these types:

```
S
SP1
SP2
SP3
PLANAR3
SP3D1
SP3D2
SP3D3
SP3D4
SP3D5
```

6.2. Atom type perception

Because so many cheminformatics algorithms depend on atom type information, determining the atom types of the atoms in a molecule is typically a very first step, after a molecule has been created. When the CDK is not able to recognize (*perceive*) the atom type, then this will most certainly

mean that the output of cheminformatics algorithms in undefined. The following two sections will describe how atom types can be perceived. It will also be shown what happens when the atom type cannot be recognized.

6.2.1. Single atoms

Instead of perceiving atom types for all atoms in the molecule, one may also perceive the type of a single atom. The former is more efficient when types need to be perceived for all atoms, but when the molecule only partly changed, it can be worthwhile to only perceive atom types for only the affected atoms:

Script 6-3: code/AtomTypePerception.groovy

```
molecule = new Molecule();
atom = new Atom(Elements.CARBON);
molecule.addAtom(atom);
matcher = CDKAtomTypeMatcher.getInstance(
  DefaultChemObjectBuilder.getInstance()
);
type = matcher.findMatchingAtomType(molecule, atom);
AtomTypeManipulator.configure(atom, type);
println "Atom type: $type.atomTypeName"
```

This reports the perceived atom type for the carbon:

```
Atom type: C.sp3
```

6.2.2. Full molecules

Because atom type perception requires the notion of ring systems, with each atom type being perceived individually, using the above approach ring detection must be done each time the atom type is perceived for each atom[1]. Therefore, perceiving atom types for all atoms in a molecule can be done more efficiently with the following code:

Script 6-4: code/AtomTypePerceptionMolecule.groovy

```
matcher = CDKAtomTypeMatcher.getInstance(
  DefaultChemObjectBuilder.getInstance()
);
type = matcher.findMatchingAtomType(molecule);
```

[1] Theoretically, this information can be cached, but there currently is no suitable solution for this in the CDK.

6.2.3. Configuring the Atom

We saw earlier how the AtomTypeManipulator class was used to configure an atom with the configure(IAtom, IAtomType) method. This class also has a convenience method to perceive and configure all atoms in a molecule with one call:

Script 6-5: code/AtomTypePerceptionAndConfigure.groovy

```
AtomContainerManipulator
  .percieveAtomTypesAndConfigureAtoms(molecule);
```

6.3. Sybyl atom types

The Sybyl atom type list is well-known for its application in then mol2 file format (see the Mol2Format class) and used in force fields [1]. Sybyl atom types can be perceived with the SybylAtomTypeMatcher class, which perceives CDK atom types and then translates this in to Sybyl atom types:

Script 6-6: code/SybylAtomTypePerception.groovy

```
molecule = new Molecule();
atom = new Atom(Elements.CARBON);
molecule.addAtom(atom);
matcher = SybylAtomTypeMatcher.getInstance(
  DefaultChemObjectBuilder.getInstance()
);
type = matcher.findMatchingAtomType(molecule, atom);
AtomTypeManipulator.configure(atom, type);
println "Atom type: $type.atomTypeName"
```

This will give you the Sybyl atom type for carbon in methane:

```
Atom type: C.3
```

A full list of Sybyl atom types is given in a table in Appendix A.

References

[1] M. Clark, R. D. Cramer, N. Van Opdenbosch, Validation of the general purpose tripos 5.2 force field, *J. Comput. Chem.* **1989**, *10*, 982–1012.

7. Graph Properties

Graph theory is the most common representation in cheminformatics, and with quantum mechanics, rule the informatics side of chemistry. The molecular graph follow graph theory and defines atoms as molecules and bonds as edge between to atoms. This is by far the only option, and the IBond allows for more complex representations, but we will focus on the molecular graph in this chapter.

7.1. Partitioning

If one is going to calculate graph properties, the first thing one often has to do, is to split ensure that one is looking at a fully connected graph. Since this is often in combination with ensuring fully connected graphs, the ConnectivityChecker is a welcome tool. It allows partitioning of the atoms and bonds in an IAtomContainer into molecules, organized into IMoleculeSet:

Script 7-1: code/ConnectivityCheckerDemo.groovy

```
atomCon = new AtomContainer();
atom1 = new Atom("C");
atom2 = new Atom("C");
atomCon.addAtom(atom1);
atomCon.addAtom(atom2);
moleculeSet = ConnectivityChecker.partitionIntoMolecules(
  atomCon
);
println "Number of isolated graphs: " +
  moleculeSet.moleculeCount
```

Which gives:

```
Number of isolated graphs: 2
```

7.2. Spanning Tree

The spanning tree of a graph, is subgraph with no cycles; that spans all atoms into a, still, fully connected graph:

Script 7-2: code/SpanningTreeBondCount.groovy

```
println "Number of azulene bonds: $azulene.bondCount"
treeBuilder = new SpanningTree(azulene)
azuleneTree = treeBuilder.getSpanningTree();
println "Number of tree bonds: $azuleneTree.bondCount"
```

which returns:

```
Number of azulene bonds: 11
Number of tree bonds: 9
```

As a side effect, it also determines which bonds are ring bonds, and which are not:

Script 7-3: code/SpanningTreeRingBonds.groovy

```
ethaneTree = new SpanningTree(ethane)
println "[ethane]"
println "Number of cyclic bonds: " +
  ethaneTree.bondsCyclicCount
println "Number of acyclic bonds: " +
  ethaneTree.bondsAcyclicCount
azuleneTree = new SpanningTree(azulene)
println "[azulene]"
println "Number of cyclic bonds: " +
  azuleneTree.bondsCyclicCount
println "Number of acyclic bonds: " +
  azuleneTree.bondsAcyclicCount
```

giving

```
[ethane]
Number of cyclic bonds: 0
Number of acyclic bonds: 1
[azulene]
Number of cyclic bonds: 11
Number of acyclic bonds: 0
```

7.3. Graph matrices

Chemical graphs have been very successfully used as representations of molecular structures, but are not always to most suitable representation.

For example, for computation of graph properties often a matrix representation is used as intermediate step. The CDK has predefined helper classes to calculate two kind of graphmatrices: the adjacency matrix and the distance matrix. Both are found in the cdk.graph.matrix package.

7.3.1. Adjacency matrix

The adjacencymatrix describes which atoms are connected via a covalent bond. All matrix elements that link to bonded atoms are 1, and those matrix elements for disconnected atoms are 0. In mathematical terms, the adjacency matrix A is defined as:

$$A_{i,j} = \begin{cases} 0 & \text{if } i = j \\ 0 & \text{if atoms } i \text{ and } j \text{ are not bonded} \\ 1 & \text{if atoms } i \text{ and } j \text{ are bonded} \end{cases} \tag{7.1}$$

The algorithm to calculate this matrix is implemented in the AdjacencyMatrix class. The matrix is calculated with the static getMatrix(IAtomContainer) method:

Script 7-4: code/AdjacencyMatrixCalc.groovy

```
int[][] matrix = AdjacencyMatrix.getMatrix(ethanoicAcid)
for (row=0;row<ethanoicAcid.getAtomCount();row++) {
  for (col=0;col<ethanoicAcid.getAtomCount();col++) {
    print matrix[row][col] + " "
  }
  println ""
}
```

This code outputs the matrix, resulting for ethanoic acid, with the atoms in the order C, C, O, and O, in:

```
0 1 0 0
1 0 1 1
0 1 0 0
0 1 0 0
```

7.3.2. Distance matrix

The distance matrix describes the number of bonds on has to traverse to get from one atom to another. Therefore, it has zeros on the diagonal and non-zero values at all other locations. Matrix elements for neighboring atoms are 1 and others are larger. The CDK uses Floyd'salgorithm to calculate this matrix [1], which is exposed via the TopologicalMatrix class:

Script 7-5: code/DistanceMatrix.groovy

```
int[][] matrix = TopologicalMatrix.getMatrix(ethanoicAcid)
```

For the ethanoic acid used earlier, the resulting matrix looks like:

```
0 1 2 2
1 0 1 1
2 1 0 2
2 1 2 0
```

7.4. Atom Numbers

Another important aspect of the chemical graph, is that the graph uniquely places atoms in the molecule. That is, the graphs allows us to uniquely identify, and therefore, number atoms in the molecule. This is an important aspect of cheminformatics, and the concept behind canonicalization, such as used to create canonicalSMILES. The InChI library (see Chapter 12) implements such an algorithm, and we can use it to assign unique integers to all atoms in a chemical graph.

7.4.1. Morgan Atom Numbers

Morgan published an algorithm in 1965 to assign numbers to vertices in the chemical graph [2]. The algorithm does not take into account the element symbols associated with those vertices, and it only based on the connectivity. Therefore, we see the same number of symmetry related atoms, even if they have different symbols. If we run:

Script 7-6: code/MorganAtomNumbers.groovy

```
oxazole = MoleculeFactory.makeOxazole();
long[] morganNumbers =
  MorganNumbersTools.getMorganNumbers(
    oxazole
  );
for (i in 0..(oxazole.atomCount-1)) {
  println oxazole.getAtom(i).symbol +
    " " + morganNumbers[i]
}
```

we see this output:

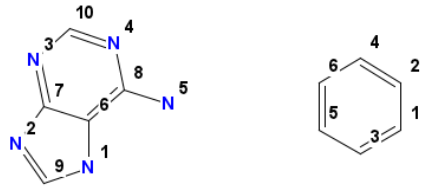

Figure 7.1.: InChI atom numbers of oxazole (left) and benzene (right). This figure was made with the code given in Section 9.3.

```
C 64
O 64
C 64
N 64
C 64
```

7.4.2. InChI Atom Numbers

The InChI library does not have a direct method to calculate atom numbers from Java, but the CDK can extract these from the auxiliary layer. These numbers are those listed in the bond layer, but to use these in the CDK molecule class, we need to mapping of the InChIatomnumbers This method is made available via the InChINumbersTools class:

Script 7-7: code/InChIAtomNumbers.groovy

```
oxazole = MoleculeFactory.makeOxazole();
long[] morganNumbers =
  InChINumbersTools.getNumbers(
    oxazole
  );
for (i in 0..(oxazole.atomCount-1)) {
  atom = oxazole.getAtom(i)
  println atom.symbol +
    " " + morganNumbers[i]
}
```

which outputs:

```
C 2
O 5
C 3
N 4
C 1
```

It is important to note that because these numbers are used in the connectivity layer, symmetry is broken in assignment of these numbers, allowing all atoms in, for example, benzene to still be uniquely identified:

Script 7-8: code/InChIAtomNumbersBenzene.groovy

```
benzene = MoleculeFactory.makeBenzene();
long[] numbers =
  InChINumbersTools.getNumbers(
    benzene
  );
for (i in 0..(benzene.atomCount-1)) {
  atom = benzene.getAtom(i)
  atom.setProperty(
    "AtomNumber",
    "" + numbers[i]
  )
}
```

which outputs:

```
C 1
C 2
C 4
C 6
C 5
C 3
```

The InChI atom numbers are shown in Figure 7.1.

References

[1] R. W. Floyd, Algorithm 97: Shortest path, *Commun. ACM* **1962**, *5*, 345+.

[2] H. L. Morgan, The Generation of a Unique Machine Description for Chemical Structures-A Technique Developed at Chemical Abstracts Service. *Journal of Chemical Documentation* **1965**, *5*, 107–113.

