Systems Biology Workbench Java™ Programmer's Manual

Michael Hucka, Andrew Finney,Herbert Sauro,
Hamid Bolouri, Frank Bergmann

{frank_bergman,hsauro}@kgi.edu, mhucka@caltech.edu
Systems Biology Workbench Development Group
ERATO Kitano Systems Biology Project
Control and Dynamical Systems, MC 107-81
California Institute of Technology, Pasadena, CA 91125, USA

Keck Graduate Institute
535 Watson Drive, Claremont
CA 91711, USA

http://www.sys-bio.org

Principal Investigators: John Doyle and Hiroaki Kitano

October 21, 2004

Contents

1 Introduction

The aims of this manual are twofold: teaching programmers how to interface Java applications to the Systems Biology Workbench (SBW), and providing a reference for the SBW Java APIs. This manual complements the overviews of the SBW system provided by Hucka et al. (2001a,c, 2002) and the description by Sauro et al. (2001) of a prototype SBW implementation.

SBW provides a software integration environment that enables applications (potentially running on separate machines) to learn about and communicate with each other. Applications can be written to be providers of software services, or consumers, or both. The SBW communications facilities allow heterogeneous packages to be connected together using a remote procedure call mechanism; this mechanism uses a simple message-passing network protocol and allows either synchronous or asynchronous invocations. The interfaces to SBW are encapsulated in client libraries for different programming languages (currently C, C++, Delphi, Java, MATLAB and Python, with more anticipated), but the protocol is open and small, and developers may implement their own interfaces to the system if they choose. The project is entirely open-source and all specifications and implementations are freely and publicly available.

Frameworks for integrating disparate software packages are certainly not new. Compared to other broker-based integration frameworks, SBW has the following combination of features:

- Free, open-source implementations available for all platforms;
- Availability today for Linux and Windows, with more platforms anticipated in the future;
- Comparatively simple APIs and data exchange protocol;
- Support for major programming and scripting languages and the seamless interaction between modules written in different languages;
- No need for a separately-compiled interface definition language (IDL); and
- Resource management performed by underlying services (which means, for example, that there is no exposed object reference counting).

SBW is being developed as part of existing collaborations in the systems biology community with the following software development groups: *BioSpice* (Arkin, 2001), *DBsolve* (Goryanin, 2001; Goryanin et al., 1999), *E-CELL* (Tomita et al., 1999, 2001), *Gepasi* (Mendes, 1997, 2001), *Jarnac* (Sauro and Fell, 1991; Sauro, 2000), *ProMoT/DIVA* (Ginkel et al., 2000), *StochSim* (Bray et al., 2001; Morton-Firth and Bray, 1998), and *Virtual Cell* (Schaff et al., 2000, 2001). These collaborations have already successfully established SBML, the Systems Biology Markup Language (SBML; Hucka et al., 2001b), as an important emerging standard in systems biology.

2 A Brief Tour of SBW

The primary goal of SBW is to allow the *integration* of software components performing a wide range of tasks common in computational biology, such as simulation, data visualization, optimization, and bifurcation analysis. SBW is not designed to be in the foreground of either the user's or the programmer's experience, but instead, to allow existing systems biology software packages to easily and transparently access functionality from each other.

In more specific terms, SBW is a computational resource brokerage system. It allows the interactive discovery and use of software resources. In the SBW scheme of things, software resources are independent applications and are called *modules.* A module instance is a running application or process. A module can implement one or more *services. Services* are interfaces to the resources inside a module and consist of one or more *methods* (i.e., callable functions).

Broker architectures are relatively common and are considered to be a well-documented software pattern (Buschmann et al., 1996). In SBW, the remote service invocations are implemented using *message passing*, another well-known and proven software technology. Communications in message-passing systems take place as exchanges of structured data bundles—messages—sent from one software entity to another over a channel. Some messages may be requests to perform an action, other messages may be notifications or status reports. Because interactions in a message-passing framework are defined at the level of messages and protocols for their exchange, it is easier to make the framework neutral with respect to implementation languages: modules can be written in any language, as long as they can send, receive and process appropriately-structured messages using agreed-upon conventions. Figure 1 illustrates the overall SBW system organization.

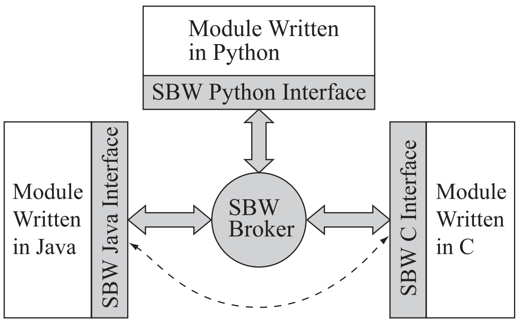

Figure 1: *The overall organization of the Systems Biology Workbench. Gray areas indicate SBW components (libraries and the broker). To individual modules, communications appear to be direct (dotted line), although they actually pass through the broker.*

From the application programmer's point of view, it is preferable to isolate the message-passing details from the application details. We provide two levels of programming interfaces in SBW: a low-level API consisting of the basic operations involved in sending and receiving messages, and a high-level API that hides the messaging level and provides ways for a module's services to be "hooked into" the messaging framework in an object-oriented fashion. Throughout this manual, we concentrate on the more convenient high-level API, but both the low- and high-level APIs are always available in the same SBW Java library. Section 5 describes the low-level API.

2.1 Overview of SBW from a Programmer's Perspective

The SBW APIs provide the following facilities:

1. *Dynamic service and module discovery*: The SBW Broker keeps track of modules, services and service categories, and provides facilities for a module to learn about them.

2. *Remote method invocation*: The bread and butter of SBW is enabling one module to invoke a service method in another module. If necessary, the SBW Broker will automatically start an instance of a module whose services are requested.

3. *Data serialization*: Method invocations involve sending messages between modules, with arguments and data packed into message streams. For some languages such as Java, Perl and Python, the SBW library provides proxy objects that hide the message-passing, so that to client programs, remote services appear as local objects whose methods can be invoked like any other object method.

4. *Exception handling*: SBW provides facilities for dealing with exceptional conditions.

5. *Event notification*: Certain events in SBW, such as the startup or shutdown of an instance of a module, are announced to all modules upon their occurrence.

6. *Module, service and method registration*: In order for a module to advertise its services to others, it must first inform the Broker about them. The registration facilities allow a module to record with the Broker the services that the module provides, the command that should be used to start up the module on demand, and other information. The SBW Broker stores this in a disk file, so that the information provided by modules is persistent between start-up and shutdown of the modules and the Broker.

2.1.1 Service Categories

Each service is described by a unique name, a humanly-readable name, and a service category. Services in SBW are categorized hierarchically as a tree in which the leaves are services. Each level in the hierarchy is named and can be uniquely described via a text string very much like directory pathnames in Unix or Windows.

The purpose of this facility is to support a hierarchy of programming interfaces. Each level has an associated, documented interface. Descendants inherit or extend the interface of their parents. Categorizing services in this way allows other applications to find services with known interfaces without having to know about specific modules. New modules, by complying with a given interface, can extend the behavior of existing applications without requiring those applications to be rewritten.

To give an example, one could define a top-level category, "Analysis", with a service interface consisting of one method:

```
void doAnalysis(string SBML)
```

Then one could have a subcategory of "Analysis" called "Analysis/Simulation", with a service interface consisting of two methods:

```
void doAnalysis(String SBML)
void ReRun()
```

Service category names can consist of any sequence of non-control characters excluding '\n' and '/' but including space. Service category strings as used in the API are sequences of one or more service category level names separated by a forward slash ('/') character.

2.1.2 Adapting Applications to Use SBW

We strove to create APIs that provide a natural interface in each of the different languages for which we have implemented libraries so far. By "natural", we mean that it uses a style and features that programmers accustomed to that particular language should find familiar. We hope that application developers will find it relatively easy to introduce SBW interoperability into their software. To give some idea of what is involved, here is a summary of the steps a developer would follow to adapt a particular application for use as a module in SBW:

1. Decide on the services that will be provided by the module to clients.
2. (Optionally) Categorize each service, using as a starting point the existing SBW service hierarchy.
3. For each service, define its methods along with their parameters and return value(s).
4. If necessary, implement the methods as they are intended to be seen by other modules.
5. Add calls in the application's main routine to the SBW module registration methods as described below.
6. Compile the application with the SBW API library (if appropriate for the programming language in use).

The resulting application will be able to run in three modes: a regular, non-SBW mode; a *registration* mode; and a *module* mode. By convention, the registration mode is invoked by starting the module with the command-line argument `-sbwregister`; it tells SBW to contact the Broker once, register the services declared by the module (if the module's author so desires), and exit. The module mode is invoked by starting the module with the command-line argument `-sbwmodule`; it allows the module to run, connecting to SBW and making its services (if any) available to other SBW modules. The non-SBW mode is invoked by not supplying either of these arguments.

3 A Tutorial on Programming with SBW in Java

We describe in this tutorial several programming scenarios that use the SBW Java API. Section 4 provides a detailed reference manual for all of the methods and objects used in this section, plus others that are not discussed in this tutorial.

3.1 Preliminary Note about Data Types

As all programmers know, different programming languages have different definitions for basic data types. In order to allow an SBW module written in one language to communicate with another written in a different language, it is necessary in some instances for SBW to define its own data types. We have tried to minimize this and use each programming language's natural types as much as possible. Table 1 shows the mappings for various programming languages currently supported in SBW.

SBW Signature	Java	C++	C	Python	Delphi
string	String	std::string	char *	string	string
int	int	Integer [a]	SBWInteger	*int*	integer
double	double	Double [a]	SBWDouble	*float*	double
boolean	boolean	bool	SBWBoolean	*int*	boolean
byte	byte	unsigned char	unsigned char	string	byte
array	*array*	std::vector [b] or *array*	*array*	array [c]	SBWArray
list	java.util.List	DataBlockWriter [a] DataBlockReader [a]	SBWDataBlockWriter * SBWDataBlockReader *	*list*	SBWList
complex	SBWComplex	std::complex<double>	SBWComplex	complex	TSBWComplex

Table 1: *Data types supported in SBW and their corresponding programming language types. The "SBW Signature" types are those permitted in service method signatures; they are described in Section 4.2.2. The italicized names* array *and* list *represent the natural versions of these data types; i.e., in Java, arrays are such things as "*`int[]`*", "*`byte[]`*", etc.* `Integer` *and* `SBWInteger` *are to defined to be a 32-bit signed integer (the same as the primitive* `int` *and* `integer` *types in the other languages).* `Double` *and* `SBWDouble` *are defined to be a double-precision floating-point number in IEEE 754 format (the same as the primitive* `double` *type defined in the other languages).* [a]*SBW symbols in the C++ library are defined in the namespace* `SystemsBiologyWorkbench`*.* [b]*In the C++ library,* `std::vector` *is used for 1-D arrays and raw C++ arrays are used for 2-D arrays.* [c]*In Python, 1-D and 2-D arrays are implemented using the* `array` *type from the* Numerical Python *package (Ascher et al., 2001).*

3.2 Preliminary Note about Accessing the APIs

The SBW Java interface library is packaged in the same familiar form as most other Java tools: it is a Java *JAR* archive (Falangan, 1999) of class files that a client application loads dynamically when it needs it. SBW is provided in two forms: an optimized version (in a file named `SBWCore.jar`) and a debugging version (in a file named `SBWCore-debug.jar`). Both forms of the SBW Java interface library depend on other third-party open-source libraries that are distributed with SBW. All of these files are provided with the SBW installation.

To use the SBW library in an application, it is necessary to instruct both the Java compiler and the Java run-time execution environment about where the necessary files are located. To avoid getting sidetracked by too many details here, we delay presenting the instructions for doing this until Section 4.1. In the rest of the present section, we omit discussions of compiling code in order to concentrate on how to program with SBW.

3.3 Calling a Known Module

The simplest use of SBW consists of invoking services on a module. The simplest case of invoking services on a module is when the module identification name and service identification names are known by the programmer in advance.

We begin with a simple example that involves calling a service named “Trig” on a module named “edu.caltech.trigonometry”. An actual implementation of this module is described in the next section (and also available with the SBW source release in the `src/tutorials/Java` area); for the moment, let us assume that the interface for service “Trig” consists of two methods:

```
double sin(double)
double cos(double)
```

Figure 2 shows the text of a Java program fragment that illustrates making a call to one of these methods. In this example, the program connects to SBW using the `connect` method on *line 16* class `SBW`, executes one method on another module, and then disconnects from SBW using the `disconnect` method. *line 22*

Let us assume there is a module identified as “edu.caltech.trigonometry” and that it has a service named “Trig”. We want to invoke the method `sin` on this service. The first step is to access an instance of the module from our program and obtain an object representing it. This is accomplished using `SBW.getModuleInstance`. Then, we obtain an object representing *line 17* the specific desired service on that module instance, using `findServiceByName` on the `Method` *line 18*

```
 1 import java.util.*;
 2
 3 import edu.caltech.sbw;
 4
 5 interface Trigonometry
 6 {
 7   double sin(double x) throws SBWException;
 8   double cos(double x) throws SBWException;
 9 }
10
11 ...
12
13   static public double trigonometry(double x)
14   {
15     try
16     {
17       SBW.connect();
18       Module module = SBW.getModuleInstance("edu.caltech.trigonometry");
19       Service service = module.findServiceByName("Trig");
20       Trigonometry trig = (Trigonometry)service.getServiceObject(Trigonometry.class);
21       double result = trig.sin(x);
22       module.shutdown();
23       SBW.disconnect();
24
25       return result;
26     }
27     catch (SBWException e)
28     {
29       e.handleWithDialog();
30     }
31
32     return 0;
33   }
34 ...
```

Figure 2: *Java example of calling a known module (“edu.caltech.trigonometry”) and service (“Trig”).*

object returned from the previous call. The SBW Java interface makes accessing this service convenient through the `getServiceObject` method which returns a proxy object for the service *line 19* (here assigned to the variable `trig`). The object permits calling the `sin` method directly on the service.

The construction of the proxy merits some additional explanation. The `getServiceObject` dynamically constructs a Java `Object` having the interface given as its argument; the result needs to be cast to the interface in a manner that will be familiar to Java programmers. Thus, *line 19* the only requirement for using proxy objects like this is the availability of a Java interface definition so that the Java compiler will accept the method calls on the proxy object. If desired, this interface may define only a subset of the methods on the implementing service (in other words, the real service may have other methods not listed in the Java interface used in the call to `getServiceObject`); however, the methods that *are* defined should match the signatures defined on the actual implementation. Note also that although the interface here has been defined using the Java interface mechanism, the implementation of the service (in the remote module) does not have to be in Java—the implementing module can be written in any other language. The interface here only defines how a Java client program sees the interface.

The call to the `sin` method causes (underneath it all) messages to be sent to, and returned from, *line 20* the implementing module. This is a blocking operation—the caller waits for a reply. (Non-blocking operations can also be implemented, simply by putting the call within a thread.)

The remaining code in Figure 2 on the page before shows the use of `try-catch` clauses to handle exceptions that may be thrown by SBW as well as calls on the proxy object. All SBW exceptions are derived from class `SBWException`. `SBWException` has a method, `handleWithDialog()`, *line 28* which displays information on the exception in a modal dialog to the user to facilitate exception handling. As an alternative, `SBWException` provides methods to directly access the messages associated with an exception; these can be used by a client application to do its own display and handling of exceptions. `SBWException` is documented in Section 4.7.

As a final note, it worth mentioning that the example of Figure 2 is somewhat contrived, in that real applications would not call `connect()` and `disconnect()` around every invocation of a service method. Instead, a real-life application would most likely connect to SBW once when starting and then disconnect at the very end before exiting.

3.4 Implementing a Service

The previous section presented an example of accessing a known service through SBW. In this section, we describe the implementation of the module “edu.caltech.trigonometry” that provides the “Trig” service used in that example.

