JWIG User Manual

Aske Simon Christensen & Anders Møller

June 2002 (revised January 2004)

Contents

1 Introduction

The JWIG programming language is a Java-based high-level language for development of interactive Web services. It contains an advanced session model, a flexible mechanism for dynamic construction of XML documents, in particular XHTML, and a powerful API for simplifying use of the HTTP protocol and many other aspects of Web service programming. To support program development, JWIG provides a unique suite of highly specialized program analyses that at compile time verify for a given program that no runtime errors can occur while building documents or receiving form input, and that all documents being shown are valid according to the document type definition for XHTML 1.0 [7]. The main goal of the JWIG project is to simplify development of complex Web services, compared to alternatives, such as, Servlets, JSP, ASP, and PHP. JWIG is a descendant of the <bigwig> research language.

JWIG is an extension of the Java programming language [1] with specialized syntactic constructs for building and showing XHTML documents and receiving form input. JWIG programmers are therefore assumed to be familiar with Java and XHTML.

The JWIG 1.2 development system consists of a runtime system based on the Apache Web server, a compiler, and a program analyzer. The full source code and documentation is freely available from the project home page at `http://www.brics.dk/JWIG/`. JWIG is being developed at the BRICS research center at University of Aarhus.

This manual describes JWIG 1.2. In the first sections, we explain how to install and run the JWIG system. We then go through the JWIG language and API. Finally, the special JWIG program analyses are described. In the research paper *Extending Java for High-Level Web Service Development* [4], the motivation and goals for the JWIG project are more thoroughly described, and, in particular, the program analysis are explained in full detail. The online tutorial *Interactive Web Services with Java* [6] contains a section that briefly summarizes the highlights of JWIG. This present manual aims to provide a concise but complete reference to the JWIG language and development system.

2 Installation of the JWIG System

The installation procedure described here typically requires root permissions on your system. After the JWIG system has been installed, installation of Web services can be done by all users without any special permissions.

Prerequisites

To install the JWIG system, you need:

- a Linux, Solaris, or IRIX machine (it may run on other Unix variants, but Windows is currently not supported)

- the Java 2 Platform, Standard Edition[1], version 1.3.1 or later (or compliant tools)
- the Apache Web server[2] with DSO (dynamic shared objects) enabled

How to install the JWIG system

The JWIG runtime system uses the `runwig` package, which contains a module for the Apache server and a garbage collection daemon.

Download the JWIG and the `runwig` source packages, and unpack them with

```
gunzip -c runwig-2.1-1.tar.gz | tar xvf -
gunzip -c jwig-1.2-1.tar.gz | tar xvf -
```

If your system supports RPM package management, we recommend that you build and install binary RPM packages tailor-made for your system configuration:

```
cd runwig-2.1
config/rpm-config
make rpm

cd ../jwig-1.2
config/rpm-config
make rpm

cd ‘rpm --eval %{_rpmdir}/%{_arch}‘
rpm -U runwig-2.1-1.*.rpm jwig-1.2-1.*.rpm
```

If your system does not support RPM package management or you encounter any configuration problems, read the `INSTALL` files in the `runwig-2.0` and `jwig-1.2` directories for further instructions. The packages use the familiar Autoconf system for easy configuration.

JWIG mode for Emacs

The JWIG distribution contains a JWIG mode for Emacs. To activate it, insert the following into your `~/.emacs` file:

```
(global-font-lock-mode t)
(load "JWIGDIR/xml-mode")
(load "JWIGDIR/jwig-mode")
(setq auto-mode-alist
  (append '(("\\.jwig$" . jwig-mode)) auto-mode-alist))
```

where *JWIGDIR* is replaced by the result of running `jwig jwigdir`.

[1] `http://java.sun.com/j2se/`
[2] `http://httpd.apache.org/`

More information

More detailed information about compilation, installation, and configuration can be found in the `INSTALL` files in the distribution and in the manual page for `jwig`. If there are other problems not covered by the documentation, feel free to send an email to amoeller@brics.dk.

For information about compilation and installation of JWIG services, see the next section.

3 Compilation and Installation of JWIG Services

The `jwig` tool makes it easy to compile and install JWIG services. As an example, copy the `Hello.jwig` service shown below to a directory named `test` and follow the steps described below. This tool can also be used to update an installed service and to analyze a service to check for errors related to construction of XHTML and receiving of form input. See also the manual page (`man jwig`) for more documentation.