8. Missing Information

Missing information is common place in chemical file formats and line notations. In many cases this information is implicit to the representation, but recovering it is not always easy, requiring assumptions which may not be true. Examples of missing informations is the lack of bonds in XYZ files, and the removed double bond location information for aromatic ring systems.

8.1. Reconnecting Atoms

XYZ files do not have bond information, and may look like:

```
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
```

Fortunately, we can reasonably assume bonds to have a certain length, and reasonably understand how many connections and atom can have at most. Then, using the 3D coordinate information available from the XYZ file, an algorithm can deduce how the atoms must be bonded. The RebondTool does exactly that. And, it does it efficiently too, using a binary search tree, which allows it to scale to protein-sized molecules.
Now, the algorithm does need to know what reasonable bond lengths are, and for this we can use the Jmol list of covalent radii, and we configure the atoms accordingly:

Script 8-1: code/CovalentRadii.groovy

```
methane = new Molecule();
methane.addAtom(new Atom("C", new Point3d(0.0, 0.0, 0.0)));
methane.addAtom(new Atom("H", new Point3d(0.6, 0.6, 0.6)));
methane.addAtom(new Atom("H", new Point3d(-0.6,-0.6,0.6)));
```

```
methane.addAtom(new Atom("H", new Point3d(0.6,-0.6,-0.6)));
methane.addAtom(new Atom("H", new Point3d(-0.6,0.6,-0.6)));
factory = AtomTypeFactory.getInstance(
  "org/openscience/cdk/config/data/jmol_atomtypes.txt",
  methane.getBuilder()
);
for (IAtom atom : methane.atoms()) {
  factory.configure(atom);
  println "$atom.symbol -> $atom.covalentRadius"
}
```

which configures and prints the atoms' radii:

```
C -> 0.77
H -> 0.32
H -> 0.32
H -> 0.32
H -> 0.32
```

Then the RebondTool can be used to rebind the atoms:

Script 8-2: code/RebondToolDemo.groovy

```
RebondTool rebonder = new RebondTool(2.0, 0.5, 0.5);
rebonder.rebond(methane);
println "Bond count: $methane.bondCount"
```

The number of bonds it found are reported in the last line:

```
Bond count: 4
```

8.2. Missing Hydrogens

The CDKHydrogenAdder class can be used to add missing hydrogens. The algorithm itself adds implicit hydrogens (see Section 2.4), but we will see how these can be converted into explicit hydrogens. The hydrogen adding algorithm expects, however, that CDK atom types are already perceived (see Section 6.2).

8.2.1. Implicit Hydrogens

Hydrogens that are not vertices in the molecular graph are called implicit hydrogens.They are merely a property of the atom to which they are connected. If these values are not given, which is common in for example SMILES, they can be (re)calculated with:

Script 8-3: code/MissingHydrogens.groovy

```
adder = CDKHydrogenAdder.getInstance(
  DefaultChemObjectBuilder.getInstance()
);
adder.addImplicitHydrogens(molecule);
println "Atom count: $molecule.atomCount"
println "Implicit hydrogens: $newAtom.hydrogenCount"
```

which reports:

```
Atom count: 1
Implicit hydrogens: 4
```

8.2.2. Explicit Hydrogens

These implicit hydrogens can be converted into explicit hydrogens using the following code:

Script 8-4: code/ExplicitHydrogens.groovy

```
adder.addImplicitHydrogens(molecule);
println "Atom count: $molecule.atomCount"
println " .. adding explicit hydrogens .."
AtomContainerManipulator.convertImplicitToExplicitHydrogens(
  molecule
);
println "Atom count: $molecule.atomCount"
```

which reports for the running methane example:

```
Atom count: 1
 .. adding explicit hydrogens ..
Atom count: 5
```

8.3. 2D Coordinates

Another bit of information missing from the input is often 2D coordinates. To generate 2D coordinate, the StructureDiagramGenerator can be used:

Script 8-5: code/Layout.groovy

```
butanol = smilesParser.parseSmiles("CCC(O)C")
sdg = new StructureDiagramGenerator();
```

```
sdg.setMolecule(butanol);
sdg.generateCoordinates(new Vector2d(0, 1));
butanol = sdg.getMolecule();
for (atom in butanol.atoms()) {
  println atom.getSymbol() + ": " +
    atom.getPoint2d()
}
```

which will generate the coordinate starting with an initial direction:

```
C: (0.0, 0.0)
C: (0.0, 1.5)
C: (-1.299038105676657, 2.2500000000000018)
O: (-1.2990381056766538, 3.7500000000000018)
C: (-2.598076211353316, 1.500000000000004)
```

8.4. Unknown Molecular Formula

Mass spectrometry (MS) is a technology where the experiment yields monoisotopic masses for molecules. In order to analyze these further, it is common to convert them to molecular formula. The MassToFormulaTool has functionality to determine these missing formulae. Miguel Rojas-Chertó developed this code for use in metabolomics [1]. Basic usage looks like:

Script 8-6: code/MissingMF.groovy

```
tool = new MassToFormulaTool(
  NoNotificationChemObjectBuilder.getInstance()
)
mfSet = tool.generate(133.0968);
for (mf in mfSet) {
  println MolecularFormulaManipulator.getString(mf)
}
```

This will create a long list of possible molecular formula. It is important to realize that it looks only at what molecular formula are possible with respect to the corresponding mass. This means that it will include chemically unlikely molecular formulae:

```
C3H11N5O
C5H13N2O2
C2H15NO5
CH9N8
```

```
H13N4O4
C10H13
C9H11N
CH15N3O4
C6H13O3
C2H11N7
C4H11N3O2
C4H13N4O
C2H9N6O
C6H15NO2
CH13N2O5
H7N9
C8H9N2
H15N5O3
C5H11NO3
C3H13N6
C3H9N4O2
C5H15N3O
C2H13O6
CH7N7O
H11N3O5
C9H9O
C7H7N3
C4H9N2O3
C4H15N5
C2H7N5O2
CH11NO6
H5N8O
C8H7NO
C6H5N4
C5H9O4
C3H7N3O3
CH5N6O2
```

This is overcome by setting restrictions. For example, we can put restrictions on the number of elements we allow in the matched formulae:

Script 8-7: code/MissingMFRestrictions.groovy

```
rules = new ArrayList<IRule>();
restriction  = new ElementRule();
MolecularFormulaRange range = new MolecularFormulaRange();
range.addIsotope( ifac.getMajorIsotope("C"), 8, 20);
```

```
range.addIsotope( ifac.getMajorIsotope("H"), 0, 20);
range.addIsotope( ifac.getMajorIsotope("O"), 0, 1);
range.addIsotope( ifac.getMajorIsotope("N"), 0, 1);
params = new Object[1];
params[0] = range;
restriction.setParameters(params);
rules.add(restriction);
tool.setRestrictions(rules);
```

Now the list looks more chemical:

```
C10H13
C9H11N
C9H9O
C8H7NO
C7H19NO
```

References

[1] M. Rojas-Chertó, P. T. Kasper, E. L. Willighagen, R. Vreeken, T. Hankemeier, T. Reijmers, Elemental Composition determination based on MSn, *Bioinformatics* **2011**, DOI 10.1093/bioinformatics/btr409.

9. Depiction

The CDK originates from the merger of Jmol and JChemPaint [1]. As such, CDK has long contained code to depict molecules. However, after the 1.0 series, a rewrite of the code base was initiated, causing the CDK 1.2 series to not be available with rendering functionality. During the development of the 1.4 series, the rendering code became gradually available as a set of patches, and, separately, as a JChemPaint applet. The new rendering code has entered the CDK.
However, if you need rendering of reaction schemes or the editing functionality found in JChemPaint, you still need the CDK-JChemPaint patch.

9.1. Molecules

Rendering molecules to an image is done in a few steps. First, an Image needs to be defined, for example, of 200 by 200 pixels. The next step is to define what is to be generated, and how. The most basic rendering requires a few generators: one for the overall scene, one for atoms, and one for bonds. Therefore, we add a BasicSceneGenerator, a BasicAtomGenerator, and a BasicBondGenerator. We will see later that we can add further generators to add further visualization. Now that we defined what we want to have depicted, we construct a renderer. Because we are rendering a molecule here, we simply use the AtomContainerRenderer.

Figure 9.1.: 2D diagram of triazole.

We also need to define, however, what rendering platform we want to use. The Java community has a few options, with the AWT/Swing platform to

be the reference implementation provided by Oracle, and the SWT toolkit as a popular second. In fact, the redesign was needed to be able to support both widget toolkits. For rendering images, we can use the AWT toolkit. Therefore, we use a AWTFontManager to help the renderer draw texts. We get our Graphics2D object to which will be drawn from the earlier created image, and we set some basic properties. Then we are ready to draw the molecule to the graphics object with the paint() method, and here again we need a AWT-specific class: the AWTDrawVisitor.
What then remains is to save the image to a PNG image file with the ImageIO helper class.
The full code example then looks like:

Script 9-1: code/RenderMolecule.groovy

```
int WIDTH = 200;
int HEIGHT = 200;
// the draw area and the image should be the same size
Rectangle drawArea = new Rectangle(WIDTH, HEIGHT);
Image image = new BufferedImage(
  WIDTH, HEIGHT, BufferedImage.TYPE_INT_RGB
);
// generators make the image elements
List<IGenerator> generators = new ArrayList<IGenerator>();
generators.add(new BasicSceneGenerator());
generators.add(new BasicBondGenerator());
generators.add(new BasicAtomGenerator());
// the renderer needs to have a toolkit-specific font manager
AtomContainerRenderer renderer =
  new AtomContainerRenderer(generators, new AWTFontManager());
// the call to 'setup' only needs to be done on the first paint
renderer.setup(triazole, drawArea);
// paint the background
Graphics2D g2 = (Graphics2D)image.getGraphics();
g2.setColor(Color.WHITE);
g2.fillRect(0, 0, WIDTH, HEIGHT);
// the paint method also needs a toolkit-specific renderer
renderer.paint(triazole, new AWTDrawVisitor(g2));
ImageIO.write(
  (RenderedImage)image, "PNG",
  new File("RenderMolecule.png")
);
```

This results in the image of triazole given in Figure 9.1.

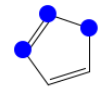

Figure 9.2.: The atoms symbols are replaced by squares using the CompactAtom rendering parameter.