We begin by creating the underlying service functionality. In Java this is straightforward: we create a class that implements the trigonometry operations `sin` and `cos`. The method signatures (that is, their names, parameters and return types) must match those expected by the client. Figure 3 on the following page shows the definition of a class that implements the necessary functionality. Note that this part of the module has no SBW-specific code in it.

So much for the methods `sin` and `cos`. Now, how does the module’s `main` method advertise this service to SBW, and how must the module’s `main` be structured?

The implementation of a module centers around SBW’s `ModuleImpl` class. This class provides a convenient, high-level interface encapsulating all the operations necessary for defining modules and services. Figure 4 on the next page shows the main routine for our example module. The first part of the implementation is the creation of a `ModuleImpl` object. The form of the *line 11* constructor used in this example is a simple one that requires only one argument: a human-readable display name for the module. The constructor automatically assigns a unique name to the module based on the package in which the definition is placed (in this case, the unique name becomes “edu.caltech.trigonometry”).

```
package edu.caltech.trigonometry;

import java.lang.Math;

class Trig
{
  public double sin(double x)
  {
    return Math.sin(x);
  }

  public double cos(double x)
  {
    return Math.cos(x);
  }
}
```

Figure 3: *Implementation of the functionality behind service "edu.caltech.trigonometry".*

The following line of code in Figure 4 adds the service named "Trig" to the `ModuleImpl` object: *line 13/14*

```
moduleImp.addService("Trig", "sin and cosine functions",
                        "trigonometry", Trig.class);
```

The first argument of `addService` is a unique name for the service (unique to the module); the second argument is a human-readable display name for the service; the third is the service category; and the last is the class that implements the service. The class definition is scanned by SBW for public methods, and these become the methods offered by the service.

The last step is starting up the module. As mentioned in Section 2.1.2, an SBW-enabled application can run in three modes: without SBW, in SBW *registration* mode, and in *module* mode. The `ModuleImpl` class provides a convenience method, `run`, which takes care of switching *line 15* between the registration and module modes at run time. The determination of which mode to use is made on the basis of a flag passed on the command-line to the application when it is started up. Note the `args` array passed as an argument to the `run` method in Figure 4; *line 15*

```
 1  package edu.caltech.trigonometry;
 2
 3  import edu.caltech.sbw;
 4
 5  class TrigApplication
 6  {
 7    public static void main(String[] args)
 8    {
 9      try
10      {
11        ModuleImpl moduleImp = new ModuleImpl("Trigonometry");
12
13        moduleImp.addService("Trig", "sin and cosine functions",
14                                "trigonometry", Trig.class);
15        moduleImp.run(args);
16      }
17      catch (SBWException e)
18      {
19        e.handleWithDialog();
20      }
21    }
22  }
```

Figure 4: *Module implementation code for "edu.caltech.trigonometry".*

this is the array of command-line arguments that was handed to the application's `main`. If the application is started with the command-line flag `-sbwregister`, the call to `ModuleImpl`'s `run` method registers the module and its services with the SBW Broker and exits; if the application is started with `-sbwmodule`, it connects to the Broker, notifies it that the module is providing services, and returns, letting the module run until it shuts down or SBW is disconnected. If neither flag is given to the application, `run` does nothing.

The use of the flags `-sbwregister` and `-sbwmodule` is only a suggested convention. A programmer can chose to use other flags (or other techniques) simply by not calling `ModuleImpl`'s `run` method, and instead calling the corresponding methods on `ModuleImpl` itself. (These methods are described in Section 4.5.)

3.5 Finding Services in a Given Category

In some applications with GUI interfaces, it can be useful to let users select a service from a list. An example might be a tool that allows users to create and edit a visual representation of a computational model, then send the model to a separate simulation and analysis engine. The choice of simulation/analysis tools could be presented as a list of services available to the user. At the programming level, the different simulation and analysis services must have some minimal defined interface to call. They can therefore be categorized. To construct the list of services to present to the user, the model editing tool first needs to obtain a list of available services from SBW. Once the user makes a selection, the application then must call one or more methods on the selected service. This section describes how this can be achieved using SBW.

The complete code is somewhat long because of all the GUI calls required, so we leave it to Appendix A.3 and here concentrate only on the most salient elements. Figure 5 on the following page shows the code with only the lines of interest included.

We begin by describing the implementation of GUI components for displaying the list of services. We can create a list box in Java's *Swing* GUI framework using the `JList` class object, and in Figure 5 on the next page, the list box is assigned to a variable named (appropriately enough) `list`. Next, we create a list of services that will be inserted into the `JList` pointed to by `list`. *line 10* This is a good opportunity to introduce another SBW concept, that of a *service descriptor*, represented programmatically by the SBW class `ServiceDescriptor`. The need for descriptors that are separate from `Service` objects stems from the fact that an application (such as our current example) may need to manipulate information about services that are not yet running. `Service` class objects represent services that *are* running and available for access; by contrast, `ServiceDescriptor` objects are used to describe basic aspects of a service such as its name and category—aspects which do not require the service's implementing module to be running.

In SBW, an array of service descriptors is obtained by calling `SBW.findServices`. In our current example, the array returned by the call to `SBW.findServices` is used to initialize the *line 17* contents of variable `list`. For purposes of this example, we assume that there exists a service *line 18* category called "Analysis". Setting up `list` thus looks like this:

```
ServiceDescriptor[] descriptors = SBW.findServices("Analysis");
list.setListData(descriptors);
```

Displaying the data in the list requires the use of a Java renderer object. In the code of Figure 5 on the following page, we create a class called `AnalysisCellRenderer` that implements *line 42* the Java `ListCellRenderer` interface. The important elements of the code are the following:

```
public Component getListCellRendererComponent(JList list, Object value, int index,
                                              boolean isSelected, boolean hasFocus)
{
  String name = ((ServiceDescriptor) value).getDisplayName();
  return defaultRenderer.getListCellRendererComponent(list, name, index,
                                                      isSelected, cellHasFocus);
}
```

```
 1  ...
 2  interface Analysis
 3  {
 4    void doAnalysis(String sbml);
 5  }
 6  ...
 7  public class AnalysisDriverFrame
 8  {
 9    ...
10    JList list = new JList();
11    ...
12    private void jbInit() throws Exception
13    {
14      ...
15      try
16      {
17        ServiceDescriptor[] descriptors = SBW.findServices("Analysis");
18        list.setListData(descriptors);
19        list.setCellRenderer(new AnalysisCellRenderer());
20      }
21      ...
22    }
23
24    void OKButton_mouseClicked(MouseEvent e)
25    {
26      if (!list.isSelectionEmpty())
27      {
28        String sbml = SBMLEditorPane.getText();
29
30        try
31        {
32          ServiceDescriptor descriptor = (ServiceDescriptor) list.getSelectedValue();
33          Service service = descriptor.getServiceInModuleInstance();
34          Analysis analysis = (Analysis) service.getServiceObject(Analysis.class);
35          analysis.doAnalysis(sbml);
36        }
37        ...
38      }
39    }
40  }
41
42  public class AnalysisCellRenderer implements ListCellRenderer
43  {
44    private DefaultListCellRenderer defaultRenderer = new DefaultListCellRenderer();
45
46    public Component getListCellRendererComponent(JList list, Object value, int index,
47                                                  boolean isSelected, boolean hasFocus)
48    {
49      String name = ((ServiceDescriptor) value).getDisplayName();
50      return defaultRenderer.getListCellRendererComponent(list, name, index,
51                                                          isSelected, hasFocus);
52    }
53  }
54  ...
```

Figure 5: *Module implementation code for "edu.caltech.trigonometry".*

The `getListCellRendererComponent` method of our renderer is called by the Java GUI framework. The parameter `value` is an object (one of our `ServiceDescriptor` objects) from the list data in `list`. We invoke the method `getDisplayName` on this object to obtain a human-readable name of the service as a Java `String` object, and then pass that to the default Java GUI rendering methods for display. *line 49*

Now that we have described the initialization of the list box, we can consider what happens when the user makes a selection. When the user chooses an item, our code must use the `ServiceDescriptor` object returned by the Java GUI calls to invoke a method on the chosen service. First, we need to extract the item selected from the list after some event such as the user clicking an "OK" button. In Java Swing, this is easily done using `getSelectedValue` on the `list` object: *lines 24–40*

```
ServiceDescriptor descriptor = (ServiceDescriptor) list.getSelectedValue();
```

After getting hold of the service descriptor, we must ask SBW for an instance of the service before we can invoke one of its methods. We can obtain an instance of the module that implements the service by using the method `getServiceInModuleInstance` on the given `ServiceDescriptor`: *line 28*

```
Service service = descriptor.getServiceInModuleInstance();
```

This method first checks whether the module that implements the requested service needs to be started up, and starts it if so. It then returns a `Service` object corresponding to it.

Let us assume that the "Analysis" category of services has a method, `doAnalysis`, that takes as an argument a representation of the model to be analyzed. We define an interface class for the *Analysis* category, and use this in an SBW call to create a proxy object for the service: *lines 2–5*

```
Analysis analysis = (Analysis) service.getServiceObject(Analysis.class);
```

Finally, we can invoke the `doAnalysis` method on the "Analysis" service. *line 35*

3.6 Calling a Known Service Having a Complex Return Value

The examples above involved simple parameters and return values to and from service methods. However, SBW supports more elaborate data structures than those used so far. In this section, we present an example of calling a service method having a complex return value.

In order to allow method parameters and return values to be described in a language-independent manner, SBW uses a simple notation for method signatures. The notation is fully defined in Section 4.2.2. For our purposes in this section, we explain the meanings of just those parts of the notation we actually need here.

The following is a signature definition for a method called `loadSBMLModel` that accepts a single string and returns an array of lists. Each item in the array of lists consists of a string and a `double` floating-point value.

```
{string, double}[] loadSBMLModel(string sbml)
```

The data type used in Java to represent a list is the `java.util.List` class. Let us assume that the method above is defined on an interface called `Simulator`, as shown in Figure 6.

```
import java.util.List;
import edu.caltech.sbw;

interface Simulator
{
    List[] loadSBMLModel(String sbml) throws SBWException;
}
```

Figure 6: *Definition of the `Simulator` interface.*

Let us also assume that there already exists in our program an instance of a `Service` object that implements this interface, assigned to a variable named `service`. As shown before, we can ask SBW to create a proxy object for this interface as follows:

```
Simulator simulator = (Simulator) service.getServiceObject(Simulator.class);
```

Given this, we can call method `loadSBMLModel` on the proxy object and get the array of lists it returns. Obtaining the individual elements of each list is a simple matter of calling some standard Java list element extraction methods. To illustrate this, Figure 7 presents a code fragment showing one approach to iterating through the list returned by `loadSBMLModel` and extracting the two elements in each list. In this example, the result of the call to `simulator.loadSBMLModel(sbml)` is stored in the variable `parameters`. A `for` loop then *line 5*
iterates over each element in the `parameters` array, using the list operator `get` to extract the *lines 10, 11*
two components of the lists.

```
 1  String sbml;
 2  ...
 3  // Assume that something assigns a value to variable sbml.
 4  ...
 5  List[] parameters = simulator.loadSBMLModel(sbml);
 6  ...
 7  for (int i = 0; i < parameters.length; i++)
 8  {
 9      List parameter = parameters[i];
10      String name = (String) parameter.get(0);
11      double value = (Double) parameter.get(1).doubleValue();
12
13      // Do something with parameter data here...
14  }
```

Figure 7: *Example of iterating through a list of return values.*

4 SBW Java API Reference

This section is a reference for the different components of the high-level SBW Java API. The classes in the API are summarized in Table 2 and described in detail in the paragraphs that follow.

As mentioned elsewhere in this manual, it is also possible to interact with SBW using a lower-level API. This API bypasses most of the classes and methods discussed in this section and instead involves exchanging messages directly between modules. It is conceptually simple, but it is more error-prone and tedious to program, hence the development of the high-level API presented here. Nonetheless, the low-level API may be useful for some situations. It is described in Section 5.

Class	Role	Section	Starting Page
`Module`	Represents a running module instance and provides access to basic information about the module as well as ways of listing and searching the module's services	4.3	22
`ModuleDescriptor`	Describes a module that may or may not be running; the information corresponds to the static data recorded in the SBW Broker's registry	4.4	23
`ModuleImpl`	Used for defining a module, including its services; only service providers need to interact with this class	4.5	24
`SBW`	Provides static methods for general SBW operations such as connecting to SBW and searching for services and modules	4.6	28
`SBWException`	The base class of exceptions thrown by SBW methods and client-side service proxies	4.7	32
`SBWListener`	Used for defining call-backs for SBW notification of certain events such as module startups	4.8	34
`Service`	Represents a service and provides access to the service methods implemented by a module instance	4.9	35
`ServiceDescriptor`	Describes basic information about a service, such as its name and category; also provides a way to get a `Service` object	4.10	37
`ServiceMethod`	Describes a method on a service and provides access to the method signature	4.11	38
`Signature`	Represents a method signature in a structured form and allows translation to and from text string form	4.12	39
`SignatureElement`	A component of a method signature consisting of a data type and optionally a variable name	4.13	40
`SignatureType`	Represents a data type in a method signature	4.14	41

Table 2: *The classes defined by the SBW Java API, presented in alphabetical order.*

4.1 Programming with the SBW Java API Library

The SBW distribution and the code examples in this manual are designed for use with Java Development Kit (JDK) version 1.3 (Sun Microsystems, 2001). For the sake of this discussion, let us assume that when you run the commands `java`, `javac` and `jar` in a shell window on your computer (under both Linux and Windows), the versions of these commands that are executed are from JDK 1.3.

In what follows, we use the term *`SBW_HOME`* to stand for the path to the SBW installation directory on your computer. If you installed SBW using the self-extracting installer, this directory is where you told the installer to place SBW. If you configured and built the system from the source files, this will be the directory supplied as the `--prefix` argument to the `configure` program supplied with SBW. Throughout the discussions below, you will have to replace the text *`SBW_HOME`* in the commands below with the actual path on your system.

Figure 8 summarizes the directory structure of the SBW installation, with *`SBW_HOME`* being the top level. All of the SBW library files are located in the `lib` subdirectory.

```
SBW_HOME
  |
  |-- bin
  |
  |-- docs
  |
  |-- include
  |
  |-- lib
  |     |
  |     |-- SBWCore.jar
  |     |-- SBWCore-debug.jar
  |     |   ...
  |
  `-- modules
```

Figure 8: *The SBW installation root directory structure and the location of critical JAR files such as `SBWCore.jar`. "`SBW_HOME`" is not meant literally as a pathname, but rather signifies the root of the directory where SBW is installed on your system. Its exact value is platform- and installation-specific.*

4.1.1 Compiling Java Applications with SBW

As mentioned in Section 3.2, there are two forms of the SBW Java interface library: an optimized version (in a file named `SBWCore.jar`) and a debugging version (in a file named `SBWCore-debug.jar`). To simplify the following discussions, we refer only to `SBWCore.jar`, but `SBWCore-debug.jar` could just as well be substituted in its place.

As is well-known to Java programmers, a program's Java source files must be placed in a directory hierarchy that matches the package structures. For example, if you were developing a package named `edu.caltech.simulator`, the Java source files would need to be placed in the subdirectory `edu/caltech/simulator`. In the following discussions, we use the italicized word *`package`* to stand for an arbitrary package directory hierarchy.

To compile a set of Java source files for use with SBW, you need to execute a command of the following form (all on one line) at the top of the hierarchical *`package`* directory:

```
javac -classpath SBW_HOME/lib/SBWCore.jar:. package/*.java
```

The `SBWCore.jar` library file incorporates within it a few third-party open-source libraries needed for its operation. For portability, `SBWCore.jar` does not have dependencies on external packages—it is a standalone library, analogous to a statically-linked binary in other programming languages.

4.1.2 Running SBW-Enabled Java Applications

The simplest way to run your application, once your Java source files are compiled, is to use a command such as the following from the top of the *package* directory:

```
java -classpath SBW_HOME/lib/SBWCore.jar:. main
```

Here, *main* refers to the class that contains the entry point to your application (that is, the class containing the `main` routine in your program). For example, this might be a path such as `edu/caltech/simulator/Main`, if the main routine were located in the file `edu/caltech/simulator/Main.java`.