The `Hello.jwig` example program:

```
package test;
import dk.brics.jwig.runtime.*;

public class Hello extends Service {
    public class Test extends Session {
        public void main() {
            exit [[ <html><head><title>JWIG</title></head><body>
                    <h1>Hello World!</h1>
                    </body></html> ]];
        }
    }
}
```

Compiling a Service

To compile a JWIG program, run

```
jwig compile files
```

where *files* is the list of `.jwig` and `.java` files constituting the program source. This will translate the JWIG files into Java code and then compile all the Java code to class files. The command must be executed from the root of the directory structure containing the source files, and the source files must appear in subdirectories matching the package names. The `CLASSPATH` environment variable can be used to specify the classpath.

For the `Hello.jwig` example, run

```
jwig compile test/Hello.jwig
```

This will generate the files `test/Hello.class` and `test/Hello$Test.class`.

Installing a Compiled Service

To install a compiled service, run

```
jwig install dir files
```

where *dir* is a subdirectory of `~/jwig-bin` and *files* is the list of class files and extra files used by the service. For each file in the list, the relative path is preserved in the installation. The path to each class file in the list must match the package name of the class. All `jar` files placed in the install directory are automatically included in the classpath when the service is started.

(The directory `~/jwig-bin` is determined by the configuration of the Apache module. It can be changed in `/etc/httpd/conf/bigwig.conf`, assuming a typical JWIG install configuation.)

For the `Hello.jwig` example, run

```
jwig install ~/jwig-bin/demo test/Hello.class test/Hello$Test.class
```

This will install the service in the *service directory* `~/jwig-bin/demo`. The class files are placed in `~/jwig-bin/demo/test` to match the package name.

Running an Installed Service

To run an installed service from a browser, open the following URL:

```
http://host/jwig-user/subdir/class
```

where *host* is your server host name, *user* is your user name, *subdir* is the directory where the service is installed relative to `~/jwig-bin`, and *class* is the name of a class defining a service thread. The class name must be fully qualified with package name.

For the `Hello.jwig` example, view

```
http://host/jwig-user/demo/test.Hello*Test
```

Note that "`.`" is used in the package name of the class, and that "`*`" is used instead of `$` (because `$` should not appear unescaped in URLs).

(The mapping from URLs to files on the file system is determined by the configuration of the Apache module mentioned above. Normally, the URL part `/jwig-`*user*`/*` is mapped to `~`*user*`/jwig-bin/*` on the file system. More details of what is happening when services are running are described in the runtime system section.)

Uninstalling a Service

To uninstall a service, run

```
jwig uninstall dir
```

where *dir* is the directory containing the installed service.

For the `Hello.jwig` example, run

```
jwig uninstall ~/jwig-bin/demo
```

This will uninstall the service and remove `~/jwig-bin/demo` entirely.

4 Constructing XHTML Documents

All output from a JWIG service has the form of XHTML documents, which are sent to the client to be shown in a browser window. These XHTML documents are constructed in the JWIG program by means of *XML templates*. An XML template is a fragment of XML syntax that may contain unspecified pieces called *gaps*. These gaps can be filled with strings or other templates to construct larger templates.

XML templates are represented in the JWIG program by values of the `XML` type. `XML` values are immutable. All operations that manipulate XML templates create new values.

XML template constants are written as plain XML text enclosed in double square brackets, `[[ ... ]]`. For example, the following piece of code declares a variable to contain XML template values and initializes it to `Hello <i>World</i>!`:

```
XML hello = [[Hello <i>World</i>!]];
```

XML template constants must be well-formed, that is, the tags must be balanced and nest properly. Implicitly, the default namespace is set to `http://www.w3.org/1999/xhtml` (the XHTML 1.0 namespace).

Gaps and Plugs

XML template constants can contain named gaps, using the syntax `<[`*name*`]>`. The gap name must be a legal Java identifier. These gaps can be placed within text and tags, as if they were themselves tags. Gaps placed like this are called *template gaps*.