9.2. Parameters

Rendering wasn't as much fun, if you could not tune it to your needs. JChemPaint has long had many rendering parameters, which are now all converting to the new API. The following code is an modification of the code example in snippet 9-1, and adds some code to list all rendering parameters for the three used generators:

Script 9-2: code/RendererParameters.groovy

```
// generators make the image elements
List<IGenerator> generators = new ArrayList<IGenerator>();
generators.add(new BasicSceneGenerator());
generators.add(new BasicBondGenerator());
generators.add(new BasicAtomGenerator());
// the renderer needs to have a toolkit-specific font manager
AtomContainerRenderer renderer =
  new AtomContainerRenderer(generators, new AWTFontManager());
// dump all parameters
for (generator in renderer.generators) {
  for (parameter in generator.parameters) {
    println "parameter: " +
      parameter.class.name.substring(40) +
      " -> " +
      parameter.value;
  }
}
```

The output will look something like:

```
parameter: BasicSceneGenerator$BackgroundColor -> java.a...
  wt.Color[r=255,g=255,b=255]
```

```
parameter: BasicSceneGenerator$ForegroundColor -> java.a...
  wt.Color[r=0,g=0,b=0]
parameter: BasicSceneGenerator$Margin -> 10.0
parameter: BasicSceneGenerator$UseAntiAliasing -> true
parameter: BasicSceneGenerator$UsedFontStyle -> NORMAL
parameter: BasicSceneGenerator$FontName -> Arial
parameter: BasicSceneGenerator$ZoomFactor -> 1.0
parameter: BasicSceneGenerator$Scale -> 1.0
parameter: BasicSceneGenerator$FitToScreen -> false
parameter: BasicSceneGenerator$ShowMoleculeTitle -> false
parameter: BasicSceneGenerator$ShowTooltip -> false
parameter: BasicBondGenerator$BondWidth -> 1.0
parameter: BasicBondGenerator$DefaultBondColor -> java.a...
  wt.Color[r=0,g=0,b=0]
parameter: BasicBondGenerator$BondLength -> 40.0
parameter: BasicBondGenerator$WedgeWidth -> 2.0
parameter: BasicBondGenerator$BondDistance -> 2.0
parameter: BasicBondGenerator$TowardsRingCenterProportio...
  n -> 0.15
parameter: BasicAtomGenerator$AtomColor -> java.awt.Colo...
  r[r=0,g=0,b=0]
parameter: BasicAtomGenerator$AtomColorer -> org.opensci...
  ence.cdk.renderer.color.CDK2DAtomColors@9300cc
parameter: BasicAtomGenerator$AtomRadius -> 8.0
parameter: BasicAtomGenerator$ColorByType -> true
parameter: BasicAtomGenerator$CompactShape -> SQUARE
parameter: BasicAtomGenerator$CompactAtom -> false
parameter: BasicAtomGenerator$KekuleStructure -> false
parameter: BasicAtomGenerator$ShowEndCarbons -> false
parameter: BasicAtomGenerator$ShowExplicitHydrogens -> t...
  rue
```

Of course, the idea is that you can override default parameter values. That way you can tune the output to your particular needs. An example use case is when a diagram gets smaller and the element symbols would become unreadable. Then you can choose to draw the non-carbon atoms as colored filled circles. To achieve this, we only need to change the CompactAtom and CompactShape parameters from the BasicAtomGenerator as listed in the above output.

We set parameter and extend our first example:

Script 9-3: code/CompactAtomParam.groovy

```
model = renderer.getRenderer2DModel();
model.set(CompactAtom.class, true);
model.set(CompactShape.class, Shape.OVAL);
```

The new output is given in Figure 9.2.

9.3. Generators

We saw earlier that *generators* are used to convert a chemical graph into a depiction. These generators implement the IGenerator interface. This interface is using Java generics, and looks something like:

```
public interface IGenerator<T extends IChemObject>  {
  public List<IGeneratorParameter<?>> getParameters();
  public IRenderingElement generate(
    T object, RendererModel model
  );
}
```

This means, if we extend the IGenerator¡IAtomContainer, the implementation is expected to provide the method generate(IAtomContainer object, RendererModel model). Thus, we can create a class to depict atom numbers with:

Script 9-4: code/AtomNumberGenerator.java

```
public class AtomNumberGenerator
implements IGenerator<IAtomContainer> {
  public IRenderingElement generate(
    IAtomContainer ac, RendererModel model
  ) {
    ElementGroup numbers = new ElementGroup();
    Vector2d offset = new Vector2d(0.5,0.5);
    for (IAtom atom : ac.atoms()) {
      Point2d p = new Point2d(atom.getPoint2d());
      p.add( offset );
      numbers.add(
        new TextElement(
          p.x, p.y,
          (String)atom.getProperty("AtomNumber"),
          Color.BLACK
```

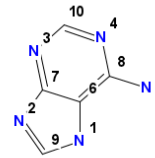

Figure 9.3.: 2D diagram of adenine with numbers atoms.

```
        )
      );
    }
    return numbers;
  }
  public List<IGeneratorParameter<?>> getParameters() {
    return Collections.emptyList();
  }
}
```

We can add this generator in the same way as the other generator:

Script 9-5: code/RenderAdenineWithNumbers.groovy

```
generators.add(new BasicSceneGenerator());
generators.add(new BasicBondGenerator());
generators.add(new BasicAtomGenerator());
generators.add(new AtomNumberGenerator());
```

And get a depiction as shown in Figure 9.3.

References

[1] S. Krause, E. L. Willighagen, C. Steinbeck, JChemPaint - Using the Collaborative Forces of the Internet to Develop a Free Editor for 2D Chemical Structures, *Molecules* **2000**, *5*, 93–98.

10. Substructure Searching

The UniversalIsomorphismTester class in the CDK can be used for substructure searching. It allows you to determine if some structure is a substructure and what the matching substructures are. As such, this can also be used to determine if two structures are identical.
In this chapter we will see how the class returns all possible substructure matches, and we'll notice that redundancy occurs due to symmetrically equivalent matches, and how these redundant matches can be removed.

10.1. Exact Search

The UniversalIsomorphismTester class implements an algorithm that was originally developed for isomorphism checking. However, it can be used for substructure search too. This section will first show how the class is used to check if two classes are identical:

Script 10-1: code/Isomorphism.groovy

```
butane = MoleculeFactory.makeAlkane(4);
println "Is isomorphic: " +
  UniversalIsomorphismTester.isIsomorph(
    butane, butane
  )
```

The step to substructure searching is made via the isSubgraph() method:

Script 10-2: code/IsSubgraph.groovy

```
butane = MoleculeFactory.makeAlkane(4);
propane = MoleculeFactory.makeAlkane(3);
println "Propane part of Butane: " +
  UniversalIsomorphismTester.isSubgraph(
    butane, propane
  )
println "Butane part of Propane: " +
  UniversalIsomorphismTester.isSubgraph(
```

```
    propane, butane
  )
```

It gives this output:

```
Propane part of Butane: true
Butane part of Propane: false
```

Now, you may wonder why propane is a subgraph of butane, because it is indeed not. This code is taking advantage of the fact that the factory returns hydrogen depleted graphs (see Section 2.4). Therefore, butane is a chain of four carbons, and propane is a chain of three carbons. Then, the latter is a chemical subgraph of the former.

10.2. Matching Substructures

Substructure searching is finding in a target molecule the atoms that match the given searched substructure. With the UniversalIsomorphismTester we can do:

Script 10-3: code/Overlap.groovy

```
butane = MoleculeFactory.makeAlkane(4)
ccc = MoleculeFactory.makeAlkane(3)
hits = UniversalIsomorphismTester
  .getOverlaps(
     butane, ccc
  )
println "Number of hits: " + hits.size
hits.each { substructure ->
  println "Substructure in Molecule:"
  println "  #atoms: " + substructure.atomCount
}
```

However, this only returns us one match, selected as being the largest:

```
Number of hits: 1
Substructure in Molecule:
  #atoms: 3
```

There is an alternative:

Script 10-4: code/Substructure.groovy

```
butane = MoleculeFactory.makeAlkane(4);
ccc = MoleculeFactory.makeAlkane(3);
hits = UniversalIsomorphismTester
  .getSubgraphAtomsMaps(
     butane, ccc
  )
println "Number of hits: " + hits.size
hits.each { substructure ->
  println "Atoms in substructure: " +
    substructure.size
}
```

The getSubgraphAtomsMaps() methods returns a List⟨List⟨RMap⟩⟩ object, where each List⟨RMap⟩ represents on substructure match. When we look at the outer list, we see that the subgraph of three carbon atoms is found 4 times in butane, each with 3 atoms:

```
Number of hits: 4
Atoms in substructure: 3
Atoms in substructure: 3
Atoms in substructure: 3
Atoms in substructure: 3
```

This is caused by the symmetrical nature of the substructure. It can map twice onto the same three atoms in butane: once in the forward direction, and once in the backward direction.

10.3. Fingerprints

Substructure searching is a relatively slow algorithm, and the time required to compare two molecules scales with the number of atoms in each molecule. To reduce the computation time, molecular fingerprints were invented. There are two key aspects to fingerprints that make them efficient: first, they have a fixed length so that the time to compare two molecule is independent of the size of the two structures; secondly, the fingerprint of a substructure always matches the fingerprint of any molecules that has that substructure.

In this section we will see two fingerprint types available in the CDK: a substructure based fingerprint, and a patch based fingerprint. Before I will explain how these fingerprints are created, we will first look at the BitSet class that is used by the CDK to represent these fingerprints. Consider this code:

Script 10-5: code/BitSetDemo.groovy

```
bitset = new BitSet(10);
println "Empty bit set: $bitset";
bitset.set(3);
bitset.set(7);
println "Two bits set: $bitset";
```

If we analyze the output, we see that all set bits are listed, and that all other bits are not:

```
Empty bit set: {}
Two bits set: {3, 7}
```

Let us now consider a simple substructure fingerprint of length four with the following bit definitions:

- bit 1: molecule contains a carbon
- bit 2: molecule contains a nitrogen
- bit 3: molecule contains a oxygen
- bit 4: molecule contains a chlorine

Let's call this fingerprinter SimpleFingerprinter:

Script 10-6: code/SimpleFingerprinter.java

```
public class SimpleFingerprinter implements IFingerprinter {
  Map<String,Integer> map = new HashMap<String,Integer>() {{
    put("C", 1);
    put("N", 2);
    put("O", 3);
    put("Cl", 4);
  }};
  public BitSet getFingerprint(IAtomContainer molecule) {
    BitSet bitSet = new BitSet(getSize());
    for (IAtom atom : molecule.atoms()) {
      if (map.containsKey(atom.getSymbol()))
        bitSet.set(map.get(atom.getSymbol()));
    }
    return bitSet;
  }
  public int getSize() {
```

```
      return 4;
    }
  }
```

We can then calculate the fingerprints for ethanol and benzene:

Script 10-7: code/SimpleFingerprintDemo.groovy

```
fingerprinter = new SimpleFingerprinter();
println "ethanol: " + fingerprinter.getFingerprint(ethanol)
println "benzene: " + fingerprinter.getFingerprint(benzene)
```

and we get these bit sets:

```
ethanol: {1, 3}
benzene: {1}
```

Now, we can replace the presence of a particular atom, by the presence of a substructure, such as a phenyl or a carbonyl group. We have then defined a substructure fingerprint.
The CDK has several kinds of fingerprints, including path-based fingerprints (Fingerprinter and HybridizationFingerprinter), a MACSS fingerprint (MACSSFingerprinter) [1], and the PubChem fingerprint (PubChemFingerprinter). These fingerprints have been used for various tasks, including ligand classification [2], and databases like BRENDA [3] and TIN [4].

References

[1] J. L. Durant, B. A. Leland, D. R. Henry, J. G. Nourse, Reoptimization of MDL Keys for Use in Drug Discovery, *Journal of Chemical Information and Computer Sciences* **2002**, *42*, 1273–1280.

[2] C. Ma, L. Wang, X.-Q. Xie, Ligand Classifier of Adaptively Boosting Ensemble Decision Stumps (LiCABEDS) and Its Application on Modeling Ligand Functionality for 5HT-Subtype GPCR Families, *Journal of Chemical Information and Modeling* **2011**, *51*, 521–531.

[3] I. Schomburg, A. Chang, C. Ebeling, M. Gremse, C. Heldt, G. Huhn, D. Schomburg, BRENDA, the enzyme database: updates and major new developments, *NUCLEIC ACIDS RESEARCH* **2004**, *32*, DOI 10.1093/nar/gkh081.

