Java and JavaBeans for Cheminformatics

Anatoli Krassavine*

Abstract. The usability of standard HTML pages for the display of inherently often very graphical chemical information is severely limited, especially with respect to interactive manipulation of the displayed data. This is where simple images, the lowest common denominator, fail. There is a definite need for a portable and flexible mechanism for the graphical display and manipulation of chemical information in a WWW context. We believe that Java, especially the JavaBeans framework, which supplies a powerful paradigm for the communication between multiple processing tools, is a very suitable technology to solve these problems. In this contribution, we illustrate the design and use of the ChemSymphony JavaBean suite as an example of this technology.

When I first conceived ChemSymphony, my intention was to develop a system for publishing chemistry models and molecular representations on the World-Wide Web. In its first iteration, ChemSymphony was seen as having the potential to be an essential tool for building web sites with chemistry content. This was the basis on which Cherwell Scientific agreed to publish the suite in 1996. The general challenge of developing a modular system which would represent chemical content using a variety of legacy file formats encouraged the adoption of a flexible and "open" data model which is well fitted to many (not all) informatics tasks in chemistry. Java was essential to this vision of a generic and cross-platform chemistry-publishing suite. In the last two years, Java has evolved in ways which encourage a broader ambition for ChemSymphony [1].

A most important improvement to the Java standard was the introduction (with JDK 1.1) of a component technology: JavaBeans. These developments have encouraged us to develop a suite of basic chemistry components which can be used to build solutions for the Internet, for corporate Intranets and eventually perhaps for incorporation in a wide variety of devices which are able to support Java Virtual Machines.

The Basics of ChemSymphony Beans

The one-sentence definition of JavaBeans is: Java software packaged as components which can be plugged together to build programs (applications or applets). The one-sentence definition of ChemSymphony Beans is: a suite of JavaBeans which offers a tool kit for chemistry Intranets and provides most of the most commonly used components that are needed for building chemistry programs. Since this is a construction kit, it is to be expected that ChemSymphony components will be incorporated in quite specialised programs and used for projects which are much more sophisticated than the original designers will have considered possible.

The core resource of the suite is the common data model which all the ChemSymphony Beans share. This is probably the most unusual feature of ChemSymphony Beans. We have not noticed in any neighbouring field, a component collection of Beans which has been built sharing a common data model and with a common architecture. But the underlying data model is not what people notice when they are first introduced to these Beans. What most people notice is "what the Beans look like". So we will start with this. In fact, there are several ways of looking at ChemSymphony Beans, and the initial view is usually of the GUI components. This is what two Beans look like (Fig. 1).

On the left is a component which lets you render a molecule in three dimensions. Its functionality is similar to Rasmol or Chime, but it is a JavaBean, so it can be connected directly to any other JavaBean or indirectly to any other information source or computational program. On the right is another GUI component which allows the user to alter properties which may have some effect on the way the molecule is represented or rendered. But note that the components are connected by an arrow and a horizontal message, a 'method' which engages the Property-Change event.

The essential strength of the JavaBeans technology is that there is a standard 'event model' with its own logic which allows the behaviour of individual components to be interconnected. ChemSymphony relies heavily on this infrastructure. We do not need to know too much about the internals of 'foreign' Beans, provided that they have been written to the Java standards defined by Sun Microsystems. In thinking about what Beans 'look like', it is more important to consider the abstract ways in which they can be connected together. Here is a diagramatic view of several connected Beans (Fig. 2).

This was an early design for an interface to the Beilstein database. It uses six ChemSymphony Beans and four generic Beans from Sun. The layout of the design gives an idea of the 'logic' which the application embodies and also how easily different applications can be adapted one from another.