4.1.3 Debugging SBW-Enabled Java Applications

When developing SBW applications, it can be extremely useful to know the dialogue that takes place between your application and the rest of SBW, particularly the SBW Broker. The SBW debugging library `SBWCore-debug.jar` is provided exactly for this reason. To use it, simply run your application using `SBWCore-debug.jar` instead of `SBWCore.jar`. For example, from the top of your *package* directory, run a command of the form

```
java -classpath SBW_HOME/lib/SBWCore-debug.jar:. main
```

The result will be that all SBW calls produce copious debugging output on the standard output stream. Each line begins with a time stamp, the name of the thread involved, the SBW class, method, and line number, followed by a debugging message. Figure 9 on the following page provides an illustrative sample from the SBW Browser module.

4.1.4 Creating Standalone Files for a Java Application

The command given above for running an application is simple and useful during development. However, it is cumbersome, and moreover the files are located in a specific directory that have to be written out as a complete path every time the program is run.

Thankfully, it is possible to create standalone Java `.jar` files that can be moved anywhere on a system and executed with a simple run command. (On MS Windows platforms, the resulting `.jar` file can even be executed by simply double-clicking its name in the Windows file browser.) To simplify the construction of these standalone Java files, we include as part of SBW a program called `sbw-make-jar`. It is located in the `bin` subdirectory of *SBW_HOME*.

To use `sbw-make-jar`, first move to the top of the *package* directory for your application. Then, execute a command such as the following:

```
SBW_HOME/bin/sbw-make-jar name.jar modulemain package/*.class
```

The command must be given three mandatory arguments: *`name.jar`* is the name that you wish to give to your application's `.jar` file; *`modulemain`* is the class that contains the entry point to your application (i.e., the class containing the `main` routine); and *`package`*`/*.class` is the set of compiled Java `.class` files for your application. The result of running this command will be a file named *`name.jar`* in the current directory.

Here is a complete example of using `sbw-make-jar`. Let us suppose that we have a module whose main class is `edu.caltech.test.Main`, and we want the `.jar` file to be named `test.jar`. If our compiled class files are all located in a subdirectory `edu/caltech/test`, then:

```
SBW_HOME/bin/sbw-make-jar test.jar edu.caltech.test.Main edu/caltech/test/*.class
```

Once you have built `test.jar` in this way, you can run your application by simply executing the following command:

```
java -jar test.jar
```

You can move `test.jar` to another directory and still be able to run it with `java -jar`. (And,

```
# java -jar Browser-debug.jar -l
[0.046 main (trace) SBWModuleRPC.connectToBroker:294] Attempting to connect to Broker
[0.075 main (trace) RuntimeProperties.reload:172] Read runtime properties from
/home/mhucka/.sbw/runtime/erato.cds.caltech.edu/run.properties
[0.089 main (trace) RuntimeProperties.reload:172] Read runtime properties from
/home/mhucka/.sbw/runtime/erato.cds.caltech.edu/run.properties
[0.099 main (trace) SBWModuleRPC.connectToBroker:308] Found sbw.module.port in SBW properties file
[0.117 main (trace) SessionKey.loadKey:79] Loading session key from
'/home/mhucka/.sbw/runtime/erato.cds.caltech.edu/key'
[0.128 main (trace) SBWModuleRPC.connectSocketToBroker:371] Attempting to connect socket to
localhost:10002
[0.131 main (trace) SBWModuleRPC.connectMessageStreams:409] Attempting to connect anonymously
[0.165 main (trace) SBWModuleRPC.connectMessageStreams:433] Successfully shook hands with Broker
[0.171 receiver-thread (trace) SBWModuleRPC.runReceiverThread:457] Receiver thread started
[0.172 main (trace) SBW.getBrokerService:709] Obtaining instance of Broker interface.
[0.175 main (trace) SBW.getServiceNamesFromModule:751] Querying module -1 about its services
[0.195 main (trace) SBWRPC.transmit:397] Transmitting message (length = 26) to module -1
[0.196 receiver-thread (trace) SBWRPC.receive:287] Module 5 receiving 'reply' msg
[0.198 main (trace) SBWRPC.call:206] Call sent; waiting for reply from module -1
[0.199 main (trace) RPCOutCall.waitForReply:168] Processing reply event
[0.199 main (trace) RPCOutCall.waitForReply:172] Received NORMAL_REPLY_EVENT
[0.209 main (trace) SBW.getSignatureStringsFromModule:796] Querying module -1 about methods for
service 0
[0.210 main (trace) SBWRPC.transmit:397] Transmitting message (length = 31) to module -1
[0.215 receiver-thread (trace) SBWRPC.receive:287] Module 5 receiving 'reply' msg
[0.216 main (trace) SBWRPC.call:206] Call sent; waiting for reply from module -1
[0.216 main (trace) RPCOutCall.waitForReply:168] Processing reply event
[0.217 main (trace) RPCOutCall.waitForReply:172] Received NORMAL_REPLY_EVENT
[0.323 main (trace) ServiceInvocationHandler.invoke:122] Invoking method 2 on service 0 of module
-1
[0.333 main (trace) SBWRPC.transmit:397] Transmitting message (length = 30) to module -1
[0.345 receiver-thread (trace) SBWRPC.receive:287] Module 5 receiving 'reply' msg
[0.356 main (trace) SBWRPC.call:206] Call sent; waiting for reply from module -1
[0.356 main (trace) RPCOutCall.waitForReply:168] Processing reply event
[0.369 main (trace) RPCOutCall.waitForReply:172] Received NORMAL_REPLY_EVENT
The following modules are registered with SBW:
  edu.caltech.NOM ("Network Object Model")
  edu.caltech.NOMClipboard ("Clipboard Network Object Model")
  edu.caltech.MatlabTranslator ("Matlab Translator")
  edu.caltech.sbw.tests.testserver ("TestServer")
  BROKER ("SBW Broker")
  simple ("server C implementation")
[0.412 main (trace) SBWModuleRPC.disconnect:213] Disconnecting from broker
[0.414 main (trace) SBWRPC.transmit:397] Transmitting message (length = 4) to module -1
[0.501 main (trace) SBWModuleRPC.disconnectSockets:508] Closing client-side socket connection
[0.502 main (trace) SBWRPC.cleanupPendingCalls:654] Stopping all incoming and outgoing message
threads
```

Figure 9: *Example output from using the debugging library `SBWCore-debug.jar`.*

as mentioned above, under MS Windows, you can double-click on `test.jar` in a file listing window and Windows will start the Java run-time environment on it automatically.)

The program `sbw-make-jar` accepts certain optional command-line switches, one of the most important being `-d`, which tells it to use the debugging version of the SBW library (i.e., `SBWCore-debug.jar`). You will probably wish to use this option whenever you are testing your application with SBW. A detailed explanation of the program `sbw-make-jar` and the options it accepts is provided in Appendix B.

Finally, note that you could simplify the use of this command further by adding the directory *SBW_HOME*`/bin` to your shell's command search path. This would obviate the need to type the full path to `sbw-make-jar` every time.

4.2 General Concepts in the SBW API

Before getting into the list of SBW classes, we discuss some general points about the Systems Biology Workbench framework and its implementation.

4.2.1 Relationships between the Different Elements in the SBW API

When dealing with the different objects listed in Table 2, it is useful to keep in mind the distinction between a local representation of a module, service or service method, and its implementation in an actual module instance.

The objects of class `Module` and `Service` are local representations or proxies for other, concrete objects that exist elsewhere in the SBW system. Actions taken on the proxy objects themselves, such as destroying a `Service` object, may have no effect on any real module instance. Figure 10 illustrates how the proxies are related to the objects they represent.

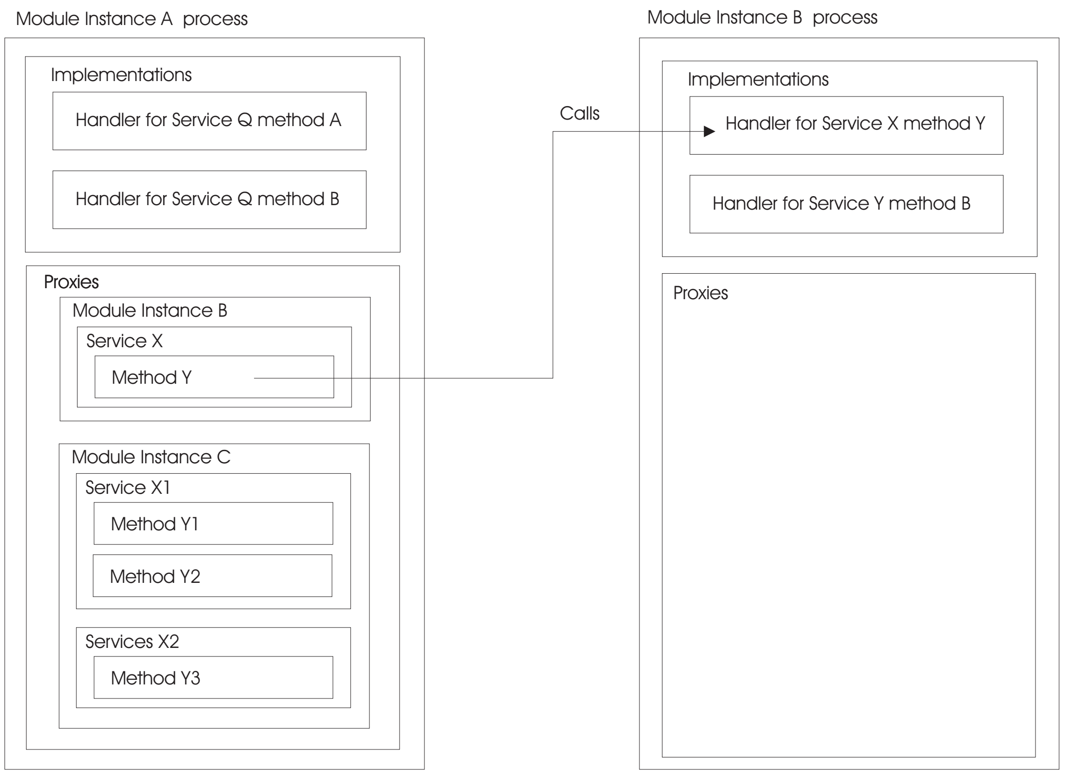
Module Instance A process
Implementations
Handler for Service Q method A
Handler for Service Q method B
Proxies
Module Instance B
Service X
Method Y
Module Instance C
Service X1
Method Y1
Method Y2
Services X2
Method Y3
Calls
Module Instance B process
Implementations
Handler for Service X method Y
Handler for Service Y method B
Proxies

Figure 10: *Relationships between proxy objects in a client and the corresponding objects in a module instance.*

SBW generally avoids caching all but the most basic information, to avoid problems that might arise if a remote module changed some part of its definition and the cached information fell out of date. However, some information is cached for efficiency. The `ServiceMethod.getName()` call, for example, does not actually contact the remote module every time. Other methods *do* involve communicating with the remote module, as for example, getting a list of its services. As will become obvious in the sections that follow, any method that may involve communicating with a remote module is declared as throwing an exception derived from class `SBWException`.

4.2.2 Method Signatures

SBW uses a textual notation for describing method names, their parameters and return values. A description of a method in this notation is called a *method signature string*. The syntax of method signatures is shown in Figure 11 on the following page.

In Figure 11 on the next page, the `Signature` element describes the signature of one method. The `Type` enumeration refers to the different possible data types in an argument or the method return value. Table 1 shows the correspondence between the values of `Type` shown here and actual programming language data types. The character sequence '`...`' is used in argument

```
Letter     ::= ‘a’..‘z’,‘A’..‘Z’
Digit      ::= ‘0’..‘9’
Space      ::= ( ‘\t’ | ‘ ’ )+
SName      ::= ‘_’* ( Letter | Digit ) ( Letter | Digit | ‘_’ )*
Type       ::= ‘int’ | ‘double’ | ‘string’ | ‘boolean’ | ‘byte’ | ArrayType | ListType| complexType
ArrayType  ::= Type Space? ‘[]’
ListType   ::= ‘{’ Space? ArgList Space? ‘}’
ArgList    ::= ( Type [Space SName] ( Space? ‘,’ Space? Type [Space SName] )* )?
ReturnType ::= ‘void’ | Type
VarArgList ::= (Space? ArgList [Space? ‘,’ Space? ‘...’] Space? ) | Space? ‘...’ Space?
Signature  ::= ReturnType Space SName Space? ‘(’ VarArgList ‘)’
```

Figure 11: *Permissible syntax of method signatures, defined using the version of EBNF notation (Extended Backus-Naur Form) used in the XML specification (Bray et al., 1998). The meta symbols '(' and ')' group the items they enclose, '[' and ']' signify that the enclosed content is optional, '*' means "zero or more times", and '+' means "one or more times".*

lists to indicate that the remainder of the arguments are variable (roughly equivalent to *varargs* in the C programming language). The `ListType` element indicates a list; it may be left empty to indicate a list whose contents are unspecified.

The following are some examples of method signatures. The following describes a method that takes a double parameter `x` as a parameter and returns a double value:

```
double f(double x)
```

The following is a method that returns an integer given a one dimensional array of integers:

```
int f(int[])
```

The following describes a method that takes a single string as argument and returns a list whose elements are a string and a double. This also demonstrates that the SBW method signature notation permits the inclusion of descriptive names such as `name` and `value`, to make signatures easier for humans to discuss.

```
{string name, double value} f(int myindex)
```

The variable names are optional annotations and are ignored for purposes of comparing signatures; the signature above is functionally equivalent to the following one:

```
{string, double} f(string)
```

The following is a method that returns an array of lists. The method takes no arguments, and the content of the returned lists is undefined:

```
{}[] f()
```

As a special feature, SBW takes an empty list specifier (i.e., `{}`) as a wild-card that matches any other list in another signature. Thus, the lists `{}` and (for example) `{x, int y}` are equal as far as signature parsing is concerned.

The following is a method that returns a complex value given two double values:

```
complex f (double, double)
```

Methods can also be declared as returning no value, using the type `void`:

```
void f(byte x, int[] y)
```

Finally, here is a method that takes a variable number of arguments but returns no value:

```
void f(...)
```

4.2.3 On Naming Modules, Services and Methods

Module Names. Each module has two names: a unique *module identification name*, and a name for display. Module identification names are used by SBW for identifying modules running in a session. They are compared by SBW on a case-sensitive basis. A module's identification name should be chosen so that it is unlikely to be equal to any other module identification name running on a particular computer. To achieve this, we suggest the following convention: use the reverse domain name, followed by a '.', followed by the module name, all in lower case. The reverse domain name is simply the reverse of an Internet domain name string affiliated with the module developer. (For the SBW team at Caltech, this is typically "edu.caltech".) An example of a module identification name following this convention is "edu.caltech.gibson".

This scheme for module identification names is voluntary. The API will still work with other naming schemes. The objective is simply to make module names likely to be unique to one host (or more specifically, to one running instance of the SBW Broker).

Service Names. Each service has two names: a unique *service identification name*, and a name for display. A service identification name must be unique to a given module when compared in a case-sensitive manner. Service identification names must follow the `SName` syntax as defined in Section 4.2.2, and by convention start with a capital letter.

Service display names are intended for use in user menus and other similar situations. They should be chosen to be fairly short and descriptive, but at the same time, fairly unique. Names that are too general, such as "Simulator", are usually not good choices.

Version numbers should *not* be included in service names (i.e., do not use a string such as "Stochastic Simulator 1.0"). Information about versions and similar attributes is best placed elsewhere such as the module's help string (see Section 4.5).

Method Names. Method names must follow the `SName` syntax defined in Figure 11 on the preceding page. By convention, they should start with a lower case letter. Method names are not compared outside of the context of method signatures.

Method signatures are compared using the same scheme as in programming languages such as C++ and Java. This means, for example, that a service may define more than one method with the same name, as long as the total signatures for the methods are different. Methods cannot differ solely on the basis of different return values (again, just as in Java and other programming languages).