Gaps can also be placed inside tags, as the values of attributes. The syntax for this is to write the name of the gap enclosed in square brackets where you would normally write an attribute value enclosed in single or double quotes. For example, `<a href=[link]>` is an anchor start tag whose `href` attribute is a gap named `link`. Gaps placed like this are called *attribute gaps*.

Gaps are filled using the *plug* operator. This is an expression of type XML with the syntax *exp1* `<[`*name* `=` *exp2*`]`, where *exp1* is an expression of type `XML`, *name* is an identifier indicating a gap name, and *exp2* is an expression of type `XML` or `String`. The result of this expression is a copy of *exp1* where all occurrences of gaps named *name* are replaced by a copy of *exp2*. All gaps in *exp1* with names different from *name* will still be present in the result, and all gaps in *exp2* are copied into the result as well, regardless of their name. Note that neither *exp1* nor *exp2* are modified in the operation.

Example:

```
XML hello1 = [[<p align=[alignment]>Hello <[what]>!</p>]];
XML hello2 = hello1 <[what = [[<i><[thing]></i>]]];
XML hello3 = hello2 <[thing = "World"] <[alignment = "left"];
```

After executing this code, the values of the variables will be:

```
hello1: <p align=[alignment]>Hello <[what]>!</p>
hello2: <p align=[alignment]>Hello <i><[thing]></i>!</p>
hello3: <p align="left">Hello <i>World</i>!</p>
```

As can be seen from the example, both strings and XML templates can be plugged into template gaps. Attribute gaps, however, can only contain strings, since plugging an XML template into an attribute gap would result in malformed XML. Attempting this will result in an exception being thrown at runtime, as described later.

As a shorthand, the plug operator allows the plugging of any object or simple value. This will plug the corresponding string representation into the gap, as if the plugged value was passed through `String.valueOf()`. An exception to this is when the plugged object is a `Class` object representing a session class (described in the next section). This will plug the URL of the session (as described in the previous section) into the gap as a string. This is convenient for making links that start new sessions.

When a document is *shown*, as explained in the next section, all remaining template gaps in the shown document are replaced by empty strings. For all remaining attribute gaps, the whole attribute is removed. This can be used to control, dynamically, which attributes will be present in a given tag.

Code Gaps

In addition to template gaps and attribute gaps, which are filled using the plug operator, XML templates can contain inlined pieces of code, called *code gaps*. The syntax for a code gap is `<{`*code*`}>`, where *code* is Java code as it would appear in a normal Java method body. This code will be executed when the document is shown, and its result (indicated by `return` statements) is inserted at the point in the document where the code gap was.

Example:

```
XML date = [[The current date is now <i><{ return new Date(); }></i>]];
```

When a document containing the value given to the `date` variable above is shown, the current date and time (as returned by `new Date().toString()`) will appear in place of the code gap. Note that any value can be returned by a code gap. It is treated as if it was plugged into a template gap.

Code gaps are executed in the order they appear in the document. If a code gap returns an XML template which itself contains code gaps, these will be executed as they appear, before processing subsequent code gaps in the original document.

External XML Templates

Instead of being written inlined in the JWIG program, an XML template constant can be placed in an external file and loaded into the program at runtime using the construct `get` *`url`*, where *`url`* is a string constant indicating the location of the template constant. This file should be a plain XML fragment (without the double square brackets).

Example:

```
XML doc = get "http://www.brics.dk/JWIG/test.xml";
```

When this code is executed, the designated file will be fetched and parsed, and the resulting XML template will be pointed to by `doc`.

The retrieval of the document will happen every time the `get` *`url`* expression is evaluated. This makes it possible to change the appearance of a running service simply by changing the external templates. (If this behavior is not needed or not desired, the `get` *`url`* expression should be used in such a way that the template is not fetched every time its value is needed, since this could drastically impair the performance of the service.)

An external XML template can contain template gaps and attribute gaps just like an inlined one. However, it cannot contain code gaps, since this would require runtime compilation of code into the running service.

Exceptions

A number of exceptions can be thrown as a result of XML template manipulation. These are:

`PlugException`: Thrown when an attempt is made to plug into a gap that does not occur in the given document or when an XML template is plugged into an attribute gap.

`XMLException`: Thrown when a syntax error is encountered in an XML template constant. For inline template constants, the exception is thrown when the constant expression is evaluated. For external templates, it is thrown when the `get` *`url`* expression is evaluated, during parsing of the loaded file. A code gap in an external XML template will also cause this exception to be thrown.