[4] K. V. Dorschner, D. Toomey, M. P. Brennan, T. Heinemann, F. J. Duffy, K. B. Nolan, D. Cox, M. F. A. Adamo, A. J. Chubb, TIN A Combinatorial Compound Collection of Synthetically Feasible Multicomponent Synthesis Products, *Journal of Chemical Information and Modeling* **2011**, *51*, 986–995.

11. Molecular Properties

Cheminformatics is about molecular properties and chemistry in general the field of finding chemicals with new properties[1]. We keep databases to store those properties, and we develop methods to predict and understand those properties. Prediction is important for one reason: there are too many chemical structures and we cannot experimentally measure the properties for all of them. The number of molecules is often said to be relevant to drug discovery is in the order of 10^{60}. The largest current databases have less then 10^8 structures. This chapter will show how the CDK can be used to calculate a number of molecular properties.

11.1. Molecular Mass

The simplest but perhaps the most reported molecular property is the molecular mass. It is important to realize this mass is not constant, and depends on the natural mixture of isotopes, which is not constant itself. If you have an atom container with explicit hydrogens, you can loop over the atoms to calculate the molecular mass as summation of the masses of the individual atoms:

Script 11-1: code/CalculateMolecularWeight.groovy

```
molWeight = 0.0
for (atom in molecule.atoms()) {
  molWeight += isotopeInfo.getNaturalMass(atom)
}
```

In this case, you can also use the AtomContainerManipulator:

Script 11-2: code/CalculateMolecularWeightShort.groovy

```
molWeight = AtomContainerManipulator
  .getNaturalExactMass(molecule)
```

The element masses are calculated from the accurate isotope masses and natural abundances defined in the Blue Obelisk Data Repository [1].

[1] Prof. Gasteiger in 2006 gave a lecture at Cologne University where he expressed this view. It sticked around.

11.1.1. Implicit Hydrogens

If your atom container has implicithydrogens specified, you will have the above code will not be sufficient. Instead, your code should look like:

Script 11-3: code/CalculateMolecularWeightImplicitHydrogens.groovy

```
molWeight = 0.0
hWeight = isotopeInfo.getNaturalMass(Elements.HYDROGEN)
for (atom in molecule.atoms()) {
  molWeight += isotopeInfo.getNaturalMass(atom)
  if (atom.getImplicitHydrogenCount() != CDKConstants.UNSET)
    molWeight += atom.getImplicitHydrogenCount() *
                 hWeight
}
```

11.2. LogP

The partition coefficient describes how a molecular structure distributes itself over two immiscible solvents. The logarithm of the partition coefficient (LogP) between octanol and water is often used in cheminformatics to describe hydrophobicity [2, 3]. Wikipedia gives this equation [2]:

$$log\ P_{oct/wat} = log\left(\frac{[solute]_{octanol}}{[solute]^{un-ionized}_{water}}\right) \tag{11.1}$$

This equation assumes that the solute is neutral, which may involve changing the pH of the water.
The CDK has implemented an algorithm based on the XLogP algorithm [4, 5]. The code is available via the descriptor API. It can be used to calculate the LogP for a single molecule. The implementation expects explicit hydrogens, so you need to add those if not present yet (see Section 8.2). The calculation returns a DoubleResult following the descriptor API:

Script 11-4: code/XLogP.groovy

```
oxazone = MoleculeFactory.makeOxazole();
benzene = MoleculeFactory.makeBenzene();
// add explicit hydrogens ...
descriptor = new XLogPDescriptor()
println "LogP of oxazone: " +
```

[2] http://en.wikipedia.org/wiki/Partition_coefficient

```
    ((DoubleResult)descriptor.calculate(oxazone).getValue())
    .doubleValue()
println "LogP of benzene: " +
    ((DoubleResult) descriptor.calculate(benzene).getValue())
    .doubleValue()
```

which returns:

```
LogP of oxazone: -0.14800000000000002
LogP of benzene: 2.082
```

11.3. Total Polar Surface Area

Another properties that frequently returns in cheminformatics is the Total Polar Surface Area (TPSA). The code in the CDK uses an algorithm published by Ertl in 2000 [6]. Here too, the descriptor API is used, so that the code is quite similar to that for the logP calculation:

Script 11-5: code/TPSA.groovy

```
oxazone = MoleculeFactory.makeOxazole();
benzene = MoleculeFactory.makeBenzene();
// add explicit hydrogens ...
descriptor = new TPSADescriptor()
println "TPSA of oxazone: " +
    ((DoubleResult)descriptor.calculate(oxazone).getValue())
    .doubleValue()
println "TPSA of benzene: " +
    ((DoubleResult) descriptor.calculate(benzene).getValue())
    .doubleValue()
```

which returns:

```
TPSA of oxazone: 21.59
TPSA of benzene: 0.0
```

11.4. Aromaticity

I am not fond of the aromaticity concept; first of all, because there is no universal definition. Most cheminformatics toolkits have different definitions of aromaticity, and so does the CDK. If a compound is aromatic, and if so, which atoms and bonds are involved in an aromatic system are not easily

defined. Ultimately, it is the delocalization energies that has a large influence on this, which are hard to reproduce with heuristic rules in chemical graph theory-based algorithms. Nevertheless, the CDK does its best.
Atom type perception is required first (see Section 6.2) before we use the CDKHueckelAromaticityDetector:

Script 11-6: code/Aromaticity.groovy

```
mol = MoleculeFactory.makeBenzene()
AtomContainerManipulator.
  percieveAtomTypesAndConfigureAtoms(mol);
aromatic = CDKHueckelAromaticityDetector.
  detectAromaticity(mol);
println "benzene is " +
  (aromatic ? "" : "not ") + "aromatic."
```

which tells us that

```
benzene is aromatic.
```

References

[1] R. Guha, M. T. Howard, G. R. Hutchison, P. Murray-Rust, H. Rzepa, C. Steinbeck, J. Wegner, E. L. Willighagen, The Blue Obelisk - Interoperability in Chemical Informatics, *Journal of Chemical Information and Modeling* **2006**, *46*, 991–998.

[2] A. Leo, C. Hansch, D. Elkins, Partition coefficients and their uses, *Chemical Reviews* **1971**, *71*, 525–616.

[3] A. J. Leo, Calculating log Poct from structures, *Chemical Reviews* **1993**, *93*, 1281–1306.

[4] R. Wang, Y. Fu, L. Lai, A New Atom-Additive Method for Calculating Partition Coefficients, *Journal of Chemical Information and Computer Sciences* **1997**, *37*, 615–621.

[5] R. Wang, Y. Gao, L. Lai, Calculating partition coefficient by atom-additive method, *Perspectives in Drug Discovery and Design* **2000**, *19*, 47–66.

[6] P. Ertl, B. Rohde, P. Selzer, Fast Calculation of Molecular Polar Surface Area as a Sum of Fragment-Based Contributions and Its Application to the Prediction of Drug Transport Properties, *Journal of Medicinal Chemistry* **2000**, *43*, 3714–3717.

12. InChI

The IUPAC International Chemical Identifier (InChI, `http://www.iupac.org/inchi/`) is an identifier developed to provide a database-independent, unique identifier for small organic molecules [1]. The CDK uses the JNI-InChI library by Adams (`http://jni-inchi.sf.net/`) to provides a Java layer on top of the open source InChI library written in C. The InChI is design to be unique for molecules, and one InChI always identifies the same molecule, and as such is aimed to be used to look up molecules in databases or on the internet [2, 3].

To overcome the common problem caused by tautomerism in database look up, the InChI applies a number of rules to determine what the possible tautomers for a particular chemical graph are. This makes it possible to find ethanal in a database when the less-stable tautomer ethenol was searched. Both give rise to the same InChI, as we will see later.

First, we need to see how we can generate InChIs in the CDK. It starts with an InChIGeneratorFactory to create an InChIGenerator. This generator is then used to run the InChI software on the given molecule. The algorithm might fail, for various reasons, and we need to check if the generation succeeded too:

Script 12-1: code/InChIGeneration.groovy

```
factory = InChIGeneratorFactory.getInstance();
generator = factory.getInChIGenerator(methane);
if (generator.getReturnStatus() == INCHI_RET.OKAY)
  print generator.getInchi()
```

which gives the InChI for methane:

```
InChI=1S/CH4/h1H4
```

This snippet of code has generated us a StandardInChI. To explain what a Standard InChI is, we first need to briefly look at the layers in InChIs.

12.1. Layers

An InChI is like an onion. No, not in the sense that it makes you cry, but in the sense that is has layers [1]. Each layer adds more detailed information to the InChI of a molecule. The aforementioned InChI for methane has a layer reflecting the molecular formula (*/CH4*) and a hydrogen layer showing the number of hydrogens for each atom (*/h1H4*). Except for the molecular formula layer, most layers start with a lower case character, as is visible in the hydrogen layer, indicated by the (*/h*).
Another important thing to note is that hydrogens are not explicitly defined in the connection table (see Section 2.4). Therefore, the InChI for methane does not have a connectivity layer, but formic acid, *mierezuur* in Dutch, does (*/c2-1-3*):

```
InChI=1S/CH2O2/c2-1-3/h1H,(H,2,3)
```

You see that the connectivitylayer shows how the atoms are connected, and this layer it does not give bond orders. The atom numbering follows the molecular formula, where the hydrogens are not numbered. Therefore, the carbon has atom number 1, while the oxygens are atoms 2 and 3.
Now, have a careful look at this InChI for formic acid. Take a few minutes for this, and make sure you fully understand the connectivity and hydrogen layers[2].
Other layers the InChI supports include those for, for example, stereochemistry. The InChI software has a number of option to enable or disable certain layers. This explains the existence of the Standard InChI. This version of the InChI is created when a particular set of layers is used, allowing the InChI string to be used as uniqueidentifier: because it removes the choice of layers, one molecule always has the same standard InChI, whereas a molecule can have multiple InChI string depending on turning on or off certain layers. However, it is of utmost importance to realize that a particular InChI layer is always unique to the molecule, independent of layers being added or removed.
A Standard InChI string is identified by the *1S* version number. If non-standard layers are turned on, the version is simply *1*, as we will see shortly.
If you had not cheated in the mierezuur exercise, you will have noted that one hydrogen is delocalized: it can be attached to either of the oxygens. This feature is picked up by the InChI algorithm to compensate for certain kinds of tautomerism. If we want to fix the hydrogens to a particular atom, we use the following code:

[1] See *Shrek*.
[2] The answer is given in code snippet 12-2.

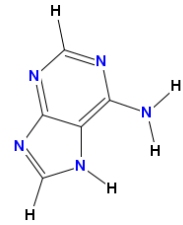

Figure 12.1.: 2D diagram of one of the tautomers of adenine.

Script 12-2: code/InChIMierezuurFixed.groovy

```
factory = InChIGeneratorFactory.getInstance();
generator = factory.getInChIGenerator(mierezuur, "FixedH");
print generator.getInchi()
```

which results in this non-standard InChI:

```
InChI=1/CH2O2/c2-1-3/h1H,(H,2,3)/f/h2H
```

By adding the FixedHoption for the InChI algorithm, we added the fixedhydrogenlayer (*/f/h2H*). This additional layer assigns one mobile hydrogen to the second atom, which is the first oxygen.