4.2.4 Module Management Types

Depending on the kind of services a module implements and the style of use intended, it may be more suitable to have multiple copies (instances) of a module running, or to have only one instance running. Modules that do not keep state information are often best implemented as having only one copy running in an SBW system. For example, a module that performs data conversion operations could run as a unique instance in an SBW system, avoiding the resource usage that would result if a fresh copy were started every time that a client module requested its services.

SBW allows a module developer to designate the type of management that SBW should assume for a module. In the Java API, this is indicated by a flag to the constructors for the `ModuleImpl` class (see Section 4.5). The possible management types are:

- `ModuleImpl.UNIQUE`: Indicates that only one instance of this module should be started up by SBW, and that it should be shared among all requests for the services it implements.
- `ModuleImpl.SELF_MANAGED`: Indicates that multiple copies of the module may run and that the module manages itself.

4.2.5 Propagating Exceptions in SBW

Exceptions that occur in a module during the execution of a service method are propagated back by SBW to the module that invoked the service. If the exception was thrown deliberately by the module, for example to indicate that the caller has made a mistake in its invocation, the caller will receive an exception of class `SBWApplicationException`. The latter is the only kind of SBW exception that a module can create. Otherwise, if the exception involves a problem in SBW or incorrect use of SBW, the caller will receive an instance of an SBW exception specific to the problem. Section 4.7 discusses SBW exceptions in more detail and lists the kinds possible.

4.2.6 Distributed Computing with SBW

SBW supports the ability to run modules on multiple computers. It does this by starting an SBW Broker on each remote computer involved in response to certain forms of invocations of `SBW.getModuleInstance` (described in Section 4.6).

4.3 Class: `Module`

The `Module` class represents an instance of a module. A `Module` class object *stands for* an actual module instance, meaning the corresponding application, that is running on the user's computer or perhaps another computer. Having a reference to a `Module` object in a program implies that the SBW Broker has started up at least one instance of the corresponding real module. (This is in contrast to having an instance of a `ModuleDescriptor` object, described in the next section; a `ModuleDescriptor` describes a module but does not imply that an instance of the module is necessarily running.)

ModuleDescriptor getDescriptor()
throws SBWCommunicationException

Returns a `ModuleDescriptor` object (Section 4.4) that describes the module. The descriptor can be used to obtain the module's display name, management type, etc.

A `Module` object contains zero or more `Service` objects which represent the services implemented by the module. These services can be listed or searched by name and category using the three methods `getServices`, `findServiceByCategory` and `findServiceByName`.

Service[] getServices()
throws SBWCommunicationException

Returns an array of `Service` objects (cf. Section 4.9) representing the services implemented by this module.

Service findServiceByName(String serviceName)
throws SBWCommunicationException

Returns a `Service` object (Section 4.9) representing the service with the given name implemented by this module instance. Returns null if such a service doesn't exist on the module. The name matching performed is case-sensitive and exact, with the modification that any leading and trailing whitespace in `serviceName` are first removed before attempting the match.

Service[] findServicesByCategory(String serviceCategory)
throws SBWCommunicationException

Returns an array of `Service` objects (Section 4.9) corresponding to all the services in the given category implemented by this module instance. Returns an empty array if there is no service with a category that matches the string `serviceCategory`. The name matching performed is case-sensitive and exact, with the modification that any leading and trailing whitespace in `serviceCategory` are first removed before attempting the match.

void shutdown()
throws SBWCommunicationException

Tells this module instance to shut down.

4.4 Class: `ModuleDescriptor`

A *module descriptor* describes a module. It embodies the static data recorded by the SBW Broker in the module registry (see Section 2.1). The corresponding real module instance may or may not be running.

It may seem confusing at first that there may be no running module instance corresponding to a module descriptor returned by SBW. However, the utility of this becomes clear when one considers the need to list all modules known by the system. SBW provides methods, such as `SBW.getModuleDescriptors`, that return module descriptors for all modules known by the Broker. It would be inefficient, and in some cases downright impractical, if every query for a module descriptor resulted in a module instance being started by SBW. Thus, module descriptors are decoupled from the running state of a module.

Module descriptors are useful for finding out a module's basic characteristics, such as its name, without necessarily starting up an instance of the module.

String getName()

Returns the unique identification of the module. This name typically follows a notational convention such as that outlined in Section 4.2.3; for example, a module's unique name might be "edu.caltech.trigonometry".

String getDisplayName()

Returns the display name of the module. This is suitable for showing to users in a menu or list. Although there is no fixed limit on the length of display names, they should be kept short.

String getCommandLine()

Returns the command line used to start up this module. Depending on how the module is implemented, the command line may have been constructed automatically by SBW or it may have been explicitly supplied by the programmer.

int getManagementType()

Returns an integer indicating the *management type* of the module (Section 4.2.4). The management type indicates how the lifetime of the module is managed by SBW. The value returned is one of the following: `ModuleImpl.UNIQUE` or `ModuleImpl.SELF_MANAGED`.

String getHelp()

Returns a help string for the module. This is an optional description of the module provided by the module's author(s). It is informational only and not actually required in the definition of a module.

ServiceDescriptor[] getServiceDescriptors()
throws SBWCommunicationException

This method returns `ServiceDescriptor` objects for all of the services implemented by the corresponding module.

4.5 Class: `ModuleImpl`

The `ModuleImpl` class is used to define the implementation of a module. Its constructor defines various basic aspects such as the module's name, and the `addService` method described below allows the definition of services associated with the module.

In addition to services, all module definitions have five pieces of information associated with them: (1) a unique name, (2) a display name, (3) a management type, (4) a class to use as the main class that will be invoked to start up the module, and (5) a help or documentation string. There are several versions of the constructor for `ModuleImpl`, providing increasingly more control over the definition of the module. The simpler forms derive some of their information dynamically from the Java stack trace at the time the method is called.

ModuleImpl(String displayName)
throws SBWModuleDefinitionException

Defines a module whose display name is `displayName` and whose remaining parameters are set to default values as follows:

- The module's unique name is determined from the package in which this constructor is called; for example, if the package name `edu.caltech.trigonometry`, the module's unique name is assigned to be "edu.caltech.trigonometry".
- The module's management type is set to `ModuleImpl.SELF_MANAGED`.
- The module's main class is set to be the class that invoked this constructor. This is determined by rummaging through the Java stack at run-time.
- The module's help string is set to the empty string.

The other constructors provide ways of explicitly setting the various parameters.

ModuleImpl(String uniqueName, String displayName, int type)
throws SBWModuleDefinitionException

Defines a module with a unique name given by `uniqueName`, a display name given by `displayName`, and a management type given by `type`. The module's main class is set to be the class that invoked this constructor, and the help string is set to the empty string. The value of the parameter `type` must be one of the predefined constants `ModuleImpl.UNIQUE` or `ModuleImpl.SELF_MANAGED`. (See Section 4.2.4 on page 20 for information about the management types.)

ModuleImpl(String uniqueName, String displayName, int type, Class moduleMainClass)
throws SBWModuleDefinitionException

Defines a module with a unique name given by `uniqueName`, a display name given by `displayName`, a management type given by `type`, and a main class given by `moduleMainClass`. The module's help string is set to the empty string.

ModuleImpl(String uniqueName, String displayName, int type,
Class moduleMainClass, String helpString)
throws SBWModuleDefinitionException

Defines a module with a unique name given by `uniqueName`, a display name given by `displayName`, a management type given by `type`, a main class given by the Java class in `moduleMainClass`, and a help string given by `helpString`.

All of the constructors above use the module's main class to create a default command line for starting up the module. This command line is used by the SBW Broker. The default has the following form:

```
/path/to/java -classpath classpath ModuleMainClass -sbwmodule
```

where the value of *classpath* is determined from the path to the SBW home directory, and the path to java is the path to the `java` executable that was used to start this module. This works for most modules (including modules packaged as JAR files using the `sbw-make-jar` utility), but in some situations it may be important to have direct control over the command line used by the SBW Broker. This is the purpose of the following method.

void setCommandLine(String commandLine)

Sets a specific command line for starting up this module. The default command for starting up the module is based on the main class given to the constructor method (see above). The command supplied must include the path to the java executable and the `-sbwmodule` argument. It should contain at least the following components:

```
/path/to/java -classpath classpath ModuleMainClass -sbwmodule
```

where `ModuleMainClass` is the class in which the `main` method for this module is located. Note that no verification is performed on the validity of the given command line. Such verification is very difficult to do (and in the most general case would be tantamount to solving the Turing halting problem.) SBW and client modules will not learn whether the given command line correctly starts up the module until the next time that SBW attempts to run it.

In the Java API for SBW, a single object is used to represent all of the methods of a service. The association of a service with its definition can be done using either an object or a Java interface given to one of the methods below. SBW inspects the object or interface given and defines the service as consisting of all the public methods on the object or interface.

void addService(String uniqueName, String displayName, String category,
Class implementationClass, String helpString)
throws SBWIncorrectCategorySyntaxException, SBWModuleDefinitionException

Adds a service to the list of services provided by this module in the module registry. This method is normally called from a module in registration mode. Parameter `uniqueName` is a unique string that identifies the service; parameter `displayName` is a human-readable name, for such uses as displaying in a menu; parameter `category` is the category into which the service belongs; and `implementationClass` is the class of the object that implements the service. The argument `helpString` can be used to provide a string that summarizes the purpose of the service; this help string is retrievable by other modules through the SBW Broker.
If the service is already defined on this module, it is redefined. Equality is determined on the basis of a case-sensitive exact match against `uniqueName`.

void addService(String uniqueName, String displayName, String category,
Object implementationObject, String helpString)
throws SBWIncorrectCategorySyntaxException, SBWModuleDefinitionException

Adds a service to the list of services provided by this module in the module registry. This method is normally called from a module in registration mode. Parameter `uniqueName` is a unique string that identifies the service; parameter `displayName` is a human-readable name, for such uses as displaying in a menu; parameter `category` is the category into which the service belongs; and `implementationObject` is an instance of the object that implements the service. (Note the distinction in this argument between this method and the previous one—here it is an object, whereas the method above requires a class.) The argument `helpString` can be used to provide a string that summarizes the purpose of the service; this help string is retrievable by other modules through the SBW Broker.
If the service is already defined on this module, it is redefined. Equality is determined on the basis of a case-sensitive exact match against `uniqueName`.

void addService(String uniqueName, String displayName, String category,
Class implementationClass)
throws SBWIncorrectCategorySyntaxException, SBWModuleDefinitionException

Adds a service to the list of services provided by this module in the module registry. This method is normally called from a module in registration mode. Parameter `uniqueName` is a unique string that identifies the service; parameter `displayName` is a human-readable name, for such uses as displaying in a menu; parameter `category` is the category into which the service belongs; and `implementationClass` is the class of the object that implements the service. The help string for the service is set to the empty string.
If the service is already defined on this module, it is redefined. Equality is determined on the basis of a case-sensitive exact match against `uniqueName`.

void addService(String uniqueName, String displayName, String category,
Object implementationObject)
throws SBWIncorrectCategorySyntaxException, SBWModuleDefinitionException

Adds a service to the list of services provided by this module in the module registry. This method is normally called from a module in registration mode. Parameter `uniqueName` is a unique string that identifies the service; parameter `displayName` is a human-readable name, for such uses as displaying in a menu; parameter `category` is the category into which the service belongs; and `implementationObject` is an instance of the object that implements the service. (Note the distinction in this argument between this method and the previous one—here it is an object, whereas the method above requires a class.) The help string for the service is set to the empty string.
If the service is already defined on this module, it is redefined. Equality is determined on the basis of a case-sensitive exact match against `uniqueName`.

Note that simply creating a `ModuleImpl` object with one of the constructors above does not actually define the module to the SBW Broker. It is not until either of the methods `registerModule` or `enableModuleServices` are called that SBW contacts the Broker.

void registerModule()
throws SBWModuleDefinitionException, SBWCommunicationException

Connects to the SBW Broker and sends the registration information for this module and its defined services, then returns. Exceptions may be thrown if the Broker determines that there is something wrong with the module definition, or if there is a problem communicating with the Broker.

void enableModuleServices()
throws SBWCommunicationException

Whereas the `registerModule` method registers the module with the SBW Broker and returns, this method notifies the Broker that the module is now operational and able to handle calls for its services.

To simplify the task of implementating simpler modules, `ModuleImpl` includes the following method that handles, in a generic fashion, the task of invoking either `registerModule` or `enableModuleServices`.

void run(String[] args)
throws SBWModuleDefinitionException, SBWCommunicationException

Convenience function for performing the tasks required for handling the `-sbwregister` and `-sbwmodule` options to a module. This method must be passed the array of command-line arguments passed to the application. This then checks for the presence of `-sbwregister` and `-sbwmodule`, and acts as follows:

1. If the flag `-sbwregister` is found first in the array of command-line arguments given to the program, this method invokes the `registerModule` method followed by Java's `System.exit(0)` call.
2. If the flag `-sbwmodule` is found first in the array of command-line arguments, this method invokes `enableModuleServices` and returns.

The calling routine should perform whatever tasks it would after enabling module services. Most modules will not need to perform anything else and should simply return. The exceptions thrown by this method are simply propagated from the calls to `registerModule` and `enableModuleServices`.

4.6 Class: `SBW`

The `SBW` class provides methods for various fundamental operations. All of the public methods on this class are static.

static void connect()
throw SBWBrokerStartException, SBWCommunicationException

Connect to SBW on the local computer as a client. The connection is anonymous. Other modules querying the Broker about which modules are connected to SBW will be informed that a module is connected, but without a name or other information.
Connections made using this call are suitable when a module will not provide any services. If a module intends to provide services to other modules, the connection should be performed implicitly using the methods on the class `ModuleImpl` (see Section 4.5).
A client must connect to SBW before performing other operations such as querying the system for services. (See the example in Figure 2 on page 7 for an illustration of the use of this method.)
A client can only be connected to one SBW Broker at a time; attempting to call `connect()` twice (without calling `disconnect()` in the interim) will result in an exception. Connection attempts that end in failure (e.g., due to a network interruption) will also result in an exception.

static void connect(String hostNameOrAddress)
throw SBWBrokerStartException, SBWCommunicationException

Deprecated method. The introduction of proper networked distributed operation in SBW version 1.0 makes this method obsolete. Programs should always connect to their local Broker using `connect()` and connect multiple Brokers together using `link(hostNameOrAddress)`.
Connects (as a client) to the SBW Broker running on the computer identified by the host name or IP address in parameter `hostNameOrAddress`. The connection is anonymous. Other modules querying the Broker about which modules are connected to SBW will be informed that a module is connected, but without a name or other information.
Connections made using this call are suitable when a module will not provide any services. If a module intends to provide services to other modules, the connection should be performed using the methods on the class `ModuleImpl` (see Section 4.5).
This is a variation on the plain `connect()` method described above. The same exception conditions apply.

static void disconnect()

Notifies SBW that this application will no longer be providing or consuming services. Invoking this method will first send a notification message to the Broker, then close the network socket and terminate the thread handling message processing.
A client can only be connected to one SBW Broker at a time; therefore, unlike the two versions of the `connect` method, there is no variant of `disconnect` that takes a host name or address as argument.

static void link(String hostNameOrAddress)
throw SBWBrokerStartException, SBWCommunicationException

Tells the local SBW Broker to connect to another SBW Broker running on a remote computer, starting the remote Broker first if necessary. The remote host is identified by the host name or IP address in `hostNameOrAddress`.
It should not normally be necessary to call this method when an application simply wants to start modules on remote hosts. The `SBW.getModuleInstance()` call described below will start remote Brokers and modules if the given module name has a specific form.