It also illustrates how simple it is to adapt the modular architecture of ChemSymphony to completely different tasks. Let's consider the diagram above once again. As drawn - it will connect to Beilstein database, submit query, extract and display hits and associated data. What if you wish to adapt the very same Beilstein interface to work with another system (an Oracle Database or a Daylight system)? Well roughly speaking, you would like to keep the same general application skeleton. You could use the same sketcher for submitting query, the same display to display resultng structures, the same buttons to control data flow.... The only difference is you want to connect to another database - so you have to substitute the Beilstein adapter with an adapter specific to the database of your choice. It is made easier due to the fact that aside from a few specific controls, all the database adapters share the same basic public interface, an example of which is shown below (In the diagram, we can compare lists of inputs and outputs between Beilstein and NCI adapters; Fig. 3).
Well, that’s it. The new design is fully and immediately functional.

A person intimately aware of the nuances of working with different databases might reasonably point out that different database use different protocols to obtain data, require queries in different formats, as well. So isn’t it true that we need to tweak all the connection and options accordingly?

Not at all. Thanks to the fact that all components share the common interface, such problems could be easily resolved in run time and, most importantly, could be resolved automatically between the components themselves. For example, albeit you might input substructures in the same sketcher, Beilstein would require the structural query data to be converted into BSD query, while Daylight or NCI would re-
quire SMILES. The Sketcher, most defi-
tinitely, is not aware of such nuances – but
fortunately, the database adapter does
know –, so it instructs the sketcher to pass
the data in the appropriate specific format
at run time.

Analogously, the display component
might not know what type of data any
particular database returns, but fortunate-
ly, the database adapter does know and
informs the display about the available
options.

All this is resolved at run time during
inter-component interactions and techni-
cally does not require user interaction. But
for the expert users, there is a backdoor –
a second line of programmable interfaces,
which allows him to specify the exact
formats he wishes for data transfer or to
tweak any data exchanges he needs to
override.

Essentially the idea is that components
could and should solve the details of par-
ticularities of data transfer and data for-
matting in run time, leaving the user to
solve more intelligent tasks.

Some of the ChemSymphony Beans
are GUI components which gives us a nice
impression of what the suite can do. Argu-
ably, a deeper understanding of the tool kit
comes from considering the designs or the
blueprints for different applications. But,
as we have several times hinted, the core
of the suite is something that one really
does not see: the data model.

All the ChemSymphony Beans share a
data model and common data exchange
interfaces. This is an important point and
it bears on another issue that newcomers
can find confusing in JavaBeans. We have
talked about JavaBeans as components, as
free-standing code units. They are this, but
we need to be careful in the way we think
of the Beans as completely individual com-
ponents. JavaBeans need the Java envi-
noment. They require the Java Virtual
Machine and the 'Event Model' which
comes with JDK 1.1. In a somewhat anal-
ogous fashion, the ChemSymphony Beans
need the underlying event model to effec-
tively communicate with each other and
underlying chemical data model to effi-
ciently manipulate the data.

The ChemSymphony data model has
evolved, and its implementation will con-
tinue to be improved. An important early
decision was to build the data model from
general abstract considerations of molec-
ular structure, rather than relying primari-
ly on any particular data file format.

For communications with the outer
world, ChemSymphony uses a number of
"input/output filters" and 'data source
adapters'.

All filters and adapters, of which we
will talk in more detail below, are dynam-
ically linked objects. This means that a
developer can write his own filter/adapter
to cover his particular data needs, and he
will be able to seamlessly incorporate it
into the system. All the interfaces/protocol
needed for writing such a code are mademade public as well as some simple anno-
tated code examples. ChemSymphony
provides several levels of interfaces, de-
pending on how much time/resources one
wants to spend on programming. Again,
there is a trade-off between ease of use and
flexibility for adaptation. Deeper levels
offer more complex interfaces which offer
a developer more control over both input
and output at the price of requesting more
complex code from the developer.

Once compiled, the particular details
of deployment will differ depending on
the IDE/environment one is using, but in
some environments, it is as simple as drop-
ping the compiled code into a predefined
directory. ChemSymphony will automatic-
ally detect the presence of new classes,
identify the relevant properties and incor-
porate it into the system.