12.2. Tautomerism

Recently, Mark Rijnbeek implemented an approach published by Thalheim et al. [4] for using the InChI library for tautomer generation. While this will not calculate all tautomers for a given structure, it can still be useful as a quick indication of tautomerism. For example, the tautomers of adenine (see Figure 12.1) can be calculated with this code:

Script 12-3: code/AdenineTautomers.groovy

```
AtomContainerManipulator.percieveAtomTypesAndConfigureAtoms(
  adenine
);
tautomerGenerator = new InChITautomerGenerator();
tautomers = tautomerGenerator.getTautomers(adenine)
for (tautomer in tautomers) {
  println smilesGenerator.createSMILES(tautomer)
}
```

giving the SMILES for the eight possible tautomers returned by this method:

```
N=C1N=CNC=2NC=NC1=2
N=C1NC=NC=2NC=NC1=2
N=C1N=CNC=2N=CNC1=2
N=C2NC=NC=1N=CNC=12
N=1C=NC2=C(N=CNC=12)N
N1=CN=C2NC=NC2(=C1N)
N=1C=NC=2C=1N=CNC=2(N)
N=1C=NC(N)=C2NC=NC=12
```

It is important to note, that this functionality does not yet use the experimental /Ket and /15T from the InChI 1.03 release. Therefore, keto-enoltautomerism and 1,5-tautomerism will not be detected. In general, the heuristic rules for tautomerism detection do not catch all possible tautomers. In fact, the current code only finds 6 out of the 42 known tautomers for warfarin [5].

12.3. Parsing InChIs

Adam's CDK interface to InChI also allows parsing of InChIs into IAtomContainers. The basic workflow looks like:

Script 12-4: code/ParsingInChIs.groovy

```
InChIToStructure parser = new InChIToStructure(
  "InChI=1/CH2O2/c2-1-3/h1H,(H,2,3)/f/h2H",
  DefaultChemObjectBuilder.getInstance()
);
IAtomContainer container = parser.getAtomContainer();
```

which will create an IAtomContainer with the following SMILES:

```
O=CO
```

References

[1] S. E. Stein, S. R. Heller, D. Tchekhovskoi Proceedings of the 2003 International Chemical Information Conference, **2003**, pp. 131–143.

[2] G. Wohlgemuth, P. K. Haldiya, E. Willighagen, T. Kind, O. Fiehn, The Chemical Translation Service: web-based tool to improve standardization of metabolomic reports, *Bioinformatics (Oxford England)* **2010**, *26*, 2647–2648.

[3] S. J. Coles, N. E. Day, P. Murray-Rust, H. S. Rzepa, Y. Zhang, Enhancement of the chemical semantic web through the use of InChI identifiers, *Org. Biomol. Chem.* **2005**, *3*, 1832–1834.

[4] T. Thalheim, A. Vollmer, R.-U. Ebert, R. Kühne, G. Schüürmann, Tautomer Identification and Tautomer Structure Generation Based on the InChI Code, *Journal of Chemical Information and Modeling* **2010**, *50*, 1223–1232.

[5] W. Porter, Warfarin: history, tautomerism and activity, *Journal of Computer-Aided Molecular Design* **2010**, *24*, 553–573–573.

13. How to install the CDK

This chapter explains how the CDK can be installed on your favorite platform. The first sections discusses how the binary version can be installed, and the second section shows how the CDK can be compiled directly from the source code.

13.1. Binary Version

Like most Java software, CDK can be downloaded in binary form as *.jar* file. This binary distribution is a precompiled version of the CDK library and can be directly used as cheminformatics library, without further processing. The CDK 1.4.1 version can be downloaded from `http://sf.net/projects/cdk/files/cdk(development)/1.4.1/cdk-1.4.1.jar/download`. This Java Archive file includes all third party dependencies, and only requires a Java Virtual Machine to be used. Alternatively, you can also find a precompiled version on the complementary CD.

13.2. Source Code

There are two primary methods to download the source code for the CDK: you can download the source distribution, or you can check out the source code from the Git repository. The source distribution has the advantage that you run exactly the version of the CDK for which you downloaded the source code; using Git has the approach to run any version you like, but requires a bit more effort to get going.
The source distribution with all required third party libraries (except a Java Virtual Machine) can be downloaded as tar.gz from `http://sf.net/projects/cdk/files/cdk(development)/1.4.1/cdk-src+libs-1.4.1.tar.gz/download` or as ZIP file from `http://sf.net/projects/cdk/files/cdk(development)/1.4.1/cdk-src+libs-1.4.1.zip/download`.

13.2.1. Git Repository

The CDK source code is hosted in a Git repository on GitHub at `https://github.com/cdk/cdk/`, and mirrored at SourceForge `http://cdk.git.`

`sourceforge.net/git/gitweb.cgi?p=cdk/cdk;a=summary`. You will there find the complete history of the source code of the CDK library. Git (`http://git-scm.com/`) is a version control system that allows us to develop the CDK in a distributed manner. Everyone is invited to write patches, publish them on their web site, upload them to the CDK patch tracker, or make them available otherwise. After a formal code review process, the CDK release managers can decide to include them in the main distribution.
In contrast to downloading the source distribution, by checking out the CDK from the git repository, you get the full history of the project. All changes ever made since the start of the project are visible. While Git is pretty efficient, this still is about 100MB in raw data, and increasing almost evert day. All these changes are available as patches, and if you ever decide to submit a patch, your name will end up in the commit history of the library.
The command to make a local copy of the git repository on SourceForge looks like:

```
$ git clone git://github.com/cdk/cdk.git
```

This will get you a copy of trunk, but since we discuss CDK 1.4.1 , you will need to get the *cdk-1.4.1* branch which you can do by making a local branch:

```
$ git checkout -b local-1.4.1 origin/cdk-1.4.1
```

Compiling the library requires you have at least Ant 1.7.1 installed, which can be downloaded from `http://ant.apache.org/`.
The source code can be compile by running:

```
$ ant clean dist-large
```

13.3. Debian GNU/Linux & Ubuntu

Debian GNU/Linux and Ubuntu users can install older version of the CDK with *aptitude*:

```
$ sudo aptitude install libcdk-java
```

If you wish to compile CDK 1.4.1 , you can take advantage of the build-dependencies for the above package. With the following command you can download most of the required dependencies:

```
$ sudo apt-get build-dep libcdk-java
```

14. Writing CDK Applications

This book gave a lot of small code snippets, which can easily be integrated in larger programs. But the book has not shown so far what such a larger program can look like. This book is not about Java programming, and therefore did not introduce those aspects of using the CDK. Nevertheless, this section gives a brief introduction on how to write a Java application, a BeanShell script, and a Groovy script. Most code snippets in this book are actually Groovy scripts, as you can see on the complementary CD.

14.1. A (Very) Basic Java Application

Given you already downloaded the CDK jar file, or compiled it from scratch, consider the following piece of Java source code:

```
import org.openscience.cdk.interfaces.IAtom;
import org.openscience.cdk.Atom;

public class BasicProgram {
    public static void main(String args[]) throws Exception {
        IAtom atom = new Atom("C");
        System.out.println(atom);
    }
}
```

This Java application can then be compiled with *javac* to byte code, creating a *BasicProgram.class*:

```
$ javac -classpath cdk-1.4.1.jar BasicProgram
```

And then run with:

```
$ java -classpath .:cdk-1.4.1.jar BasicProgram
```

The downside of pure Java applications is the relative overhead needed to define an application. Other programming language provide a simpler syntax, including the BeanShell, Groovy, and Clojure described below.

14.2. BeanShell

BeanShell (`http://www.beanshell.org/`) is a simple interactive environment where one can experiment with Java libraries. For example, consider this simple script:

Script 14-1: code/BeanShell.bsh

```
import org.openscience.cdk.Atom;
Atom atom = new Atom("C");
print(atom);
```

Figure 14.1 shows the effect of running this script in the graphical frontend *xbsh*.
Beanshell needs to be made aware of the CLASSPATH, which uses the common approach for setting this:

```
$ CLASSPATH=cdk-1.4.1.jar bsh
```

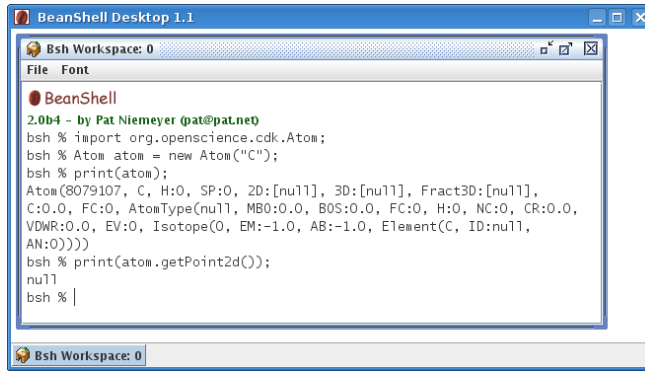

Figure 14.1.: Screenshot of *xbsh* showing a simple BeanShell script.

14.3. Groovy

Groovy (`http://groovy.codehaus.org/`) is a programming language that advertizes itself as *an agile and dynamic language for the Java Virtual Machine*. Indeed, like BeanShell, it provides an environment to quickly try Java code. However, unlike BeanShell, it provides more linguistic changes to the Java language, and adds quite interesting sugar too.
A simple script may look like:

Script 14-2: code/IterateAtoms.groovy

```
for (IAtom atom : molecule.atoms()) {
  System.out.println(atom.getSymbol());
}
```

But in Groovy it can also look like:

Script 14-3: code/IterateAtomsGroovy.groovy

```
for (atom in molecule.atoms()) {
   println atom.getSymbol()
}
```

Groovy needs to aware of the localtion of the CDK so that it can properly load the classes, for which we uses the common CLASSPATH approach. To start the GUI console shown in Figure 14.2:

```
$ CLASSPATH=cdk-1.4.1.jar groovyConsole
```

14.3.1. Closures

One of the more interesting features of Groovy is something called closures. I have know this programming pattern from R and happily used for a long time, but only recently learned them to be called closures. Closures allow you to pass a method as a parameter, which can have many applications, and I will show one situation here.
Consider the calculation of molecular properties which happen to be a mere summation over atomic properties, such as the total charge, or the molecular weight. Both these calculations require an iteration over all atoms. If we need those properties at the same time, we can combine the calcultion into one iteration. However, for the purpose of this section, we will not combine the two calculations to use one iteration, but use closures instead. Therefore, we have two slices of code which share a large amount of source code statements:

Script 14-4: code/CalculateTotalCharge.groovy

```
totalCharge = 0.0
for (atom in molecule.atoms()) {
  totalCharge += atom.getCharge()
}
```

and

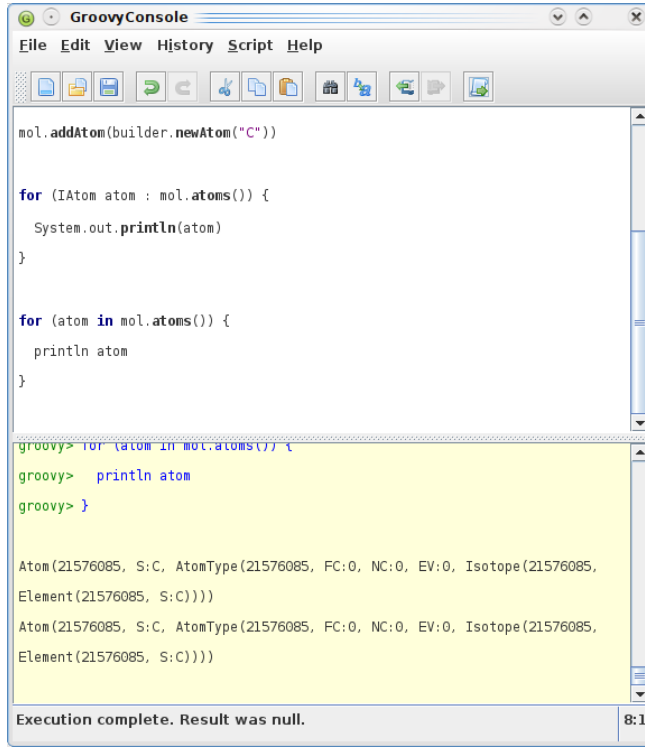

Figure 14.2.: Screenshot of *groovyConsole* showing a simple Groovy script.