The following method provide ways of discovering services and service categories.

static String[] getServiceCategories(String parentCategory)
throws SBWCommunicationException, SBWIncorrectCategorySyntaxException

Returns an array of strings listing all of the immediate subcategories of the given category `parentCategory`. An empty string as the value of `parentCategory` stands for the root of the hierarchy tree; thus, `getServiceCategories("")` returns the top-level categories known to SBW.

static ServiceDescriptor[] findServices(String category, boolean recursive)
throws SBWCommunicationException, SBWIncorrectCategorySyntaxException

Returns an array of `ServiceDescriptor` objects corresponding to all of the services registered with SBW in the given `category`. The boolean flag `recursive` indicates whether the search should be performed recursively through the service hierarchy. If `true`, the string `category` is matched against all categories and subcategories. An example use of this method might be to use the category "Analysis" to query for all the services which provide analyses of SBML models.

The following methods are concerned with obtaining module instances from the SBW Broker.

static Module getModuleInstance(String moduleName)
throws SBWCommunicationException, SBWModuleStartException,
SBWModuleNotFoundException

Returns an instance of the given module. If the module instance has management type `ModuleImpl.UNIQUE` (see Section 4.2.4 on page 20), then a `Module` object corresponding to an existing module instance is returned. Otherwise, a new module instance is launched by the SBW Broker. A new instance is always launched if no existing instance exists.
The name `moduleName` may be either a module's unique name, or a name prefixed by a host name in the following form: `hostname:modulename`. If a host name is given, then this call causes the module to be started on the named remote host (if possible). If a Broker is not already running on the remote host, one is started first. Note that for this facility to work, the SSH client software must be installed on the client computer, the remote host must be running an SSH daemon, and you must have set up SSH for password-less authentication as described in Appendix C.
Various unusual conditions may lead to exceptions. These include: null module names, inability to contact the Broker, inability to start the requested module, inability to start a Broker on the remote host, etc.

static Module[] getExistingModuleInstances()
throws SBWCommunicationException

Returns an array of `Module` objects corresponding to every module connected to the system at this time.

static Module[] getExistingModuleInstances(String name)
throws SBWCommunicationException

Returns an array of `Module` objects corresponding to every running module that has the given module unique name. (Multiple instances of a module may be running if the module does not have management type `ModuleImpl.UNIQUE` and multiple client modules have requested instances.)

static Module getThisModule()

Returns a `Module` object representing the currently-running module.

static ModuleDescriptor getModuleDescriptor(String name)
throws SBWCommunicationException, SBWModuleNotFoundException

Returns a `ModuleDescriptor` object corresponding to the named module. If no such module is known to SBW, this method throws the exception `SBWModuleNotFoundException`.

static ModuleDescriptor[] getModuleDescriptors(boolean includeRunning)
throws SBWCommunicationException

Returns an array of `ModuleDescriptor` objects corresponding to all modules known to the SBW Broker. If the boolean flag `includeRunning` is `false`, only registered modules will be included (since those are the only kinds of modules for which the Broker will have registration information). If the flag is `true`, this method additionally includes running module instances. The difference in behavior is relevant when there are unregistered modules connected to SBW: when `includeRunning` is `false`, unregistered modules are not included in the array of module descriptors returned.

The following methods are used to specify optional listeners to SBW. Listeners are used in SBW to notify an application of certain events such as module startups and shutdowns. (See also the class `SBWListener` described in Section 4.8 on page 34.)

static void addListener(SBWListener listener)

Adds the given listener object to the list of listeners that SBW will call when certain events occur. (See Section 4.8 on page 34.)

static void removeListener(SBWListener listener)

Removes the given listener object from the list of listeners that SBW will call when certain events occur. (See Section 4.8 on page 34.)

The following methods are provided mainly for maintenance purposes.

static String getVersion()

Returns the version of the SBW Java API library. The version is a string of the form "major.minor.build", corresponding to the major, minor and build numbers, respectively.

static String getBrokerVersion()
throws SBWCommunicationException

Returns the version of the SBW broker. The version is a string of the form "major.minor.build", corresponding to the major, minor and build numbers, respectively.

The following method is only needed for using the SBW low-level API, described in Section 5.

static SBWLowLevel getLowLevelAPI()

Returns an object implementing the interface `SBWLowLevel` (see Section 5.8). Only clients wishing to use the low-level API should ever call this method.

4.7 Class: `SBWException`

All exceptions used in the SBW API are derived from `SBWException`. In Java, `SBWException` is derived from the standard Java `Exception` class.

String getMessage()

Returns the message string associated with this exception. This is a string that can be displayed to the user. (In Java, this method is inherited from Java's `Throwable` class).

String getDetailedMessage()

Returns additional details about the exception. This is typically at a level best understood by developers and may include a stack trace.

String getCode()

Returns the integer code assigned to the exception. This is not of much use in Java, but may be useful when writing low-level API code that interacts with non-Java modules.

The next two methods are useful for simple handling of SBW exceptions in user applications. They are typically used in code that looks like the following general structure:

```
try
{
  // Make some SBW calls here...
}
catch (SBWException e)
{
  e.handleWithDialog();
}
```

handleWithDialog()

Displays information about this exception to the user in a modal dialog box. The box has two buttons, **OK** and **Details...**. The user can click the **OK** button to dismiss the dialog, or click on **Details...** to see the detailed message (if any) associated with the exception.

handleWithDialog(java.awt.Component parentGUI)

Displays information about this exception to the user in a modal dialog box. The box has two buttons, **OK** and **Details...**. The user can click the **OK** button to dismiss the dialog, or click on **Details...** to see the detailed message (if any) associated with the exception.
This is identical to the previous method, but takes a single argument, `parentGUI`, to set the parent GUI component for the dialog.

Applications may need to construct and communicate their own exceptions, in situations that call for sending application-level exceptions to other modules. SBW provides the class `SBWApplicationException` for this purpose. It has the following constructor:

SBWApplicationException(String message, String detailedMessage)

Create an exception with the given message and detailed message.

A number of exceptions are derived from `SBWException`. Table 3 on the next page lists each exception and its meaning.

`Exception`	**Meaning**
`SBWApplicationException`	Application-specific exception, thrown deliberately by a module. This is the only type of exception that can be created by a module in SBW.
`SBWRawException`	Wrapper around any Java exceptions inadvertently thrown by a module.
`SBWCommunicationException`	Communications between a caller and receiver failed, possibly due to a lost connection.
`SBWModuleStartException`	An attempt to start a new module failed.
`SBWTypeMismatchException`	The type of data element that was attempted to be read from a message was not the type found. This often indicates a mismatch between the messages expected by a caller and receiver.
`SBWIncompatibleMethodSignatureException`	An interface or class definition uses method signatures that don't correspond to the signatures on the corresponding service.
`SBWModuleIdSyntaxException`	Indicates that a supplied module instance identifier has incorrect syntax.
`SBWIncorrectCategorySyntaxException`	A supplied category string has incorrect syntax. This is thrown by methods that find or search services by categories.
`Unused`	*This exception code is reserved for future use.*
`SBWServiceNotFoundException`	Requested service is not present on this module.
`SBWMethodTypeNotBlockTypeException`	The supplied class uses types which are not SBW message data block types.
`SBWMethodAmbiguousException`	Thrown by search methods such as `getMethod` on the `Service` class to indicate that the given signature or signature component matches with more than one method on the given service.
`SBWUnsupportedObjectTypeException`	The library encountered an object of a type that it cannot encode or decode from a message block.
`SBWMethodNotFound`	The given method identification number, signature or partial signature does not match any of the methods that exist on a given service. This can occur if a caller uses the low-level SBW API to attempt to invoke a method on a service and the service has no method with that index.
`SBWSignatureSyntaxError`	Thrown if a signature string does not contain a valid SBW signature. In the Java API, this is only thrown by method `getMethod` on class `Service`.
`SBWModuleDefinitionException`	Thrown by the `ModuleImpl` constructors if any aspect of a module definition is incorrect.
`SBWModuleNotFoundException`	Thrown if the given module identification name doesn't match any module known to SBW.
`SBWBrokerStartException`	Thrown by methods that may cause a local or remote Broker to be started; it indicates that SBW was unable to start the required Broker

Table 3: *Exceptions derived from SBWException and their encoding.*

4.8 Class: `SBWListener`

Listeners are used in SBW to notify an application that certain events have occurred. The class `SBWListener` is an abstract adapter class for receiving notifications of these events. The methods in this class are empty. This class exists as a convenience for creating `SBWListener` objects.

You can extend this class to create an SBW Listener and override the methods for the particular events of interest. Since this class defines null methods, you only have to define methods for the events that you care about.

A listener may be registered or deregistered with SBW using the methods `SBW.addListener` and `SBW.removeListener` on the `SBW` class (Section 4.6 on page 28).

void onModuleShutdown(Module module)

This method on the listener is called every time a module instance somewhere in the SBW system disconnects from the SBW Broker. The `module` passed to the method represents the module instance that has just shut down.

void onModuleStart(Module module)

This method on the listener is called every time a module instance starts up or connects to the SBW Broker. The `module` passed to the method represents the module instance that has just shut down.

void onRegistrationChange()

This method on the listener is called whenever a registration change for a module occurs in the SBW Broker. “Registration changes” are: a module registering itself with the Broker, a module registering a service with the Broker, or a module being unregistered.

4.9 Class: `Service`

The `Service` class represents a *service*, an interface to resources inside a module. Obtaining an instance of a `Service` object implies that a module implementing the service has been launched by the SBW Broker.

ServiceDescriptor getDescriptor()
throws SBWCommunicationException, SBWModuleNotFoundException,
SBWServiceNotFoundException, SBWException

Returns a `ServiceDescriptor` object (see Section 4.9) corresponding to this service. The service descriptor holds information about this service's unique name, display name, and other attributes.

Module getModule()

Returns a `Module` class object corresponding to the module that implements this service. Since the module instance must already be running, this action does not result in a new instance being launched; however, it does create a new object of class `Module` referring to the same instance.

The most powerful methods on class `Service` are the following two, which return objects that serve as proxies for the service. Most applications access remote modules by first obtaining a proxy object using one of these methods.

Object getServiceObject(Class interfaceClass)
throws SBWCommunicationException, SBWServiceNotFoundException

Returns a proxy object for this service. The proxy object will have the interface specified by argument `interfaceClass`. The methods of this object will be the methods specified by the interface.
The `interfaceClass` argument is necessary because of how the Java compiler operates.

Object getServiceObject(Class[] interfaceClasses)
throws SBWCommunicationException, SBWServiceNotFoundException

Returns a proxy object for this service. The proxy object will have the interfaces specified by array of classes in `interfaceClasses`. The methods of this object will be the methods specified by the union of the interfaces.

The following two methods return information about the methods defined on this service. The information is represented using `ServiceMethod` class objects (see Section 4.11 on page 38), which only serve to describe service methods at the level of name, signature string, and help string.

ServiceMethod[] getMethods()
throws SBWCommunicationException

Returns an array of `ServiceMethod` objects, one for each of the methods on this service. Since obtaining the method list requires contacting the module instance, this method may throw an `SBWCommunicationException` if something goes wrong during communications.

ServiceMethod getMethod(String nameAndArgs)
throws SBWCommunicationException, SBWMethodAmbiguousException,
SBWSignatureSyntaxException

Returns a `ServiceMethod` object corresponding to the single method described by the partial signature `nameAndArgs`. The parameter may consist of only a method name, or a name and arguments written in the form discussed in Section 4.2.2 on page 18. Specifically, the allowable syntax is:

```
SName
```

or

```
[ReturnType Space] SName Space? '(' VarArgList ')'
```

where the optional arguments list is used to locate specific methods with the same name. The return type is ignored when matching signatures. If the syntax is incorrect, this method will throw `SBWSignatureSyntaxException`.
The given name (and optional arguments) is compared to the parsed signatures of the methods on the service. If more than one method on the service matches the given argument, this method throws `SBWMethodAmbiguousException`.
Since obtaining the method list requires contacting the module instance, this method may throw an `SBWCommunicationException` if something goes wrong during communications.

4.10 Class: `ServiceDescriptor`

A *service descriptor* is used to describe a single service provided by a resource available through SBW. The information is obtained from the SBW Broker.

A `ServiceDescriptor` object is similar to a `ModuleDescriptor` (Section 4.4 on page 23) in that it provides basic information about a service, but does not necessarily imply that a module instance implementing the service has been started up. This decoupling of services from service descriptors allows service descriptor objects to be used as handles to potential services. For example, if a list of services in a given category is to be displayed to the user before those services are invoked, a representation is required of those services outside the context of a module instance.

public Service getServiceInModuleInstance()
throws SBWCommunicationException, SBWModuleStartException

Returns a `Service` object (Section 4.9 on page 35) corresponding to this service. Since `Service` objects are tied to running module instances, invoking this method will either create a new module instance or reuse an existing instance (depending on the module's management type and whether an instance already exists).

String getName()

Returns the unique identification of the service. This name must follow the conventions outlined in Section 4.2.3. In particular, the name must use the `SName` syntax as defined in Figure 11 on page 19, and should start with a capital letter. An example of a service name is “Simulation”.

String getDisplayName()

Returns the display name of the service. This is suitable for showing to users in a menu or list. Although there is no fixed limit on the length of display names, they should be kept short.

String getCategory()

Returns the category of this service. Service categories can consist of any sequence of non-control characters excluding \n, \ and / but including space. Service category strings as used in the API are sequences of one or more service category level names separated by either \ or / characters. An example service category is “Analysis/Bifurcation”.

String getHelp()

Returns a help string for the service. This is an optional description of the service provided for informational purposes only.

ModuleDescriptor getModuleDescriptor()

Returns a module descriptor object for the module that implements this service.

4.11 Class: `ServiceMethod`

This class is used to describe a method on a service. (In the Java SBW API, its role is limited to being informational only, although in some of the other language bindings the `ServiceMethod` class has additional functionality.)

String getName()

Returns the name of this method.

String getSignatureString()

Returns the signature of this method as a string. Contrast this to the next method, `getSignature()`, which returns a tree-structure representation of the method signature.

Signature getSignature()

Returns a `Signature` object (Section 4.12 on the following page) representing the signature for this service method. The `Signature` object provides access to a parsed representation of the individual components of a method signature.

String getHelp()

Returns the documentation string for this service method.

4.12 Class: `Signature`

The `Signature` class is used to represent signatures in a parsed, normalized format. This allows more precise matching of method signatures than using simple string forms of signatures.

String getName()

Returns the name of this method.

SignatureElement[] getArguments()

Returns an array of `SignatureElement` class objects, one for each of the arguments for this method. The order of the elements in the array is the order of the arguments in the method signature.

SignatureElement getReturnType()

Returns a single `SignatureElement` class object representing the return type of this method.

String toString()

Converts this method signature to string form.

4.13 Class: `SignatureElement`

A *signature element* is a name-type pair in a method signature, with the name portion being optional. This provides a single common structure for representing both method arguments (where an argument may be either a data type or a data type followed by a name) and list content types. For example, the method signature `"void foo(string a, int b)"` contains two signature elements for its arguments, `string a` and `int b`.

String getName()

Returns the name portion of this signature element. This may be null.

SignatureType getType()

Returns a `SignatureType` class object (Section 4.14 on the next page) representing the data type of this signature element.

boolean equals(SignatureElement se)

The standard `equals` method of the Java `Object` class is overloaded by `SignatureElement` to permit comparison between two `SignatureElement` objects. The equality is determined on the basis of the signature data types only, *not* the name portion.

4.14 Class: `SignatureType`

A `SignatureType` class object represents a data type in a signature. The allowable types are those of the element `Type` in the syntax definition for method signatures in Figure 11 on page 19.

SignatureType getArrayInnerType()

If this `SignatureType` object is an array, this method returns the data type of the elements in the array.

SignatureElement[] getListContents()

If this `SignatureType` object is a list, this method returns an array of `SignatureElement` class objects corresponding to the types of each element in the list.

boolean equals(SignatureType other)

The standard `equals` method of the Java `Object` class is overloaded by `SignatureType` to permit comparison between two `SignatureType` objects.

String toString()

Returns a string corresponding to the data type. The possible types are `byte`, `int`, `double`, `boolean`, `string`, `void`, `[]` to indicate an array, and to indicate a list.