`IOException`: Thrown by the `get` *`url`* expression if the specified file could not be found, or if some other I/O error occurred during the fetching.

The JWIG static analysis is able to verify statically for a given JWIG program, that none of the two first of these exceptions can occur at runtime, provided that no external XML templates have been changed since the time of the analysis. In addition, the analysis also checks that all documents being constructed are valid XHTML 1.0, which is the XML variant of HTML 4.01.

5 Services and Sessions

JWIG is a session-centered language, in contrast to most other Web service programming languages. A *session* consists of a sequence of interactions between a server and a client. Sessions are initiated by the clients but are controlled by the service code, conceptually by threads running on the server. An interaction is performed by sending a Web page with a form to the client. The session thread waits for the client to fill out and submit the form, and then continues execution. A *service* consists of a number of session types, which are entry points for the session threads.

The Structure of a JWIG Program

A JWIG program consists of a *service class*, which is a subclass of `Service`. It will be instantiated upon the first client request to the service, and only a single instance will exist over the entire lifetime of the service.

The service class contains a number of *session classes*. These are non-static inner classes of the service class, and are subclasses of `Service.Session`. Each client request to start a session will instantiate the corresponding session class and run its `main` method in a freshly created thread. The session object will exist until the session terminates, either explicitly by executing an `exit` statement (described below), or implicitly due to a timeout caused by a nonresponding client.

Client interaction is performed through the `show` and `exit` statements. The statement `show` *`exp`*, where *`exp`* is an expression of type `XML`, sends the given document to the client, waits for the client's response, and then resumes execution of the session thread. The statement `exit` *`exp`*, where *`exp`* again is of type `XML`, sends the given document to the client and then terminates the session thread. If the session thread falls off the end of the `main` method, it will implicitly execute `exit` with a standard "Session terminated" document.

A document shown to the client, using either `show` or `exit`, must consist of a single `<html>` ... `</html>` element. The XHTML 1.0 document type declaration and a character encoding are automatically inserted. Any document given as argument to the `show` statement should contain at least one `form` element with no `action` attribute. This will instruct the JWIG runtime system to insert a special `action` attribute that will resume execution of the session thread. The client requests continuation of the session by submitting such a form. Other `form`

elements with explicit `action` attributes may be present in the document, for submitting to other kinds of services. These are ignored by JWIG.

The values of the input fields in the submitted form can be read using the `receive` *name* expression. Here, *name* is the name of the field to read, and the result of the expression is a `String` containing the value submitted for the field. If no such field was submitted, or if more than one value was submitted for it, a `ReceiveException` is thrown. To read the values of an input field for which more than one value (or a varying number of values) is submitted, the expression `receive[]` *name* is used. This returns a `String` array containing all the values of the field, in the order in which they were submitted. This expression never fails. If no values were submitted for the given field, an empty array is returned.

Example:

```
import dk.brics.jwig.runtime.*;

public class Hello extends Service {
    private static final XML wrap = [[
        <html>
          <head>
             <title>A JWIG example</title>
          </head>
          <body>
            <[body]>
          </body>
        </html>
    ]];

    private static final XML question = [[
        <form>
           What is your name?
           <input type="text" name="person" />
           <br/>
           <input type="submit" name="answer" value="Answer" />
        </form>
    ]];

    private static final XML greeting = [[
        <h1>Hello <[who]>!</h1>
    ]];

    public class Test extends Session {
        public void main() {
            show wrap <[body = question];
            exit wrap <[body = greeting <[who = receive person]];
        }
    }
}
```

This JWIG program defines a service called `Hello` with a single session called `Test`. The

session shows to the client a document containing a text input field named `person` and a submit button entitled "Answer". The session then waits for the client to press this button. The name entered in the text field is read using the `receive person` expression. This value is plugged into the `who` gap of the `greeting` template, and the resulting document is shown as the final output of the session. After sending this document to the client, the session terminates, without waiting for the client to respond. Notice the use of a wrapper template (the template assigned to the `wrap` variable) to supply the outer structure at every `show` and `exit` statement. This is a standard technique used in most JWIG programs.