Script 14-5: code/CalculateMolecularWeight.groovy

```
molWeight = 0.0
for (atom in molecule.atoms()) {
  molWeight += isotopeInfo.getNaturalMass(atom)
}
```

In both cases we want to apply a custom bit of code to all atoms, while the iteration over the atoms is identical. Groovy allows us to share the common code, by defining a forAllAtoms function into which we inject a code block using closures:

Script 14-6: code/GroovyClosureForAllAtoms.groovy

```
def forAllAtoms(molecule, block) {
  for (atom in molecule.atoms()) {
```

```
    block(atom)
  }
}
totalCharge = 0.0
forAllAtoms(molecule, { totalCharge += it.getCharge() } )
totalCharge = String.format('%.2f', totalCharge)
println "Total charge: ${totalCharge}"
molWeight = 0.0
forAllAtoms(molecule, {
  molWeight += isotopeInfo.getNaturalMass(it)
} )
molWeight = String.format('%.2f', molWeight)
println "Molecular weight: ${molWeight}"
```

which gives the output:

```
Total charge: -0.00
Molecular weight: 16.04
```

This language feature makes it possible to write more compact code.

14.4. Clojure

Clojure is a programming language which runs in a Java Virtual Machine, just like Groovy [1]. However, being Lisp-based, the syntax as well as its characteristics is quite different. A full introduction it far outside the scope of this book, but to just gave a taste of what Clojure looks like, the following code is given:

```
(import '(org.openscience.cdk Atom Bond Molecule))
(def ethanol (Molecule.))
(.addAtom ethanol (Atom. "C"))
(.addAtom ethanol (Atom. "C"))
(.addAtom ethanol (Atom. "O"))
(println methane)
```

14.5. Other Languages

There are even other languages at your disposal for using the CDK library. This book will mostly use Groovy code snippets, but this section points a few alternatives. These alternatives do not always provide access to the full CDK API, but at the same time often do offer a customized API which hides certain more technical details.

14.5.1. Bioclipse

Bioclipse has a custom scripting language with a JavaScript interface [2, 3]. Functionality is provided using *managers*, and CDK functionality is provided using two such managers. Bioclipse can be downloaded from `http://www.bioclipse.net/` and example scripts are available from the following bookmark lists: `http://delicious.com/tag/bioclipse+gist+manager:cdk` and `http://delicious.com/tag/bioclipse+gist+manager:cdx`.

14.5.2. Cinfony

Cinfony is a Python module that integrates to the CDK as well as two other cheminformatics toolkits [4]. Cinfony can be downloaded from `http://code.google.com/p/cinfony/`.

14.5.3. R

The statistical software R (`http://www.r-project.org/`) also provide access to the CDK functionality via the rcdk package [5]. This package can be downloaded from CRAN from `http://cran.r-project.org/web/packages/rcdk/`.

References

[1] S. Halloway, *Programming Clojure (Pragmatic Programmers)*, Pragmatic Bookshelf, **2009**.

[2] O. Spjuth, T. Helmus, E. L. Willighagen, S. Kuhn, M. Eklund, J. Wagener, P. Murray-Rust, C. Steinbeck, J. E. Wikberg, Bioclipse: an open source workbench for chemo- and bioinformatics, *BMC Bioinformatics* **2007**, *8*, 59+.

[3] O. Spjuth, J. Alvarsson, A. Berg, M. Eklund, S. Kuhn, C. Mäsak, G. Torrance, J. Wagener, E. L. Willighagen, C. Steinbeck, J. E. Wikberg, Bioclipse 2: A scriptable integration platform for the life sciences, *BMC Bioinformatics* **2009**, *10*, 397+.

[4] N. M. O'Boyle, G. R. Hutchison, Cinfony–combining Open Source cheminformatics toolkits behind a common interface. *Chemistry Central journal* **2008**, *2*, DOI `10.1186/1752-153X-2-24`.

[5] R. Guha, Chemical Informatics Functionality in R, *Journal of Statistical Software* **2007**, *18*, 1–16.

15. Documentation

15.1. JavaDoc

Besides this book, and in particular the keyword index at the end, you will find the Java API documentation (JavaDoc) valuable. If you have downloaded the source distributions will, you can generate the documentation with Ant in the doc/cdk-javadoc-1.4.1 folder, using:

```
$ ant -f javadoc.xml html
```

Alternatively, you can download the documentation here at `http://sf.net/projects/cdk/files/cdk(development)/1.4.1/cdk-javadoc-1.4.1.tar.gz/download`.

15.2. Other Sources

More information can be found in the following resource:

- Planet CDK: `http://pele.farmbio.uu.se/planetcdk/`
- CDK Wiki: `https://apps.sourceforge.net/mediawiki/cdk/index.php?title=Documentation`

15.2.1. Unit tests

One excellent source of information on how to use particular classes in the CDK, is to check their respective unit tests. This does require some dare, as you will have to dive into the source code repository, and you need to find the appropriate unit tests. Now, for the latter the CDK has adopted a naming scheme. First of all, the unit tests for a certain class are located in a test class in the same package. However, the functionality and their respective unit tests are located in different folders. To demonstrate that, compare these two URLs (to the current GitHub source code repository):

1. src/main/org/openscience/cdk/Atom.java, and
2. src/test/org/openscience/cdk/AtomTest.java.

We can here see that the functionality itself is found in the code/main folder, while the tests are found in the src/test folder. These two URLs also show the pattern in file naming, where the test class uses a Test suffix in the class name.
It his highly recommended to browse these classes.

16. Migration

Going from one CDK release to another brings in API changes. While the project tries to keep the number of changes minimal, these are inevitible. This chapter discusses some API changes, and shows code examples on how to change your code. Section 16.1 discusses changes from 1.2 to 1.4, Section 16.2 discusses changes from 1.0 to 1.2, and Section 16.3 discusses changes from 1.0 to 1.4,
The set of changes include changed class names. For example, the CDK 1.2 class MDLWriter is now called MDLV2000Writer to reflect the V2000 version of the MDL formats.

16.1. CDK 1.2 to 1.4

This section highlights the important API changes between the CDK 1.2 and 1.4 series.

16.1.1. Creating objects with an IChemObjectBuilder

One very prominent changes is how the IChemObjectBuilder works.
The CDK 1.2 code:

```
IChemObjectBuilder builder =
  DefaultChemObjectBuilder.getInstance();
IMolecule molecule = builder.newMolecule();
molecule.addAtom(builder.newAtom("C"));
```

looks now like:

Script 16-1: code/MigrationNewBuilder.groovy

```
IChemObjectBuilder builder =
  DefaultChemObjectBuilder.getInstance();
IMolecule molecule = builder.newInstance(
  IMolecule.class
);
molecule.addAtom(
  builder.newInstance(IAtom.class, "C")
);
```

Please note that the builder.newInstance() method may actually return null. This is not the case for the DefaultChemObjectBuilder, or the alternative NoNotifiationChemObjectBuilder builder, but future releases may have dedicated builders that do have such functionality. However, these builder would not supposed to be used for building molecules anyway.
The general patterns of newInstance() calls is that the first argument is the interface for which you want an instance. All further parameters are passed as parameters for the object's constructor. The builder maps the input to appropriate class constructors. To know what parameters you can pass when instantiating an IAtom with the DefaultChemObjectBuilder, you would look at the constructor of Atom. Therefore, we can also call:

Script 16-2: code/MigrationNewBuilder2.groovy

```
IAtom atom = builder.newInstance(
  IAtom.class, "C", new Point2d(0,0)
);
```

16.1.2. Implicit hydrogens

A second API change lies deep in the IAtom interface. To reflect more accurately the meaning of the method, the IAtomType.getHydrogenCount() has been renamed to IAtomType.getImplicitHydrogenCount(), and likewise the setter methods. The 1.2 code:

```
carbon.setHydrogenCount(4);
```

has to be updated to:

Script 16-3: code/MigrationImplicitHydrogens.groovy

```
carbon.setImplicitHydrogenCount(4);
```

Yeah, that is a simple one. Just to make clear, in both versions the count reflected the number of implicit hydrogens. The getHydrogenCount() suggested, however, to return the number of all hydrogens attached to that atom, that is, the sum of implicit and explicit hydrogens. See also Section 2.4.

16.2. CDK 1.0 to 1.2

This section highlights the important API changes between the CDK 1.0 and 1.2 series.

16.2.1. MFAnalyser

Version 1.2 removed the MFAnalyser class in favor of a more elaborate framework to handle molecular formulas. Please refer to Sections 2.3.3 and 8.4 for more detail on the new framework.

16.3. CDK 1.0 to 1.4

As discussed elsewhere, the CDK 1.2 series was released without rendering support (see Chapter 9), because the rendering engine was undergoing a complete rewrite. A good part of this has been finished, and a new rendering API is now available. The changes are too extensive to discuss here, and the reader is refered to new API discussed in Chapter 9 and start from scratch.