5 SBW Java Low-Level API Reference

In addition to using the high-level API described so far, an application can be interfaced to SBW using a more basic, lower-level API. The *SBW low-level API* involves building up messages directly and using call/send methods to communicate with modules and methods directly, instead of using the object-oriented interface provide by the SBW Java API library. It allows a module direct and complete control over its interactions with SBW, at the expense of requiring more work on the part of the programmer. Both the low-level API and the object-oriented API are available in the same SBW Java library. Their use in a module can even be mixed if desired.

There are three main elements in the SBW low-level API:

- Message buffers and methods for reading from and writing to them. In SBW, this is encapsulated by the classes `DataBlockReader` and `DataBlockWriter`.
- A receiver object that handles incoming requests from other modules. This is implemented through the `Receiver` class.
- An interface for obtaining service and method identifiers and calling the methods, as well as for performing certain basic operations. This is implemented through the `SBWLowLevel` interface class.

The classes associated with the elements above are described in the sections below. In addition to these classes, there are certain predefined protocols and messages that all SBW modules obey, and a set of messages that the SBW Broker understands. The protocols are defined below in Sections 5.3 and 5.4.

5.1 Using the Low-Level API

A client program using the low-level API typically follows a certain pattern:

1. The client first obtains an object implementing `SBWLowLevel`. This is done using the `SBW.getLowlLevelAPI` call (Section 4.6), using code such as the following:

```
try
{
  rpc = SBW.getLowLevelAPI();
}
catch (Exception e)
{
  e.handleWithDialog();
}
```

2. The client next connects to SBW using one of the variants of the `connect` method on `SBWLowLevel`.
3. Once connected to SBW, the client can obtain information about the modules known to the Broker. It can do this by sending specific messages to the SBW Broker using the `call` method on `SBWLowLevel`. The messages are described in Section 5.4. This allows a module to obtain the numeric identifiers corresponding to specific instances of modules connected to the Broker.
4. Given one or more module identifiers, the client can then use the `SBWLowLevel` methods `getServiceId` and `getMethodId` to determine the numeric service and method identifiers for desired services and methods.

5. The client can then use the `SBWLowLevel` methods `call` and/or `send` to invoke specific service methods on specific modules.

If a module is to provide services for use by other modules, it must in addition register a `Receiver` object (see Section 5.7). This object must handle a number of SBW system messages, in addition to dispatching any of its own methods for implementing the services it provides.

5.2 Identifiers

Modules, services and service methods are referred to, in the low-level API calling methods, using integer numbers as identifiers. In particular, the low-level `call` and `send` methods on `SBWLowLevel` (see Section 5.8) use only these numerical identifiers, and not names, to address which specific method on a module is to be invoked. To determine the identifiers for services and methods on a remote module, a client module must use the methods `getServiceId` and `getMethodId` on class `SBWLowLevel`.

The SBW Broker is preassigned an identifier of -1, and every module must implement a service called *SYSTEM* whose identifier is -1. There are a number of predefined methods on the SYSTEM service, and these have preassigned identifiers as well. They are listed in Table 5 on page 54.

Service identifiers, like service names, are local to modules, and method identifiers are local to services on specific modules. There may be a module whose identifier is 0, as well as a service 0, and a service method 0 on that service, and they all mean different things. Similarly, there may be two different modules that both have services with identifiers 0, but the two services can be completely unrelated.

Except for the preassigned identifiers, a module must *not* assume that the identifier for a module, a service, or a service method is the same from one SBW session to another. Module identifiers are assigned by the SBW Broker in an unspecified order to modules as they connect to the Broker. These identifiers are not persistent and will change when the Broker is restarted. Moreover, a module's internal implementation may change from session to session, and it is permissible for a module to reorder service identifiers and the method identifiers of methods on those services. (The exception, of course, is that the SYSTEM service and the methods on that service must be given the values defined in Table 5 on page 54.) The implication of this is that client applications should not be written with hardwired module, service or service method identifiers. At the beginning of its interactions with a remote module in a given session, a client should always obtain the service identifiers and method identifiers using `SBWLowLevel`'s `getServiceId` and `getMethodId`. It can then use those identifiers for the remainder of its interactions with the remote module during that session.

5.3 Methods Required to Be Implemented by All Modules

As mentioned above, all modules must implement at least one service, called the *SYSTEM* service. This service has the following predefined methods (where the method signatures are given in the SBW signature string format defined in Figure 11 on page 19):

`string[] getServices()`
: This method returns the unique names of all services implemented by this module. The index number of each service in the returned array must correspond to the identification number for that service.

`string[] getMethods(int serviceId)`
: This method returns an array of method signatures, one for each method defined on the service indexed by service identifier `serviceId`. The index number of each method in the returned array must correspond to the identification number for that method on the service.

`string getMethodHelp(int serviceId, int methodId)`

This method returns the help string associated with the method identified by `methodId` on the service identified by `serviceId`.

These methods must be implemented as part of the `Receiver` object described in Section 5.7.

5.4 Methods Implemented by the SBW Broker

5.5 Class: `DataBlockWriter`

The message-passing system implemented in SBW is a relatively simple scheme that encodes data values as bytes and sends them in a stream across a socket connection. The message encoding has a custom format in which each data element in a message is prefixed by a byte that describes its data type. The supported data types have already been described in Section 3.1. The SBW *message block* is the programming abstraction used to interact with messages. The class `DataBlockWriter` implements methods for writing message blocks.

DataBlockWriter()

Constructor; creates an empty message block. Contents can be added to the message block using the methods listed below.

void release()

Indicates to SBW that this data block will no longer be used and may be reclaimed. Client applications should call this method after having used the contents of a message in an `SBWLowLevel.call` or `SBWLowLevel.send` operation.

void add(Object o)
throws SBWUnsupportedObjectTypeException

Appends the given Java object to the current message block. The object is converted to the appropriate SBW primitive type automatically. The object in parameter `o` must be of one of the following Java data types, or 1-D or 2-D arrays of these types:

- `java.lang.String`
- `java.lang.Integer`
- `java.lang.Double`
- `java.lang.Byte`
- `java.lang.Boolean`
- `java.lang.List`
- `edu.caltech.sbw`

In the case of `java.lang.List`, each of the items in the list must be one of the data types listed above, or 1-D or 2-D arrays of these types, potentially including more lists. If the object in parameter `o` is not one of the supported SBW data types (Section 3.1), this method throws an exception of class `SBWUnsupportedObjectTypeException`.

void add(boolean b)

Appends the given boolean data item to the current message block.

void add(Collection c)
throws SBWUnsupportedObjectTypeException

Adds a Java collection as a list of items into the current message block. Each object in the collection is converted to the appropriate SBW primitive data type automatically. The reader of the message will see this element as a single list.
The objects in parameter `c` must be of one of the following Java data types, or 1-D or 2-D arrays of these types:

- `java.lang.String`
- `java.lang.Integer`
- `java.lang.Double`
- `java.lang.Byte`
- `java.lang.Boolean`
- `java.lang.List`
- `edu.caltech.sbw`

In the case of `java.lang.List`, each of the items in the list must be one of the data types listed above, or 1-D or 2-D arrays of these types, potentially including more lists. If the object in parameter `c` is not one of the supported SBW data types (Section 3.1), this method throws an exception of class `SBWUnsupportedObjectTypeException`.

void add(byte b)

Appends the given byte to the current message block. The data item becomes a separate element in the message; this is in contrast to writing an array of bytes, where a sequence of bytes all become part of a single data element in the message.

void add(double d)

Appends the given `double`-type data item to the current message block.

void add(int i)

Appends the given integer to the current message block.

void add(String s)

Appends the given Java string item to the current message block.

void add(int[] a)

Appends the given one-dimensional array of integers to the current message block.

void add(int[][] a)

Appends the given two-dimensional array of integers to the current message block.

void add(byte[] a)

Appends the given one-dimensional array of bytes to the current message block.

void add(byte[][] a)

Appends the given two-dimensional array of bytes to the current message block.

void add(boolean[] a)

Appends the given one-dimensional array of booleans to the current message block.

void add(boolean[][] a)

Appends the given two-dimensional array of booleans to the current message block.

void add(String[] a)

Appends the given one-dimensional array of strings to the current message block.

void add(String[][] a)

Appends the given two-dimensional array of strings to the current message block.

void add(double[] a)

Appends the given one-dimensional array of doubles to the current message block.

void add(double[][] a)

Appends the given two-dimensional array of doubles to the current message block.

void add(SBWComplex a)

Adds the variable of complex datatype to the current message block.

void add(SBWComplex[] a)

Appends the given one-dimensional array of complex types to the current message block.

void add(SBWComplex[][] a)

Appends the given two-dimensional array of complex types to the current message block.

void add(Collection[] a)
throws SBWUnsupportedObjectTypeException

Appends the given one-dimensional array of Java collections to the current message block.
Each of the objects in each collection of the array `a` must be of one of the following Java data types, or 1-D or 2-D arrays of these types:

- `java.lang.String`
- `java.lang.Integer`
- `java.lang.Double`
- `java.lang.Byte`
- `java.lang.Boolean`
- `java.lang.List`
- `edu.caltech.sbw`

In the case of `java.lang.List`, each of the items in the list must be one of the data types listed above, or 1-D or 2-D arrays of these types, potentially including more lists.
If the object in parameter `c` is not one of the supported SBW data types (Section 3.1), this method throws an exception of class `SBWUnsupportedObjectTypeException`.

void add(Collection[][] a)
throws SBWUnsupportedObjectTypeException

Appends the given two-dimensional array of Java collections to the current message block.
Each of the objects in each collection of the array `a` must be of one of the following Java data types, or 1-D or 2-D arrays of these types:

- `java.lang.String`
- `java.lang.Integer`
- `java.lang.Double`
- `java.lang.Byte`
- `java.lang.Boolean`
- `java.lang.List`
- `edu.caltech.sbw`

In the case of `java.lang.List`, each of the items in the list must be one of the data types listed above, or 1-D or 2-D arrays of these types, potentially including more lists.
If the object in parameter `c` is not one of the supported SBW data types (Section 3.1), this method throws an exception of class `SBWUnsupportedObjectTypeException`.

Several static constants are also defined by `DataBlockWriter` for identifying message data types. These are listed in Table 4. At the level of client code, only the method `getNextType` on `DataBlockReader` (see Section 5.6) can return one of these type codes. They are not exposed to client code in the writing operations defined below, because all of the writing operations encapsulate the task of prepending the appropriate type specifier when a data element is written to a message block.

Type Code	**Meaning**
`DataBlockWriter.BYTE_TYPE`	Data element is a byte
`DataBlockWriter.INTEGER_TYPE`	Data element is an integer
`DataBlockWriter.DOUBLE_TYPE`	Data element is a double
`DataBlockWriter.BOOLEAN_TYPE`	Data element is a boolean
`DataBlockWriter.STRING_TYPE`	Data element is a string
`DataBlockWriter.ARRAY_TYPE`	Data element is an array
`DataBlockWriter.LIST_TYPE`	Data element is a list
`DataBlockWriter.TERMINATE_TYPE`	Message terminator
`DataBlockWriter.COMPLEX_TYPE`	Data is an SBWComplex

Table 4: *Type codes defined by SBW for identifying data element types in messages. Each code has the Java data type `byte`. These are the possible values that the message block reader class, `DataBlockReader`, can return from the `getNextType()` method call.*

5.6 Class: `DataBlockReader`

The class `DataBlockReader` implements methods for reading message block data. SBW returns an object of this class from calls that involve receiving messages, and client applications can use the methods on this class to extract the data elements out of the message.

Note that client applications cannot actually create new `DataBlockReader` objects themselves; only SBW does that. The `SBWLowLevel` class (Section 5.8) and the `Receiver` class (Section 5.7) are the two low-level API classes that make use of `DataBlockReader`.

Object getObject()
throws SBWTypeMismatchException

Extracts and returns the next element in the message block and returns it as an object. Primitive types are wrapped appropriately, so that for example, an integer is returned as a Java `Integer` object.
If an unknown data item is encountered, this method throws an `SBWTypeMismatchException`.

byte getNextType()

Returns a byte representing the data type of the next data item in the message block. Unlike the other `get` methods described below, `getNextType()` does not consume the data item in the message. The possible byte codes are listed in Table 4 on the page before.

int getInt()
throws SBWTypeMismatchException

Extracts and returns the next element in the message block, assuming that it is an integer. If the next element is not an integer, this method throws an `SBWTypeMismatchException`.

boolean getBoolean()
throws SBWTypeMismatchException

Extracts and returns the next element in the message block, assuming that it is a boolean. If the next element is not a boolean, this method throws an `SBWTypeMismatchException`.

byte getByte()
throws SBWTypeMismatchException

Extracts and returns the next element in the message block, assuming that it is a byte. If the next element is not a byte, this method throws an `SBWTypeMismatchException`.

String getString()
throws SBWTypeMismatchException

Extracts and returns the next element in the message block, assuming that it is a string. If the next element is not a string, this method throws an `SBWTypeMismatchException`.

double getDouble()
throws SBWTypeMismatchException

Extracts and returns the next element in the message block, assuming that it is a `double`-type floating-point number. If the next element is not a double, this method throws an `SBWTypeMismatchException`.

SBWComplex getComplex()
throws SBWTypeMismatchException

Extracts and returns the next element in the message block, assuming that it is a complex type. If the next element is not a complex type, this method throws an `SBWTypeMismatchException`.

Collection getIntoCollection()
throws SBWTypeMismatchException

Extracts the entire contents of the rest of this message and returns it as a Java collection. This works by calling `getObject()` repeatedly until it reaches the end of the message.

List getList()
throws SBWTypeMismatchException

Extracts and returns the next element in the message block, assuming that it is a list. If the next element is not a list, this method throws an `SBWTypeMismatchException`.

int[] get1DIntArray()
throws SBWTypeMismatchException

Extracts and returns the next element in the message block, assuming that it is a one-dimentionsal array of `int`. If the next element is not a 1-D array of integers, this method throws an `SBWTypeMismatchException`.

int[][] get2DIntArray()
throws SBWTypeMismatchException

Extracts and returns the next element in the message block, assuming that it is a two-dimentionsal array of `int`'s. If the next element is not a 2-D array of integers, this method throws an `SBWTypeMismatchException`.

boolean[] get1DBooleanArray()
throws SBWTypeMismatchException

Extracts and returns the next element in the message block, assuming that it is a one-dimentionsal array of booleans. If the next element is not a 1-D array of booleans, this method throws an `SBWTypeMismatchException`.

boolean[][] get2DBooleanArray()
throws SBWTypeMismatchException

Extracts and returns the next element in the message block, assuming that it is a two-dimentionsal array of booleans. If the next element is not a 2-D array of booleans, this method throws an `SBWTypeMismatchException`.

byte[] get1DByteArray()
throws SBWTypeMismatchException

Extracts and returns the next element in the message block, assuming that it is a one-dimentionsal array of bytes. If the next element is not a 1-D array of bytes, this method throws an `SBWTypeMismatchException`.

byte[][] get2DByteArray()
throws SBWTypeMismatchException

Extracts and returns the next element in the message block, assuming that it is a two-dimentionsal array of bytes. If the next element is not a 2-D array of bytes, this method throws an `SBWTypeMismatchException`.

String[] get1DStringArray()
throws SBWTypeMismatchException

Extracts and returns the next element in the message block, assuming that it is a one-dimentionsal array of strings. If the next element is not a 1-D array of strings, this method throws an `SBWTypeMismatchException`.

String[][] get2DStringArray()
throws SBWTypeMismatchException

Extracts and returns the next element in the message block, assuming that it is a two-dimentionsal array of strings. If the next element is not a 2-D array of strings, this method throws an `SBWTypeMismatchException`.

double[] get1DDoubleArray()
throws SBWTypeMismatchException

Extracts and returns the next element in the message block, assuming that it is a one-dimentionsal array of `double`-type items. If the next element is not a 1-D array of doubles, this method throws an `SBWTypeMismatchException`.

double[][] get2DDoubleArray()
throws SBWTypeMismatchException

Extracts and returns the next element in the message block, assuming that it is a two-dimentionsal array of doubles. If the next element is not a 2-D array of doubles, this method throws an `SBWTypeMismatchException`.