Input Fields

Input fields in the submitted form result in name/value pairs being sent back to the server. These pairs are accessible from the JWIG program through the `receive` *name* and `receive[]` *name* constructs. To make it easier to handle the received information, some input fields behave somewhat differently than normal, as seen by the JWIG program. For this reason, the exact behavior of the different kinds of input fields are described in the following:

- A **text input** (an `input` element with `type="text"` or `type="password"`, or a `textarea` element) returns its contents as a single value under the name given in its `name` attribute.
- A group of **radio buttons** (`input` elements with `type="radio"` having the same value in their `name` attribute) return the value of the `value` attribute of the selected button as a single value under the name given in the `name` attribute of the radio buttons. If no button was selected (which can happen in some browsers if no button is initially selected using the `checked` attribute), `null` is returned.
- A group of **checkboxes** (`input` elements with `type="checkbox"` having the same value in their `name` attribute) return the values of the `value` attributes of all the checked checkboxes. Since the number of values vary, they must be received using the `receive[]` *name* construct.
- A **single-select menu** (a `select` element without the `multiple` attribute) returns the value of the `value` attribute of the selected entry (`option` element) as a single value under the name given in the `name` attribute of the `select` element. If no entry was selected (which can happen in some browsers if no entry is initially selected using the `selected` attribute of the `option` element), `null` is returned.
- A **multi-select menu** (a `select` element with the `multiple` attribute) returns the values of the `value` attributes of all the selected entries. Since the number of values vary, they must be received using the `receive[]` *name* construct.
- A **file select control** (an `input` element with `type="file"`) returns several values describing the file. If the name of the control is *name*, then receiving *name* will give the actual contents of the file, *name*`.filename` will give the name of the file, *name*`.contenttype` will give the content type of the file, and *name*`.charset` will give the character encoding used to encode the file into a string. All these values are strings, since a `receive`

expression always returns a string. The original byte sequence of the file data can be retrieved by decoding the file content string according to the given character encoding. This will either be the one specified by the browser or, if this is empty or not recognised as a valid character encoding, some other valid encoding chosen by the JWIG runtime system. The `contenttype` might specify a content type or be an empty string, depending on the behavior of the browser. Note that the `form` containing the file select control must have the attribute `enctype="multipart/form-data"` for file upload to work as described here.

- A **submit button** (an `input` or `button` element with `type="submit"`) returns the value of its `name` attribute under the name "submit".
- A **graphical submit button** (an `input` element with `type="image"`) returns the value of its `name` attribute under the name "`submit`", just like a normal submit button. Furthermore, receiving the names "`submit.x`" and "`submit.y`" will give the coordinates inside the image at which the client clicked.

If the form was submitted but no submit button was pressed (which can happen in some browsers), receiving the name "`submit`" will give an empty string. If a non-graphical submit button was pressed, receiving the names "`submit.x`" and "`submit.y`" will both give `-1`.

The special behavior of submit buttons makes it easy to decide which submit button was pressed, since its name always appears under the name "`submit`". The `value` attribute of submit buttons can then be used to specify the textual label on the button, independent of the internal handling of the button.

In addition to the normal and graphical submit buttons, JWIG includes a special kind of submit button - the **submit anchor**. If an `a` element has a `submit` attribute but no `href` attribute, the whole anchor will act as a submit button with the given name. This way, anything that can be an anchor can also be a submit button. This is implemented by JavaScript code in the `href` attribute. If the `a` element in question furthermore contains a `status` attribute, this will be the text shown in the status line of the browser when the mouse is over the anchor.

Document Post-Processing

The XHTML specification require that lists, tables, menus, and some other constructs must contain at least one element. Requirements like this are very inconvenient when the output is generated dynamically, since any code that generates, for instance, a list with a dynamic number of elements must handle zero elements as a special case. For this reason, JWIG does some preprocessing of the generated XHTML to allow such constructs to be generated by the program and still show only valid XHTML to the client. Specifically, the post-processing does the following:

- The following elements are removed: `ul`, `ol`, `menu` and `dir` elements containing no `li` elements, `dl` elements containing no `dt` or `dd` elements, `map` elements containing no `area` elements, `tr` elements containing no `th` or `td` elements, `thead`, `tbody` and

`tfoot` elements containing no `tr` elements, `table` elements containing no `tbody` or `tr` elements, and `optgroup` elements containing no `option` or `optgroup` elements.