A. Atom Type Lists

The table listed in this Appendix is generated with the following code, listing all six properties of CDK atom types, as outlined in Section 6.1:

Script A-1: code/ListAllCDKAtomTypes.groovy

```
factory = AtomTypeFactory.getInstance(
  "org/openscience/cdk/dict/data/cdk-atom-types.owl",
  NoNotificationChemObjectBuilder.getInstance()
);
IAtomType[] types = factory.getAllAtomTypes();
for (IAtomType type : types) {
  lonepairs = type.getProperty(
    CDKConstants.LONE_PAIR_COUNT
  )
  output.append(
    type.atomTypeName + " & " +
    type.symbol + " & " +
    type.formalCharge + " & " +
    type.formalNeighbourCount + " & " +
    (type.hybridization == null
      ? ""
      : type.hybridization) + " & " +
    (lonepairs == null
      ? ""
      : lonepairs) + " & " +
    type.getProperty(
      CDKConstants.PI_BOND_COUNT
    ) + "\\\\\n")
}
```

For the Sybyl atom types we can do the same, just by updating to code to load the proper atom type list:

Script A-2: code/ListAllSybylAtomTypes.groovy

```
factory = AtomTypeFactory.getInstance(
  "org/openscience/cdk/dict/data/sybyl-atom-types.owl",
  NoNotificationChemObjectBuilder.getInstance()
);
```

A.1. CDK Atom Types

atom type	element symbol	formal charge	number of neighbors	hybridization	number of lone pairs	number of π bonds
Al.3plus	Al	3	0	S	0	0
Al	Al	0	3	SP3	0	0
Ar	Ar	0	0	SP3	4	0
As.plus	As	1	4	SP3	0	0
As	As	0	3	SP3	1	0
As.5	As	0	4	SP3	0	1
As.2	As	0	2	SP2	1	1
As.3plus	As	3	0			0
As.minus	As	-1	6			0
B.minus	B	-1	4	SP3	0	0
B	B	0	3	SP3	0	0
Be.2minus	Be	-2	4	SP3	0	0
Br	Br	0	1	SP3	3	0
Br.minus	Br	-1	0	SP3	4	0
Br.radical	Br	0	0	SP3	3	0
Br.plus.radical	Br	1	1	SP3	2	0
Br.plus.sp3	Br	1	2	SP3	2	0
Br.plus.sp2	Br	1	1	SP2	2	1
C.sp3	C	0	4	SP3	0	0
C.sp	C	0	2	SP1	0	2
C.sp2	C	0	3	SP2	0	1
C.plus.planar	C	1	3	PLANAR3	0	0
C.plus.sp2	C	1	2	SP2	0	1
C.minus.sp3	C	-1	3	SP3	1	0
C.minus.planar	C	-1	3	PLANAR3	1	0
C.radical.planar	C	0	3	PLANAR3	0	0
C.radical.sp2	C	0	2	SP2	0	1
C.radical.sp1	C	0	1	SP1	0	2
C.minus.sp2	C	-1	2	SP2	1	1
C.minus.sp1	C	-1	1	SP1	1	2
C.plus.sp1	C	1	1	SP1	0	2
Ca.2plus	Ca	2	0	S	0	0
Cl	Cl	0	1	SP3	3	0
Cl.plus.sp3	Cl	1	2	SP3	2	0
Cl.plus.sp2	Cl	1	1	SP2	2	1
Cl.minus	Cl	-1	0	SP3	4	0
Cl.radical	Cl	0	0	SP3	3	0
Cl.plus.radical	Cl	1	1	SP3	2	0
Cl.perchlorate	Cl	0	4	SP3	0	3
Cl.perchlorate.charged	Cl	3	4	SP3	0	0
Cl.chlorate	Cl	0	3	SP2	0	2
Co.2plus	Co	2	0			0
Co.3plus	Co	3	0			0

Co.metallic	Co	0	0			0
Cr	Cr	0	6		0	0
Cu.2plus	Cu	2	0			0
F	F	0	1	SP3	3	0
F.radical	F	0	0	SP3	3	0
F.minus	F	-1	0	SP3	4	0
F.plus.sp3	F	1	2	SP3	2	0
F.plus.sp2	F	1	1	SP2	2	1
F.plus.radical	F	1	1	SP3	2	0
Fe.2plus	Fe	2	0			0
Ga.3plus	Ga	3	0		0	0
Ga	Ga	0	3		0	0
Ge	Ge	0	4	SP3	0	0
Ge.3	Ge	0	3	SP2	0	1
X	H	0	null			null
H	H	0	1	S	0	0
H.plus	H	1	0	S	0	0
H.minus	H	-1	0	S	0	0
H.radical	H	0	0	S	0	0
He	He	0	0	S	1	0
Hg.minus	Hg	-1	2			0
I	I	0	1	SP3	3	0
I.minus	I	-1	0	SP3	4	0
I.minus.5	I	-1	2	SP3D1	3	0
I.plus.sp2	I	1	1	SP2	2	1
I.3	I	0	2	SP2	1	1
I.5	I	0	3	SP2	0	2
I.plus.sp3	I	1	2	SP3	2	0
I.radical	I	0	0	SP3	3	0
I.plus.radical	I	1	1	SP3	2	0
I.sp3d2.3	I	0	3	SP3D2	2	0
K.plus	K	1	0	S	0	0
K.metallic	K	0	0			0
K.neutral	K	0	1			0
Kr	Kr	0	0			0
Li	Li	0	1	S	0	0
Li.neutral	Li	0	0			0
Li.plus	Li	1	0		0	0
Mg.2plus	Mg	2	0	S	0	0
Mn.metallic	Mn	0	0			0
Mn.2plus	Mn	2	0			0
Mn.3plus	Mn	3	0			0
Mn.2	Mn	0	2			0
N.sp3	N	0	3	SP3	1	0
N.sp3.radical	N	0	2	SP3	1	0
N.sp2.radical	N	0	1	SP2	1	1
N.sp2	N	0	2	SP2	1	1
N.sp2.3	N	0	3	SP2	0	2
N.sp1.2	N	0	2	SP1	0	3
N.planar3	N	0	3	PLANAR3	1	0
N.amide	N	0	3	SP2	1	0
N.oxide	N	0	4	SP2	0	1
N.thioamide	N	0	3	SP2	1	0
N.sp1	N	0	1	SP1	1	2
N.plus	N	1	4	SP3	0	0
N.plus.sp2	N	1	3	SP2	0	1
N.plus.sp2.radical	N	1	2	SP2	0	1
N.plus.sp3.radical	N	1	3	SP3	0	0
N.plus.sp1	N	1	2	SP1	0	2
N.minus.sp3	N	-1	2	SP3	2	0
N.minus.sp2	N	-1	1	SP2	2	1
N.minus.planar3	N	-1	2	PLANAR3	2	0

N.nitro	N	0	3	PLANAR3	0	2
Na.plus	Na	1	0	S	0	0
Na	Na	0	1	S	0	0
Na.neutral	Na	0	0		0	0
Ne	Ne	0	0			0
Ni.2plus	Ni	2	0			0
Ni	Ni	0	2			0
O.sp3	O	0	2	SP3	2	0
O.sp3.radical	O	0	1	SP3	2	0
O.sp2	O	0	1	SP2	2	1
O.sp2.co2	O	0	1	SP2	2	1
O.planar3	O	0	2	PLANAR3	2	0
O.plus.sp2	O	1	2	SP2	1	1
O.plus.sp2.radical	O	1	1	SP2	1	1
O.plus.sp1	O	1	1	SP1	1	2
O.plus	O	1	3	SP3	1	0
O.plus.radical	O	1	2	SP3	1	0
O.minus	O	-1	1	SP3	3	0
O.minus.co2	O	-1	1	SP3	3	0
O.minus2	O	-2	0	SP3	4	0
P.se.3	P	0	0	SP3	0	0
P.ate	P	0	4	SP3	0	1
P.ate.charged	P	1	4	SP3	0	0
P.ine	P	0	3	SP3	1	0
P.anium	P	1	3	SP2	0	1
P.sp1.plus	P	1	2		0	2
P.irane	P	0	2	PLANAR3	1	1
P.ane	P	0	5	SP3D1	0	0
P.ide	P	0	1	SP1	1	2
Po	Po	0	2			0
Pt.2plus	Pt	2	0			0
Pt.4	Pt	0	4			0
Pt.6	Pt	0	6			0
Rn	Rn	0	0			0
S.2	S	0	1	SP2	2	1
S.3	S	0	2	SP3	2	0
S.inyl.2	S	0	2	SP2	0	2
S.oxide	S	0	2	PLANAR3	3	2
S.plus	S	1	2	SP2	1	1
S.planar3	S	0	2	PLANAR3	2	0
S.minus	S	-1	1	SP3	3	0
S.thionyl	S	0	4	SP3	0	2
S.onyl	S	0	4	SP3	0	2
S.onyl.charged	S	2	4	SP3	0	0
S.inyl	S	0	3	SP2	0	1
S.inyl.charged	S	1	3	SP2	0	0
S.trioxide	S	0	3	SP2	0	3
S.octahedral	S	0	6	SP3D2	0	0
S.anyl	S	0	4	SP3D2	1	0
Sc.3minus	Sc	-3	6		0	0
Se.3	Se	0	2	SP3	2	0
Si.2minus.6	Si	-2	6	SP3D2	0	0
Si.3	Si	0	3	SP3	0	1
Si.sp3	Si	0	4	SP3	0	0
Si.2	Si	0	2	SP1	0	2
Sn.sp3	Sn	0	4	SP3	0	0
Te.3	Te	0	2	SP3	2	0
Ti.3minus	Ti	-3	6		0	0
Ti.sp3	Ti	0	4	SP3	0	0
V.3minus	V	-3	6		0	0
W.metallic	W	0	0			0
Xe	Xe	0	0			0

Xe.3	Xe	0	4	SP3D2		0
Zn	Zn	0	2			0
Zn.2plus	Zn	2	0	S	0	0

A.2. Sybyl Atom Types

atom type	**element symbol**	**formal charge**	**number of neighbors**	**hybridization**	**number of lone pairs**	**number of π bonds**
Al	Al	0				
Br	Br	0				
C.3	C	0	4	SP3	0	0
C.2	C	0	3	SP2	0	1
C.ar	C	0	3	SP2	0	1
C.1	C	0	2	SP1	0	2
C.cat	C	1				
Du.C	C	0				
Ca	Ca	0				
Cl	Cl	0				
Co.oh	Co	0				
Cr.th	Cr	0				
Cr.oh	Cr	0				
Cu	Cu	0				
F	F	0				
Fe	Fe	0				
H	H	0	1		0	0
H.spc	H	0				
H.t3p	H	0				
LP	H	0				
Du	H	0				
Any	H	0				
Hal	H	0				
Het	H	0				
Hev	H	0				
X	H	0				
I	I	0				
K	K	0				
Li	Li	0				
Mg	Mg	0				
Mn	Mn	0				
Mo	Mo	0				
N.3	N	0	3	SP3	1	0
N.2	N	0	2	SP2	1	1
N.1	N	0	1	SP1	1	2
N.pl3	N	0	3		1	0
N.4	N	1	4	SP3	0	0
N.ar	N	0	2	SP2	1	1
N.am	N	0	3	SP2	1	0
Na	Na	0				
O.3	O	0	2	SP3	2	0
O.2	O	0	1	SP2	2	1
O.co2	O	0	1			
O.spc	O	0				
O.t3p	O	0				
P.3	P	0	4	SP3	0	1
S.3	S	0	2	SP3	2	0
S.2	S	0	1	SP2	2	1
S.O	S	0				
S.O2	S	0				

Se	Se	0
Si	Si	0
Sn	Sn	0
Zn	Zn	0

B. Isotope List

The table listed in this Appendix is generated with the following code, listing all six properties of CDK atom types, as outlined in Section 6.1. Abundances, exact masses are inherited from the BODR project [1], which contains values found in IUPAC recommendations.