List[] get1DListArray()
throws SBWTypeMismatchException

Extracts and returns the next element in the message block, assuming that it is a one-dimentionsal array of lists. If the next element is not a 1-D array of lists, this method throws an `SBWTypeMismatchException`.

List[][] get2DListArray()
throws SBWTypeMismatchException

Extracts and returns the next element in the message block, assuming that it is a two-dimentionsal array of lists. If the next element is not a 2-D array of lists, this method throws an `SBWTypeMismatchException`.

5.7 Interface: `Receiver`

This interface class is used to define a receiver for messages sent to a module. The interface has only one method, `receive`, as defined below. If a module wishes to use the SBW low-level API, it must create an object that implements the `Receiver` interface, and then register that object using the `registerReceiver` method on class `SBWLowLevel` (see Section 5.8).

DataBlockWriter receive(int fromModule, int serviceId, int methodId, DataBlockReader args) throws SBWException

Whenever a message arrives for this module, this method is called with arguments to indicate the source module (`moduleId`), the service being invoked (`serviceId`), and the method on that service (`methodId`), along with a message block containing any arguments for that method.

Figure 12 gives an example implementation of a `receive` method. It begins with a switch on the service identifier to determine whether the incoming message is invoking a SYSTEM

```
public DataBlockWriter receive(int fromModuleId, int serviceId,
                               int methodId, DataBlockReader args)
    throws SBWException
{
  if (serviceId == SBWLowLevel.SYSTEM_SERVICE)
  {
    DataBlockWriter writer = new DataBlockWriter();
    try
    {
      switch (methodId)
      {
      case SBWLowLevel.GET_SERVICES_METHOD:
        writer.add(// ...  list of this module's services ...);
        return writer;

      case SBWLowLevel.GET_METHODS_METHOD:
        int queryServiceId = args.getInt();
        writer.add(// ...  signatures of methods on the service ...);
        return writer;

      case SBWLowLevel.GET_METHOD_HELP_METHOD:
        int targetServiceId = args.getInt();
        int targetMethodId = args.getInt();
        writer.add(// ...  help string for the method ...);
        return writer;

      default:
        throw new SBWApplicationException(
          "Method " + methodId + " does not exist on service " + serviceId, "");
      }
    }
    catch (Throwable t)
    {
      // Handle the error here ...);
    }
  }
  else
  {
    return getService(serviceId).invokeMethod(fromModuleId, methodId, args);
  }
}
```

Figure 12: *Example implementation of a `Receiver` `receive` method for a module.*

service or a service on the module proper. In the former case, it then switches between a set of predefined method identifiers. In the latter case, it hands control to another method that handles service invocations for this module.

5.8 Interface: `SBWLowLevel`

This interface class defines generic operations in the SBW low-level API. As discussed in Section 5.1, a client program must obtain an object implementing the `SBWLowLevel` interface by calling `SBW.getLowLevelAPI`. From that point on, it can invoke the `SBWLowLevel` methods in order to connect to SBW, obtain information about modules and services, and use the `call`/`send` methods to invoke methods on remote services.

Certain static constants are defined by `SBWLowLevel` to identify the fundamental services and methods necessary for SBW to function. Table 5 lists these constants and their meanings.

Constant	Value	Meaning
`SBWLowLevel.BROKER_MODULE`	−1	Module identifier for the SBW Broker
`SBWLowLevel.SYSTEM_SERVICE`	−1	Service identifier for the "SYSTEM" service
`SBWLowLevel.GET_SERVICES_METHOD`	0	Identifier for method to get a list of services offered by a module
`SBWLowLevel.GET_METHODS_METHOD`	1	Identifier for method to get a list of methods available on a given service
`SBWLowLevel.GET_METHOD_HELP_METHOD`	4	Identifier for method used to obtain the help string of a specific method

Table 5: *Predefined constants in the SBW low-level API.*

void connect(String moduleName)
throws SBWCommunicationException

Connect to the SBW Broker running on the current host computer. The argument `moduleName` should be a unique module name that identifies this module to the rest of SBW. If the argument is empty or null, it signifies that this module will not provide services to SBW, just as if it connected using `SBW.connect()`. (In effect, it will act as an anonymous module.)

void connect(String moduleName, String hostNameOrIP)
throws SBWCommunicationException

Connect to the SBW Broker running on the host identified by the name or IP address in parameter `hostNameOrIP`. The argument `moduleName` should be a unique module name that identifies this module to the rest of SBW. If the `moduleName` is empty or null, it signifies that this module will not provide services to SBW, just as if it connected using `SBW.connect()`. (In effect, it will act as an anonymous module.)
A client can only be connected to one SBW Broker at a time. An attempt to connect to SBW when a client is already connected results in an exception of class `SBWCommunicationException`.

boolean isConnected()

Returns `true` if this application is currently connected to SBW.

void disconnect()

Disconnects from from SBW. This causes disconnection messages to be sent to the SBW Broker and the communications sockets to be closed.

int getServiceId(int moduleId, String serviceName)
throws SBWException

Given a module's numeric identifier and the name of a service, this method returns the numeric service identifier. This is achieved by contacting the module directly. The SBW Broker purposefully does not cache this information to avoid the risk of having the information fall out of date.
Applications should not store numeric service identifiers beyond the duration of a single session with SBW. A module may be updated or otherwise changed in-between sessions, and the correspondence between services and their identifiers may be reassigned by the module. A client should always obtain the service identifiers afresh in each new session.

int getMethodId(int moduleId, int serviceId, String signature)
throws SBWException

Given a module's numeric identifier, a service identifier, and a method's signature string (cf. Section 4.2.2), `getMethodId` returns the numeric method identifier. This is achieved by contacting the module directly. The SBW Broker purposefully does not cache this information to avoid the risk of having the information fall out of date.
The same warnings apply as for service identifiers: applications should not store numeric method identifiers beyond the duration of a single session with SBW. A module may be updated or otherwise changed in-between sessions, and the correspondence between methods and their identifiers may be reassigned by the module. A client should always obtain the method identifiers anew in each new session.

int getThisModule()
throws SBWException

This method returns the numeric identifier for the currently-running module. (See also `SBW.getThisModule()` in sec:ref:sbw-class.)

The following two methods, `call` and `send`, are the two communications method available in the SBW low-level API. The `call` method is analogous to a traditional RPC (remote-procedure call) mechanism. The `send` method is more in the spirit of a message-oriented invocation.

The following is an example of a call on the SBW Broker, using the SYSTEM service and the method GET_METHODS_METHOD. The argument to the call is the integer 0, written into a `DataBlockWriter` object. The effect of this call is to ask the Broker to provide a list of the methods on its 0'th service (which happens to be the Broker service—see the discussion of the Broker messages in Section 5.4). The return value from the call is an array of strings.

```
DataBlockWriter data;
DataBlockReader results;

data = new DataBlockWriter();
try
{
  data.add(0);
  results = rpc.call(SBWLowLevel.BROKER_MODULE,
                     SBWLowLevel.SYSTEM_SERVICE,
                     SBWLowLevel.GET_METHODS_METHOD,
                     data);
  //The results are supposed to be an array of strings.
  String[] names = results.get1DStringArray();
  ...
}
catch (Exception e)
{
  System.err.println("Unexpected problem:");
  e.printStackTrace();
```

```
        System.exit(1);
    }
```

Note the use of the predefined constants for identifying the module, service and method to be invoked; see Table 5 on page 54 for a listing.

DataBlockReader call(int moduleTo, int serviceId, int methodId, DataBlockWriter args) throws SBWException

This implements the fundamental, synchronous (blocking) remote method call. Given a module's identifier (in `moduleTo`), a service identifier (in `serviceId`), and a method identifier (in `methodId`), this invokes the indicated service method on the indicated module with arguments `args`. It waits for a return value and returns it as a `DataBlockReader`. See Section 5.2 for a discussion of identifiers and Section 5.6 for a definition of class `DataBlockWriter`.
Any of the exceptions listed in Table 3 may be thrown while attempting to make this call.

void send(int moduleTo, int serviceId, int methodId, DataBlockWriter args) throws SBWException

This implements the fundamental, asynchronous (non-blocking) message sending operation. Given a module's identifier (in `moduleTo`), a service identifier (in `serviceId`), and a method identifier (in `methodId`), this invokes the indicated service method on the indicated module with arguments `args`. Unlike the previous method (`call`), this does not wait for a return value; instead, it returns immediately.
Any of the exceptions listed in Table 3 may be thrown while attempting to make this call.

void registerReceiver(Receiver rec)

Installs the given `Receiver`-class object as the receiver for this module. This object will be called to handle the bodies of incoming messages. The receiver object must implement the functionality described in Section 5.7.

A Complete Examples of Using the APIs

This section presents complete examples of programs using the SBW Java interface. In the first example below, we provide full details of how to compile and run the code; subsequent examples omit the additional details. The instructions for compiling and running the code largely repeat information already presented in Section 4.1, but we do so anyway for clarity and the reader's convenience.

A.1 Example of Implementing a Module

This example presents the implementation of a module named "edu.caltech.trigj" providing a service named "Trig". This is essentially the same example as used in Section 3.4; the code presented here is also included with the SBW source distribution in the subdirectory `src/tutorials/Java/TrigJServerModule`.

The "Trig" service in this example is placed in a top-level service category named "trigonometry". For convenience, the code is split into two files, one for the service implementation (`Trig.java`, shown in Figure 13) and one for the main routine (`TrigApplication.java`, shown in Figure 14 on the next page). Note that in Java, it is important to place these files in a subdirectory structure that matches the package name; in other words, the two files must be placed in a directory called `edu/caltech/trigj`.

```java
package edu.caltech.trigj;
import java.lang.Math;

public class Trig
{
    /** This becomes the "sin" method of service "Trig". **/
    public double sin(double x)
    {
        return Math.sin(x);
    }

    /** This becomes the "cos" method of service "Trig". **/
    public double cos(double x)
    {
        return Math.cos(x);
    }
}
```

Figure 13: *Contents of the file `Trig.java`.*

To run these examples, follow the instructions given in Section 4.1. Here we only provide a summary of those steps. First, place the two Java source files in a directory tree having the path `edu/caltech/trigj`. Then, at the top of this directory tree, execute the following command to compile the Java code:

```
javac -classpath SBW_HOME/lib/SBWCore.jar:. edu/caltech/trigj/*.java
```

where, as usual, *SBW_HOME* stands for the path to the SBW directory on your computer.

Once you have compiled the Java source files, you can run the module in "registration" mode by executing the following command (all on one line):

```
java -classpath SBW_HOME/lib/SBWCore.jar:. \
    edu.caltech.trigj.TrigApplication -sbwregister
```

The module will run briefly and then exit, having registered itself with the SBW Broker. To run the module in "module" mode, where it offers its service to other modules, use the `-sbwmodule`

```
package edu.caltech.trigj;

import edu.caltech.sbw.*;            // Import SBW.

public class TrigApplication
{
    public static void main(String[] args)
    {
        // Calls to ModuleImpl must be wrapped with a try-catch, in order
        // to catch possible exceptions from SBW.

        try
        {
            // First, we create the module definition.  This uses the
            // simplest form of the ModuleImpl constructor.  The various
            // parts of the module's definition, such as its unique name,
            // are defaulted.  There are other versions of the ModuleImpl
            // constructor that accept more arguments.

            ModuleImpl moduleImpl = new ModuleImpl("Java Trig Module");

            // Next, we add the service definitions.

            moduleImpl.addService("Trig", "Trigonometric functions",
                                  "trigonometry", Trig.class);

            // Finally, we take action based on the command-line arguments
            // passed to us.  If one of the arguments is -sbwregister or
            // -sbwmodule, run() will take appropriate action; otherwise it
            // will do nothing and just return.  There is nothing else to
            // do here, so we don't follow up the call to run() with
            // anything else in this particular program.

            moduleImpl.run(args);
        }
        catch (SBWException e)
        {
            // SBW reported an exception.  Use the methods on SBWException
            // to show the exception to the user.

            e.handleWithDialog();
        }
    }
}
```

Figure 14: *Contents of the file* `TrigApplication.java`.

flag as follows (again, all on one line):

```
java -classpath SBW_HOME/lib/SBWCore.jar:. \
    edu.caltech.trigj.TrigApplication -sbwmodule
```

You could also create a self-contained JAR file for this module by using the `sbw-make-jar` program discussed in Appendix B.

A.2 Example of Calling a Service

This example presents a client that accesses the trigonometry service implemented in the previous example. This is the complete Java code for the example discussed in Section 3.3. The code is also included with the SBW source distribution in the subdirectory `src/tutorials/Java/TrigJClientModule`.

There are two files in this example. The first (`Trigonometry.java`, shown in Figure 15 on the following page) defines the interface of the service. A second file that contains the actual client code (`TrigClientApplication.java` is shown in Figure 16 on the next page.

```
package edu.caltech.trigjclient;

import edu.caltech.sbw.*;

public interface Trigonometry
{
    double sin(double x) throws SBWException;
    double cos(double x) throws SBWException;
}
```

Figure 15: *Contents of the file `Trigonometry.java`.*

The code for the client in Figure 16 uses a Java Swing graphical widget for creating an input dialog box. The user can enter a numerical value for an angle. The program then takes this value and calls `trigSin`, a method that encapsulates invoking the `edu.caltech.trigj` module's "Trig" service. *line 10*

The steps for compiling the code and running it are similar to those detailed in the previous example.

A.3 Example of Finding Multiple Modules in a Category

This example corresponds to the code discussed in Section 3.5. Here we use a Java list and populate it with the names of all known services in the category "Analysis". When the user selects a value from the list, the program asks SBW for a module instance implementing that service and then invokes the method `void doAnalysis(string sbml)` on that module. The variable `sbml` is a model description in SBML format Hucka et al. (2001b), obtained from the user using a separate Java `JEditorPane`.

```java
package edu.caltech.trigjclient;

import javax.swing.*;
import edu.caltech.sbw.*;           // Import SBW.

public class TrigClientApplication
{
  public static void main(String[] args)
  {
      String angleString = JOptionPane.showInputDialog(null, "Enter angle:");
      double angle = Double.valueOf(angleString).doubleValue();

      // The following does its work using the method below.

      double result = trigSin(angle);

      JOptionPane.showMessageDialog(null, "Result: " + Double.toString(result));

      // Done.

      System.exit(0);
  }

  public static double trigSin(double the_angle)
  {
      try
      {
          SBW.connect();

          // Request an instance of the module "edu.caltech.trigj", and
          // look up the service "Trig" on it.

          Module module = SBW.getModuleInstance("edu.caltech.trigj");
          Service service = module.findServiceByName("Trig");

          // Get an object that will act as a proxy for the remote service.

          Trigonometry trig
              = (Trigonometry) service.getServiceObject(Trigonometry.class);

          // Now make use of a method on the service using the proxy object.

          double result = trig.sin(the_angle);

          // We're done, so shutdown the module and disconnect from SBW.

          module.shutdown();
          SBW.disconnect();

          return result;
      }
      catch (SBWException e)
      {
          e.handleWithDialog();
      }

      // Under normal circumstances, things shouldn't get this far.
      return 0;
  }
}
```