- `select` elements containing no `option` or `optgroup` elements are given a single, initially selected option with an empty string as its `value` attribute.

Furthermore, for `form` elements without a `method` attribute, `method="POST"` is inserted by default.

Pages and seslets

Service.Session is not the only inner class of the Service class that can be subclassed to create client-instantiable server threads. Two more such classes exist - Service.Page and Service.Seslet.

If a session always proceeds directly to an `exit` statement without ever executing a `show` statement, it can be a page instead of a session. A page is just like a session, except that `shows` and `receives` are not allowed. Its `main` method is invoked in exactly the same manner. For this single-page purpose, pages are usually more efficient than sessions.

A seslet is used for arbitrary communication with a non-browser client, such as, an applet or another Web service. As seslet's `main` method takes as arguments an `InputStream` and an `OutputStream` for the communication. A seslet produces no XHTML output. It should simply terminate when the communication is over. On the client side, an `URLConnection` should be opened to the seslet URL. The streams obtained by calling `getInputStream()` and `getOutputStream()` will then be connected to the seslet streams.

Exceptions

A number of exceptions can be thrown by the `show`, `exit` and `receive` constructs:

`ShowException`: Thrown by the `show` and `exit` statements if some error, e.g. an I/O error, occurs while showing the document.

`ValidateException`: If runtime validation is enabled (by the `validate_xhtml` field), this exception is thrown by the `show` and `exit` statements if the shown document does not validate according to the XHTML 1.0 specification.

`TimeoutException`: Thrown by the `show` statement if the client does not respond within the show timeout set for the session.

`CodeGapException`: Thrown by the `show` and `exit` statements if some error occurs while executing the code gaps in the document, or if an exception is thrown from within a code gap.

`ReceiveException`: Thrown by the `receive` expression if no field of the given name was submitted by the client, or if more than one value was submitted for it.

The JWIG static analysis is able to verify statically for a given JWIG program, that neither the ValidateException nor the ReceiveException can occur at runtime, provided that the client behaves according to the specification with respect to the submitted field values, and that no external XML templates have been changed since the time of the analysis.

6 Static Analysis of JWIG Programs

The unique design of JWIG allows some specialized program analyses to be performed, such that the programmer can check at compile time that certain kinds of errors related to the dynamic document construction cannot occur at runtime. The JWIG analyzer considers the following properties for a given program:

plug consistency: that gaps are always present when subjected to the plug operation and XML templates are never plugged into attribute gaps

receive consistency: that input fields occur the right number of times in the shown documents so `receive` and `receive[]` operations always succeed

show validity: that all documents being shown are valid XHTML 1.0 [7]

If any of these correctness properties is not satisfied at runtime, an exception will be thrown, as described earlier. The analyses try to verify at compile time that these exceptions cannot occur, and if then can occur, an explanatory warning message is automatically produced.

To analyze a JWIG program, run

```
jwig analyze files
```

where *files* is a collection of class files from the compiled program. This collection is called the *application classes*. For efficiency reasons, the application classes can be just the few classes that actually constitute the JWIG service, not including all the standard Java classes that the program uses. Exactly one of the application classes must be a subclass of `Service`.

As an example, if we try analyzing a buggy version of the guessing game example from the JWIG distribution, the following errors are reported (abbreviated with ". . ."):

```
Gap ‘holder’ does not exist on line 67
Field ‘name’ is not always available exactly once on line 70
*** Invalid XHTML at line 69
--- element ’html’ in XML template at line 10: illegal content:
<sequence>
 <element xmlns:h="http://www.w3.org/1999/xhtml" name="h:head" />
 <element xmlns:h="http://www.w3.org/1999/xhtml" name="h:body" />
</sequence>
’body’ <- XML template at line 14 in class BuggyGuess in template plug
          into ‘body’ at line 49 in class BuggyGuess$Play
...
```

The first line explains that a plug operation on line 67 in the source program may fail because the specified gap does not exist. Similarly, the next line means that a receive operation may fail. The remaining lines mean that the `show` operation on line 69 may send invalid XHTML to the client. The XML fragment in the error message is the part of the DSD2 schema constraint that is violated. In this example, the validity requirement that `html` elements always must contain a `head` element followed by a `body` element is not satisfied. The last line shows the relevant plug operations.