It is useful at this stage to say a little bit
more about the particular differences be-
 tween filters and adapters. The task of a
filter is to take a piece of data in the form
of a flat file, an object or another stream-
lined piece of data, and map it onto the
internal data model. Equally, the filters
should be able to reverse the process: to
compile a presentation of internal data
into the standard output data format. Gen-
erally, filters operate with a stream of
incoming data. As such, they operate in a
somewhat ideal world, where somebody
else takes care of where the data really
reside. Filters have no idea where this
stream has come from or whether any
special activities are needed to keep data
coming. This is not their job, but it is the
job of adapters.

Data source adapters are operating with
the outer world. Their task is to connect to
a particular data source, request data, iden-
tify the returned data type, open and main-
tain the connection until all the necessary
data are dumped into an appropriate data
filter, etc. 'Data source' is our general
term in ChemSymphony, by which we
understand any external source of chemi-
cal data, including – but not limited to –
disk drives, network servers, database en-
gines, clipboards, computational packag-
es and such.

Instead of requiring the outer world to
comply with a specific protocol or inter-
faced in ChemSymphony is surrounded by
shields of autodetecting, customised adap-
ters (we use the word 'adapter' analogously
to the electrical adapters which con-
nect your laptop/hair dryer to the electric-
ity source in different countries). Each
adapter has two sides. On the inside, it
supports the common ChemSymphony's
interfaces. On the outside, it supports
whatever protocol this particular data
source requires.

Different adapters are needed for han-
dling each specific data source, but since
they are in effect black boxes, the develop-
er can easily swap them or swap the appli-
cations to which they are connected (the
adapter which connected a Beilstein 'data
source' to a visualisation application could
be connected directly to a calculation bean
if it is clear that the chemist who is using
Beilstein wants to use the results of search-

Fig. 3. Lists of inputs and outputs between Beilstein and NCI adapters
es in calculations). We claim that it is a
strength of the approach that through its
modular adapters and filters ChemSym-
phony can 'learn' more about chemistry
and molecular formats. This can be done
by amending existing filters or by adding
new filters.

Once the particular data source has
been connected and the raw data were
extracted by one adapter or another, and
after the raw data has been routed through
the appropriate filter to get them into the
system, the central data model itself comes
into play.

ChemSymphony's internal data mod-
el, what we call its **Generic Chemical
Model**, is the engine at the core of any
ChemSymphony applet or application built
with the Beans. The idea was to create a
system for describing chemical structures,
which will be shared by all the compo-
ments inside the tool kit and any applica-
tions using the tool kit. Once this is done,
we have an efficient and flexible way of
transferring data inside the system. The
system for description is sufficiently ab-
tract to allow us to incorporate additional
properties and pieces of data, as the model
is to be applied to a variety of tasks. The
**Generic Chemical Model** in particular
should provide means for storing/retriev-
ing all the information found in common
data file formats, *i.e.*, if a PDB file is to be
downloaded into the system, the system
should be capable at any time to restore all
the information found in the original file.

The **Generic Chemical Model** was
designed with the well-behaved organic
structures in mind. It uses an abstract rep-
resentation of a list of vertices to hold
information on atoms and groups of at-
oms. A list of bonds is used to link the
atoms (groups of atoms).

Special consideration was given to
implementing multiple ways of associat-
ing elements of the structure into logical
groups and dealing with those groups.
'Chains', 'atoms within an individual res-
idue', 'secondary structure assignments'
are just some of the examples of possible
default groupings.

Properties can be associated with indi-
vidual bonds or with vertices, groups of
bonds, or with the structure as a whole. In
addition to the default list of available
groupings, the developer can program his
own groupings specific to his particular
needs, which will operate alongside the
default ones.