Figure 16: *Contents of the file* `Trigonometry.java`*.*

```java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import com.borland.jbcl.layout.*;
import edu.caltech.sbw.*;

interface Analysis
{
    void doAnalysis(String sbml);
}

public class AnalysisDriverFrame
{
  JEditorPane SBMLEditorPane = new JEditorPane();
  JList list = new JList();
  JButton OKButton = new JButton();
  JLabel SBMLLabel = new JLabel();
  JLabel AnalysisLabel = new JLabel();

  public AnalysisDriverFrame(SBWGUIModule p)
  {
    super(p, "Analysis Driver");
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }

  //Component initialization
  private void jbInit() throws Exception
  {
    this.getContentPane().setLayout(null);
    OKButton.setText("OK");
    OKButton.setBounds(new Rectangle(199, 212, 83, 33));
    OKButton.addMouseListener(new java.awt.event.MouseAdapter() {
      public void mouseClicked(MouseEvent e) {
        OKButton_mouseClicked(e);
      }
    });
    list.setBorder(BorderFactory.createLoweredBevelBorder());
    list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    list.setBounds(new Rectangle(72, 150, 292, 49));
    SBMLEditorPane.setBorder(BorderFactory.createLoweredBevelBorder());
    SBMLEditorPane.setBounds(new Rectangle(71, 30, 292, 94));
    SBMLLabel.setText("SBML:");
    SBMLLabel.setBounds(new Rectangle(28, 34, 42, 33));
    AnalysisLabel.setText("Analysis:");
    AnalysisLabel.setBounds(new Rectangle(15, 146, 50, 27));
    this.getContentPane().add(OKButton, null);
    this.getContentPane().add(SBMLEditorPane, null);
    this.getContentPane().add(list, null);
    this.getContentPane().add(SBMLLabel, null);
    this.getContentPane().add(AnalysisLabel, null);

    // Get analysis service records.
    try
    {
```

Example of finding multiple modules in a category (continued on the next page).

```
      ServiceDescriptor[] descriptors = SBW.findServices("Analysis");
      list.setListData(descriptors);
      list.setCellRenderer(new AnalysisCellRenderer());
    }
    catch (SBWException e)
    {
      e.handleWithDialog();
    }
  }

  void OKButton_mouseClicked(MouseEvent e)
  {
    if (!list.isSelectionEmpty())
    {
      String sbml = SBMLEditorPane.getText();

      try
      {
        ServiceDescriptor descriptor
           = (ServiceDescriptor)list.getSelectedValue();
        Service service
           = descriptor.getServiceInModuleInstance();
        Analysis analysis
           = (Analysis)service.getServiceObject(Analysis.class);

        analysis.doAnalysis(sbml);
      }
      catch (SBWException e)
      {
        e.handleWithDialog();
      }
    }
  }
}

public class AnalysisCellRenderer implements ListCellRenderer
{
  private DefaultListCellRenderer defaultRenderer
      = new DefaultListCellRenderer();

  public Component getListCellRendererComponent(JList list, Object value,
                                                int index, boolean isSelected,
                                                boolean hasFocus)
  {
    String name = ((ServiceDescriptor) value).getDisplayName();
    return defaultRenderer.getListCellRendererComponent(list, name, index,
                                                         isSelected, hasFocus);
  }
}
```

Figure 17: *Example of finding multiple modules in a category.*

B The `sbw-make-jar` Program

The standard Java run-time environment version 1.3 and above provides only two ways of starting a Java program. The one that is probably familiar to most users has the following form:

```
java -classpath path... main.entry.point
```

This form requires that the path to the program's files be specified either through the environment variable CLASSPATH, or via the optional *`-classpath path...`* argument, or via a special manifest entry in the program's JAR file(s). In addition, the form of the command above also requires specifying the class holding the entry point of the program.

When using an integrated development environment (IDE), having to specify this information may not be onerous. However, when working with multiple Java programs outside of an IDE, and/or deploying standalone modules, this scheme quickly becomes too tedious. One issue is that the class path must include the files for both the application and the libraries (such as `SBWCore.jar`) on which the application depends. If one is using a CLASSPATH environment variable, each program's JAR file (or directory of class files) must be added to the CLASSPATH. If one is using the command-line argument `-classpath`, the value of the *`path...`* must be adjusted for each program.

The second way of starting a Java program is to use a command of the following form:

```
java -jar jarfile
```

The *`jarfile`* can be constructed in such a way that its manifest specifies the main class to run. If the *`jarfile`* also includes all class files that it depends on (such as the SBW library files), then it is relocatable and does not require any external CLASSPATH environment. This form of Java program invocation is obviously much simpler than the first form, and has the added advantage that on Microsoft Windows and Sun Solaris systems, simply double-clicking on the *`jarfile`* name in a file explorer window will start up the Java program. Isn't that nice?

SBW provides a utility called `sbw-make-jar` for constructing stand-alone JAR files suitable for use with this second form of Java program invocation. `sbw-make-jar` is a command-line utility installed in the "`bin`" subdirectory of *`SBW_HOME`* available for Linux and Cygwin (in Windows) environments. It can be invoked using a command line with the following form:

```
sbw-make-jar -a addJAR -d jarname mainclass files...
```

The meanings of the three required arguments are:

`jarname`: The filename of the final JAR file that will be created.

`textitmainclass`: The Java class (in full dotted classpath notation) that contains the main entry point of the program.

`textitfiles...`: A list of all files that are to be incorporated in the *jarname* produced. Usually these are `.class` files, but depending on the application, they may perhaps include other resource files.

The underlined arguments are optional and have the following meanings:

`-a` *addJAR*: This flag requires one argument, *addJAR*. The program `sbw-make-jar` will extract all of the files in the named JAR file *addJAR* and merge them into the final JAR file (*jarname*). Multiple `-a` *addJAR* arguments can be given. You must use this flag if your Java application depends on additional libraries besides the SBW Java library and the standard Java run-time libraries.

`-d`: By default, `sbw-make-jar` uses the optimized version of the SBW library from the SBW installation (i.e., the file *`SBW_HOME`*`/lib/SBWCore.jar`). The optional flag `-d` instructs the program to use the debugging version of the SBW library (i.e., *`SBW_HOME`*`/lib/SBWCore-debug.jar`).

The program `sbw-make-jar` builds a JAR file that contains all of the files listed on the command line *and* the class files that comprise the SBW core library. This has two implications. First, the resulting JAR file is larger than if the SBW core library was left out and referenced through the CLASSPATH environment variable or other means. Second, **if the SBW library changes, the JAR file must be rebuilt**. These issues must be traded off against the ease of invocation and portability afforded by using this approach.

As an aside, we have tested many other approaches besides this, in order to make the invocation of SBW Java applications as simple as possible. For example, it is possible (in Java versions 1.3 and later) to place a classpath specification in the manifest of a JAR file. We have tried using this to make JAR files refer to an external copy of the SBW Java library instead of incorporating the SBW classes. However, there is a limitation in what can be placed in the class path specified by the manifest of a JAR file: the items must be relative pathnames. If the dependent libraries are not located in the same directory as the JAR file itself, constructing a valid pathname can become difficult and dependent on the installation of the user/developer constructing the JAR file. It also means that the JAR file is not portable to other systems. Other approaches have had other limitations, and so we have settled on constructing the analogs of statically-linked executables for JAR modules.

C Setting Up SBW for Distributed SBW Operation

Although many situations involve all modules running on the same host computer, SBW supports the ability to run modules on remote hosts as well. Each remote host runs a separate Broker process; modules on that host connect to the Broker (the local Broker for that host), and each Broker Broker arranges to route messages between modules whether they are local or remote. Every Broker thus knows about every other Broker's modules, and offers an amalgamated view to any module that queries for available modules and services. Figure 18 illustrates the organization of modules and Brokers when multiple Brokers are running.

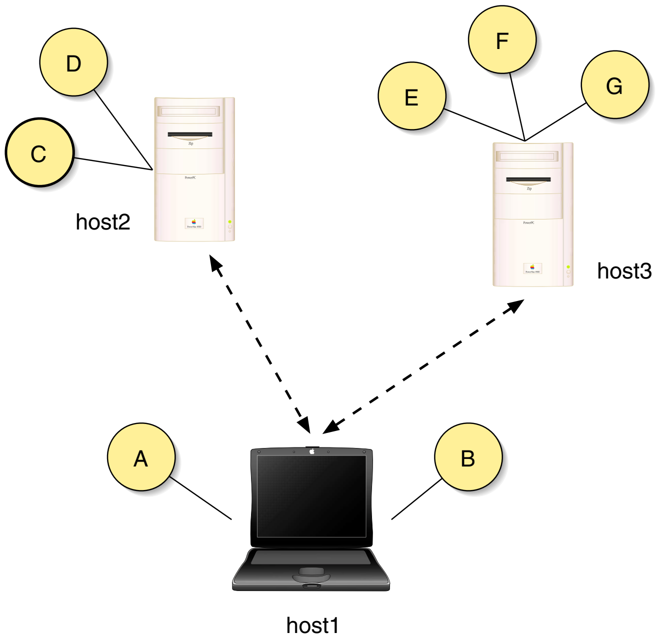

Figure 18: *Example of multiple Brokers and multiple modules running on separate computers. Each module (represented by the yellow circles) connects only to the Broker on the local host; the Brokers on separate hosts connect to each other and route messages between modules as appropriate.*

- `psbw.getModuleInstance(moduleName)`, described in Section **??**, accepts module names of the form `hostName:moduleName`. If the argument has this form, then the `hostName` part is used as the name of the remote host where the module should be started.
- `psbw.link(hostNameOrAddress)`, described in Section **??**, allows a program to connect explicitly to a Broker running on the host indicated by parameter `hostNameOrAddress`. This is useful when a module wishes to browse the list of modules and services known to a remote Broker. However, in practice, most remote Brokers and modules are started using the `getModuleInstance` call described above.

SBW uses SSH (Barrett and Silverman, 2001) to start remote Brokers and establish a secure tunnel connection between Brokers on separate hosts. In order for this to work, the following conditions must be met:

- The remote host must be running the SSH daemon and allow remote SSH connections. This is the case for many Unix/Linux hosts; if SSH is not enabled on the target host, you must either enable it yourself (if you have administrative control over the computer), or ask your systems administrator to enable SSH for you. If the remote host is running Microsoft Windows, you will first have to install an SSH daemon package, and then enable incoming SSH connections on that host. The steps for doing this are beyond the scope of this API manual. [Since SSH is a very popular system, it is easy to find instructions on the internet and in books such as the one by Barrett and Silverman (2001).]

- You must have set up your SSH accounts in such a way that you can use SSH to run commands on the remote host without having to supply a password. A good test to verify that this is the case (at least in the case of a remote Unix/Linux host) is to try typing a command such as the following in a command shell:

  ```
  ssh remotehost date
  ```

  The `date` command simply returns a one-line reply indicating the current date. If SSH is set up properly for password-less execution, then the command above should *not* prompt you for a password. If it prompts you for a password, your SSH account or other elements are not set up properly.

Once SSH is set up properly, it is possible to start remote modules and Brokers seamlessly using the two API calls listed above.

References

Arkin, A. P. (2001). *Simulac* and *Deduce*. Available via the World Wide Web at `http://gobi.lbl.gov/~aparkin/Stuff/Software.html`.

Ascher, D., Dubois, P. F., Hinsen, K., Hugunin, J., and Oliphant, T. (2001). Numerical Python. Available via the World Wide Web at `http://www.numpy.org`.

Barrett, D. J. and Silverman, R. E. (2001). *SSH, the Secure Shell*. O'Reilly & Associates, Sebastopol, CA, USA.

Bray, D., Firth, C., Le Novère, N., and Shimizu, T. (2001). *StochSim*. Available via the World Wide Web at `http://www.zoo.cam.ac.uk/comp-cell/StochSim.html`.

Bray, T., Paoli, J., and Sperberg-McQueen, C. M. (1998). Extensible markup language (XML) 1.0, W3C recommendation 10-February-1998. Available via the World Wide Web at `http://www.w3.org/TR/1998/REC-xml-19980210`.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons.

Falangan, D. (1999). *Java^{TM} in a Nutshell*. O'Reilly & Associates, Sebastopol, CA, USA.

Ginkel, M., Kremling, A., Tränkle, F., Gilles, E. D., and Zeitz, M. (2000). Application of the process modeling tool ProMot to the modeling of metabolic networks. In Troch, I. and Breitenecker, F., editors, *Proceedings of the 3rd MATHMOD*, pages 525–528.

Goryanin, I. (2001). *DBsolve*: Software for metabolic, enzymatic and receptor-ligand binding simulation. Available via the World Wide Web at `http://homepage.ntlworld.com/igor.goryanin/`.

Goryanin, I., Hodgman, T. C., and Selkov, E. (1999). Mathematical simulation and analysis of cellular metabolism and regulation. *Bioinformatics*, 15(9):749–758.

Hucka, M., Finney, A., Sauro, H. M., and Bolouri, H. (2001a). Introduction to the Systems Biology Workbench. Available via the World Wide Web at `http://www.sbw-sbml.org/`.

Hucka, M., Finney, A., Sauro, H. M., and Bolouri, H. (2001b). Systems Biology Markup Language (SBML) Level 1: Structures and facilities for basic model definitions. Available via the World Wide Web at `http://www.sbml.org`.

Hucka, M., Finney, A., Sauro, H. M., Bolouri, H., Doyle, J. C., and Kitano, H. (2001c). The ERATO Systems Biology Workbench: Architectural evolution. In Yi, T.-M., Hucka, M., Morohashi, M., and Kitano, H., editors, *Proceedings of the Second International Conference on Systems Biology*, pages 352–361. Omnipress, Inc.

Hucka, M., Finney, A., Sauro, H. M., Bolouri, H., Doyle, J. C., and Kitano, H. (2002). The ERATO Systems Biology Workbench: Enabling interaction and exchange between software tools for computational biology. In Altman, R. B., Dunker, A. K., Hunker, L., Lauderdale, K., and Klein, T. E., editors, *Pacific Symposium on Biocomputing 2002*. World Scientific Press.

Mendes, P. (1997). Biochemistry by numbers: Simulation of biochemical pathways with Gepasi 3. *Trends in Biochemical Sciences*, 22:361–363.

Mendes, P. (2001). Gepasi 3.21. Available via the World Wide Web at `http://www.gepasi.org`.

Morton-Firth, C. J. and Bray, D. (1998). Predicting temporal fluctuations in an intracellular signalling pathway. *Journal of Theoretical Biology*, 192:117–128.

Sauro, H. M. (2000). Jarnac: A system for interactive metabolic analysis. In Hofmeyr, J.-H. S., Rohwer, J. M., and Snoep, J. L., editors, *Animating the Cellular Map: Proceedings of the 9th International Meeting on BioThermoKinetics*. Stellenbosch University Press.

Sauro, H. M. and Fell, D. A. (1991). SCAMP: A metabolic simulator and control analysis program. *Mathl. Comput. Modelling*, 15:15–28.

Sauro, H. M., Hucka, M., Finney, A., and Bolouri, H. (2001). The Systems Biology Workbench concept demonstrator: Design and implementation. Available via the World Wide Web at `http://www.cds.caltech.edu/erato/sbw/docs/detailed-design/`.

Schaff, J., Slepchenko, B., and Loew, L. M. (2000). Physiological modeling with the Virtual Cell framework. In Johnson, M. and Brand, L., editors, *Methods in Enzymology*, volume 321, pages 1–23. Academic Press, San Diego.

Schaff, J., Slepchenko, B., Morgan, F., Wagner, J., Resasco, D., Shin, D., Choi, Y. S., Loew, L., Carson, J., Cowan, A., Moraru, I., Watras, J., Teraski, M., and Fink, C. (2001). Virtual Cell. Available via the World Wide Web at `http://www.nrcam.uchc.edu`.

Sun Microsystems (2001). Java Development Kit 1.3. Available via the World Wide Web at `http://java.sun.com/j2se/1.3/`.

Tomita, M., Hashimoto, K., Takahashi, K., Shimizu, T., Matsuzaki, Y., Miyoshi, F., Saito, K., Tanida, S., Yugi, K., Venter, J. C., and Hutchison, C. (1999). E-Cell: Software environment for whole cell simulation. *Bioinformatics*, 15(1):72–84.

Tomita, M., Nakayama, Y., Naito, Y., Shimizu, T., Hashimoto, K., Takahashi, K., Matsuzaki, Y., Yugi, K., Miyoshi, F., Saito, Y., Kuroki, A., Ishida, T., Iwata, T., Yoneda, M., Kita, M., Yamada, Y., Wang, E., Seno, S., Okayama, M., Kinoshita, A., Fujita, Y., Matsuo, R., Yanagihara, T., Watari, D., Ishinabe, S., and Miyamoto, S. (2001). E-Cell. Available via the World Wide Web at `http://www.e-cell.org/`.