Script B-1: code/ListAllIsotopes.groovy

```
isofac = IsotopeFactory.getInstance(
  NoNotificationChemObjectBuilder.getInstance()
);
maxAtomicNumber = 150;
for (atomicNumber in 1..maxAtomicNumber) {
  element = isofac.getElement(atomicNumber)
  isotopes = isofac.getIsotopes(element.symbol)
  for (isotope in isotopes) {
    if (isotope.naturalAbundance > 0.1) {
      output.append(
        atomicNumber + " & " +
        element.symbol + " & "
      )
      output.append(
        isotope.massNumber + " & " +
        isotope.naturalAbundance + " & " +
        isotope.exactMass + "\\\\\n"
      )
    }
  }
}
```

The full version of the above script lists all (natural) isotopes with an abundance of more than 0.1:

atomic number	element symbol	mass number	abundance	exact mass
1	H	1	99.9885	1.007825032
2	He	4	99.999863	4.002603254
3	Li	6	7.59	6.015122795

		7	92.41	7.01600455
4	Be	9	100.0	9.0121822
5	B	10	19.9	10.012937
		11	80.1	11.0093054
6	C	12	98.93	12.0
		13	1.07	13.00335484
7	N	14	99.632	14.003074
		15	0.368	15.0001089
8	O	16	99.757	15.99491462
		18	0.205	17.999161
9	F	19	100.0	18.99840322
10	Ne	20	90.48	19.99244018
		21	0.27	20.99384668
		22	9.25	21.99138511
11	Na	23	100.0	22.98976928
12	Mg	24	78.99	23.9850417
		25	10.0	24.98583692
		26	11.01	25.98259293
13	Al	27	100.0	26.98153863
14	Si	28	92.2297	27.97692653
		29	4.6832	28.9764947
		30	3.0872	29.97377017
15	P	31	100.0	30.97376163
16	S	32	94.93	31.972071
		33	0.76	32.97145876
		34	4.29	33.9678669
17	Cl	35	75.78	34.96885268
		37	24.22	36.96590259
18	Ar	36	0.3365	35.96754511
		40	99.6003	39.96238312
19	K	39	93.2581	38.96370668
		41	6.7302	40.96182576
20	Ca	40	96.941	39.96259098
		42	0.647	41.95861801
		43	0.135	42.9587666
		44	2.086	43.9554818
		48	0.187	47.952534
21	Sc	45	100.0	44.9559119
22	Ti	46	8.25	45.9526316
		47	7.44	46.9517631
		48	73.72	47.9479463
		49	5.41	48.94787
		50	5.18	49.9447912
23	V	50	0.25	49.9471585
		51	99.75	50.9439595
24	Cr	50	4.345	49.9460442
		52	83.789	51.9405075
		53	9.501	52.9406494
		54	2.365	53.9388804
25	Mn	55	100.0	54.9380451
26	Fe	54	5.845	53.9396105
		56	91.754	55.9349375
		57	2.119	56.935394
		58	0.282	57.9332756
27	Co	59	100.0	58.933195
28	Ni	58	68.0769	57.9353429
		60	26.2231	59.9307864
		61	1.1399	60.931056
		62	3.6345	61.9283451
		64	0.9256	63.927966
29	Cu	63	69.17	62.9295975
		65	30.83	64.9277895

30	Zn	64	48.63	63.9291422
		66	27.9	65.9260334
		67	4.1	66.9271273
		68	18.75	67.9248442
		70	0.62	69.9253193
31	Ga	69	60.108	68.9255736
		71	39.892	70.9247013
32	Ge	70	20.84	69.9242474
		72	27.54	71.9220758
		73	7.73	72.9234589
		74	36.28	73.9211778
		76	7.61	75.9214026
33	As	75	100.0	74.9215965
34	Se	74	0.89	73.9224764
		76	9.37	75.9192136
		77	7.63	76.919914
		78	23.77	77.9173091
		80	49.61	79.9165213
		82	8.73	81.9166994
35	Br	79	50.69	78.9183371
		81	49.31	80.9162906
36	Kr	78	0.35	77.9203648
		80	2.28	79.916379
		82	11.58	81.9134836
		83	11.49	82.914136
		84	57.0	83.911507
		86	17.3	85.91061073
37	Rb	85	72.17	84.91178974
		87	27.83	86.90918053
38	Sr	84	0.56	83.913425
		86	9.86	85.9092602
		87	7.0	86.9088771
		88	82.58	87.9056121
39	Y	89	100.0	88.9058483
40	Zr	90	51.45	89.9047044
		91	11.22	90.9056458
		92	17.15	91.9050408
		94	17.38	93.9063152
		96	2.8	95.9082734
41	Nb	93	100.0	92.9063781
42	Mo	92	14.84	91.906811
		94	9.25	93.9050883
		95	15.92	94.9058421
		96	16.68	95.9046795
		97	9.55	96.9060215
		98	24.13	97.9054082
		100	9.63	99.907477
44	Ru	96	5.54	95.907598
		98	1.87	97.905287
		99	12.76	98.9059393
		100	12.6	99.9042195
		101	17.06	100.9055821
		102	31.55	101.9043493
		104	18.62	103.905433
45	Rh	103	100.0	102.905504
46	Pd	102	1.02	101.905609
		104	11.14	103.904036
		105	22.33	104.905085
		106	27.33	105.903486
		108	26.46	107.903892
		110	11.72	109.905153
47	Ag	107	51.839	106.905097

		109	48.161	108.904752
48	Cd	106	1.25	105.906459
		108	0.89	107.904184
		110	12.49	109.9030021
		111	12.8	110.9041781
		112	24.13	111.9027578
		113	12.22	112.9044017
		114	28.73	113.9033585
		116	7.49	115.904756
49	In	113	4.29	112.904058
		115	95.71	114.903878
50	Sn	112	0.97	111.904818
		114	0.66	113.902779
		115	0.34	114.903342
		116	14.54	115.901741
		117	7.68	116.902952
		118	24.22	117.901603
		119	8.59	118.903308
		120	32.58	119.9021947
		122	4.63	121.903439
		124	5.79	123.9052739
51	Sb	121	57.21	120.9038157
		123	42.79	122.904214
52	Te	122	2.55	121.9030439
		123	0.89	122.90427
		124	4.74	123.9028179
		125	7.07	124.9044307
		126	18.84	125.9033117
		128	31.74	127.9044631
		130	34.08	129.9062244
53	I	127	100.0	126.904473
54	Xe	128	1.92	127.9035313
		129	26.44	128.9047794
		130	4.08	129.903508
		131	21.18	130.9050824
		132	26.89	131.9041535
		134	10.44	133.9053945
		136	8.87	135.907219
55	Cs	133	100.0	132.9054519
56	Ba	130	0.106	129.9063208
		132	0.101	131.9050613
		134	2.417	133.9045084
		135	6.592	134.9056886
		136	7.854	135.9045759
		137	11.232	136.9058274
		138	71.698	137.9052472
57	La	139	99.91	138.9063533
58	Ce	136	0.185	135.907172
		138	0.251	137.905991
		140	88.45	139.9054387
		142	11.114	141.909244
59	Pr	141	100.0	140.9076528
60	Nd	142	27.2	141.9077233
		143	12.2	142.9098143
		144	23.8	143.9100873
		145	8.3	144.9125736
		146	17.2	145.9131169
		148	5.7	147.916893
		150	5.6	149.920891
62	Sm	144	3.07	143.911999
		147	14.99	146.9148979
		148	11.24	147.9148227

		149	13.82	148.9171847
		150	7.38	149.9172755
		152	26.75	151.9197324
		154	22.75	153.9222093
63	Eu	151	47.81	150.9198502
		153	52.19	152.9212303
64	Gd	152	0.2	151.919791
		154	2.18	153.9208656
		155	14.8	154.922622
		156	20.47	155.9221227
		157	15.65	156.9239601
		158	24.84	157.9241039
		160	21.86	159.9270541
65	Tb	159	100.0	158.9253468
66	Dy	160	2.34	159.9251975
		161	18.91	160.9269334
		162	25.51	161.9267984
		163	24.9	162.9287312
		164	28.18	163.9291748
67	Ho	165	100.0	164.9303221
68	Er	162	0.14	161.928778
		164	1.61	163.9292
		166	33.61	165.9302931
		167	22.93	166.9320482
		168	26.78	167.9323702
		170	14.93	169.9354643
69	Tm	169	100.0	168.9342133
70	Yb	168	0.13	167.933897
		170	3.04	169.9347618
		171	14.28	170.9363258
		172	21.83	171.9363815
		173	16.13	172.9382108
		174	31.83	173.9388621
		176	12.76	175.9425717
71	Lu	175	97.41	174.9407718
		176	2.59	175.9426863
72	Hf	174	0.16	173.940046
		176	5.26	175.9414086
		177	18.6	176.9432207
		178	27.28	177.9436988
		179	13.62	178.9458161
		180	35.08	179.94655
73	Ta	181	99.988	180.9479958
74	W	180	0.12	179.946704
		182	26.5	181.9482042
		183	14.31	182.950223
		184	30.64	183.9509312
		186	28.43	185.9543641
75	Re	185	37.4	184.952955
		187	62.6	186.9557531
76	Os	186	1.59	185.9538382
		187	1.96	186.9557505
		188	13.24	187.9558382
		189	16.15	188.9581475
		190	26.26	189.958447
		192	40.78	191.9614807
77	Ir	191	37.3	190.960594
		193	62.7	192.9629264
78	Pt	192	0.782	191.961038
		194	32.967	193.9626803
		195	33.832	194.9647911
		196	25.242	195.9649515

		198	7.163	197.967893
79	Au	197	100.0	196.9665687
80	Hg	196	0.15	195.965833
		198	9.97	197.966769
		199	16.87	198.9682799
		200	23.1	199.968326
		201	13.18	200.9703023
		202	29.86	201.970643
		204	6.87	203.9734939
81	Tl	203	29.524	202.9723442
		205	70.476	204.9744275
82	Pb	204	1.4	203.9730436
		206	24.1	205.9744653
		207	22.1	206.9758969
		208	52.4	207.9766521
83	Bi	209	100.0	208.9803987
90	Th	232	100.0	232.0380553
91	Pa	231	100.0	231.035884
92	U	235	0.72	235.0439299
		238	99.2745	238.0507882

References

[1] R. Guha, M. T. Howard, G. R. Hutchison, P. Murray-Rust, H. Rzepa, C. Steinbeck, J. Wegner, E. L. Willighagen, The Blue Obelisk - Interoperability in Chemical Informatics, *Journal of Chemical Information and Modeling* **2006**, *46*, 991–998.

C. CDK Authors

In acknowledgment to all the work done by the many people who have contributed to the success of the CDK, here is a list of those who wrote smaller or larger parts of the CDK library:

Sam Adams, Jonathan Alvarsson, Rich Apodaca, Saravanaraj N Ayyampalayam, Ulrich Bauer, Stephan Beisken, Arvid Berg, Ed Cannon, Fabian Dortu, Martin Eklund, Matteo Floris, Dan Gezelter, Uli Fechner, Rajarshi Guha, Yonquan Han, Thierry Hanser, Kai Hartmann, Tobias Helmus, Christian Hoppe, Oliver Horlacher, Miguel Howard, Nina Jeliazkova, Geert Josten, Dmitry Katsubo, Jules Kerssemakers, Anatoli Krassavine, Stefan Kuhn, Uli Köhler, Violeta Labarta, Jonty Lawson, Daniel Leidert, Edgar Luttmann, Todd Martin, Nathanaël Mazuir, Stephan Michels, Scooter Morris, Peter Murray-Rust, Carl Mäsak, Irilenia Nobeli, Peter Odéus, Niels Out, Jerome Pansanel, Julio Peironcely, Chris Pudney, Syed Asad Rahman, Jonathan Rienstra-Kiracofe, Mark Rijnbeek, David Robinson, Miguel Rojas Cherto, Bhupinder Sandhu, Jean-Sebastien Senecal, Onkar Shinde, Sulev Sild, Bradley Smith, Ola Spjuth, Christoph Steinbeck, Aleksey Tarkhov, Stephan Tomkinson, Gilleain Torrance, Andreas Truszkowski, Paul Turner, Jörg Wegner, Stephane Werner, Egon Willighagen, Yong Zhang, and Daniel Zaharevitz.

List of Scripts

Index