For example, the code below shows
one of the ways of selecting specific groups
inside the structure - highlight all nitrogen
and sulfur atoms inside the structure:

```java
GenericChemModel md = new GenericChemModel();
Vector ns = new Vector();
// this will locate all the nitrogens inside the chemical model
// and append them to the vector
md.pickUpProperty(ns, 'name', 'n')
// this is one of the methods which simplifies matters for
// the user
md.select(ns, 2);
```

The division between abstract and cus-
tom methods causes enough confusion, so
some more explanations are needed. All
the properties could be roughly divided
into two classes.

All the properties support generic data
manipulation interfaces, which allow code
to get/modify/remove any particular piece
of data in abstract way, non dependent on
particular contents. Likewise all the lists
of properties are not fixed, but dynamical-
ly defined, which allows new properties
for each object inside the data model to be
defined in run time and still be subjected to
the much the same manipulation as the
default properties.

But some properties have a set of cus-
tom methods for special manipulations in
addition to the abstract ones. Such meth-
ods are usually created for common prop-
erties - which are likely to be present in
many objects and often referred to. For
example, it could be the 3D coordinates of
vertexes. The developer can still use the generic abstract methods for handling them, but they are also provided with a set of custom methods/interfaces for more efficient and rapid manipulations.

For example, since 'simple charge' is considered to be a common feature, it is provided for both in a generic implementation (being treated as one of the default named properties)

```java
v.getProperty("charge");
and in specific implementations
v.getCharge();
```

The basic data model is being continuously enhanced and improved, and these improvements are modular and global (Fig. 4). If you write a new filter for a ChemSymphony project (e.g., to cope with a legacy data source), this resource will always be available to you.

a) What the molecule looks like
b) Molecular graph
c) Internal representation by lists of vertices and bonds

Below is an example of source code using the representation above to make something useful:

```java
// create generic datamodel - initially empty
GenericChemModel md = new GenericChemModel();

// create new vertex
D3Vertex v1 = new D3Vertex("C", 2.765, 8.4, 0.0);
//add this vertex to datamodel
md.addVert(v1);

// set one of the default properties
v1.setProperty("charge", "+1");
// since this property is default - it has generic implementation as well, i.e.
// v1.setCharge(1) is also correct, albeit the first one is more general

// now some user property
v1.setProperty("my atom index", "1");

// create second vertex
D3Vertex v2 = new D3Vertex("C", 1.0, 9.0, 0.0);
// add this vertex to the datamodel
md.addVert(v2);

// create new bond connecting the two vertexes created above
D3Bond bond = md.connect(v1, v2);
// or bond = md.findBond(v1, v2); // or bond = v1.getBond(v2);
// all of those use different points of reference, but result in the same behaviour
```

As I have already mentioned the ChemSymphony data model was designed and implemented with 'well-behaved' organic molecules in mind. The built-in flexibility allowed us to adapt it to the variety of completely different tasks (many of which could not even be considered at the time of the initial design), such as database searches, chemical reactions, quantum chemistry calculations and others. It nevertheless has its inherent limitations when we leave the scope of single molecules (limitations which are also present in most of the widely used formats for describing molecules).

The ChemSymphony data model is not very effective in operating on inter-related ensembles of molecules with complicated dependencies. It also cannot be applied effectively to crystallographic structures and data sets. As of now, we have limited support for advanced chirality manipulations, although this support is easy to implement within the existing data model. It now seems to us that the most efficient way of generalising the ChemSymphony data model so as to cope with these complications, complexities which are fundamental and inherent in real chemistry, will lead us to address some of the issues pertinent to handling data in other scientific domains.

The ChemSymphony data model was designed for chemistry, but it could be adapted for certain other fields (in the early days of ChemSymphony, some computer scientists experimented with using the schema to represent hypertext networks on the WWW). We think that it should be possible to generalise the data model to cope with other information domains [2].

Received: October 13, 1998

[1] ChemSymphony's WWW site has full documentation and a number of examples (http://www.chemsymphony.com/).

[2] Early thoughts on an extension to the data model were presented to the Objects in the Bioinformatics conference in August, 1998 (http://www.ebi.ac.uk/oib98/Abstracts/loeffler.html).