The soundness of the analyses is based on a set of well-formedness assumptions:

- all invocation sites in the application classes must either always invoke methods in the application classes or always invoke methods in the non-application classes
- no fields or methods of application classes are accessed by a non-application class
- no XML operations are performed in non-application classes
- XML casts are always valid, according to the definition in the previous section

These assumptions usually do not limit expressibility in practice. In some cases, the second assumption can be relaxed slightly, for instance if some method called from a non-application class does not modify any `String` or `XML` value that will ever reach other application class methods. This makes it possible to safely use callback mechanisms such as the `Comparator` interface.

The current prototype implementation may require large amounts of memory when analyzing complex JWIG programs. The running times for the analysis typically range from a few seconds to a number of minutes, depending on the program size and complexity. Analyzing a JWIG program is recommended, not after every single compilation, but periodically as a supplement to extensive testing before the deployment.

The technical details of the JWIG program analyses are explained in the research paper *Extending Java for High-Level Web Service Development* [4]. The JWIG development team is working on improving the analysis implementation to produce more precise error messages and decrease the time and space requirements.

News in JWIG 1.1: The program analysis now uses the Java String Analyzer, which is described in the paper *Precise Analysis of String Expressions* [5]. This analyzer is able to track the values of string expressions more precisely than previously possible.

7 The JWIG Runtime System

The JWIG runtime system is based on the `jwig.jar` package running on a standard Java virtual machine, such as J2SE, together with the `runwig` package for the Apache Web server.

The `runwig`[3] package consists of

- `mod_bigwig` - an Apache server module, handles communication between the client's browsers and the JVM running the service code, and
- `bigwigd` - a garbage collection daemon,

This package is also used in the `<bigwig>` system (hence the naming). The structure of the runtime system in its initial design is described in the research paper *A Runtime System for Interactive Web Services* [3].

Configuration

The `mod_bigwig` module is configured by a file `bigwig.conf`, which is included by `httpd.conf`. This configuration file defines

1. how URLs defining JWIG requests are mapped to file system paths, and
2. how the JVM is started when the first service thread is initiated.

The default configuration is sufficient for typical use. A URL of the form

PROTOCOL`://`*HOST*`/jwig-`*USER*`/`*PATH*

is mapped to

`~`*USER*`/jwig-bin/`*PATH*

on the file system.

[3] `http://www.brics.dk/bigwig/runwig/`

The Service JVM and Thread Directories

Requesting a URL that denotes a class file defining a JWIG service thread will result in a thread to be created. If a JVM is not already running in that service directory, one is started. Each service directory may contain multiple services located in different class files, although typically, services are installed in distinct directories. One JVM is associated with each service directory. Each service thread owns one subdirectory of the service directory. This subdirectory contains files that are local to the individual thread. The various directories and their corresponding URLs are available in the service code as the following fields:

`servicedir` - the service directory where the JVM resides

`sessiondir` - the local directory for the current session thread

`serviceurl` - the URL (excluding protocol and host) corresponding to `servicedir`

`sessionurl` - the URL (excluding protocol and host) corresponding to `sessiondir`

`serverurl` - the server root URL (protocol and host)

As an example, these fields could contain the following values:

```
servicedir = /home/amoeller/jwig-bin/demo/
sessiondir = /home/amoeller/jwig-bin/demo/03296hulktnnc1/
serviceurl = /jwig-amoeller/demo/
sessionurl =/home/amoeller/jwig-bin/demo/03296hulktnnc1/
serverurl = http://freewig.brics.dk
```

If a file named `jvm.lock` is present in the service directory, new service threads cannot be created. Instead, the clients receive a "503 Service Unavailable" message. This is, for instance, used by the `jwig update` command.

Every `jar` file occurring in the service directory is automatically included in the JVM classpath. This makes it easy to use extra packages in the service code. (Being used in the JWIG runtime system, `jar` files for Xerces, JDOM, and dk.brics.automaton are implicitly included in the classpath.)

In each thread directory, a file named `status` describes the current state of the thread as either running (currently executing), showing (waiting for client response), or terminated, which is used by the garbage collector.

Sessions and Reply Indirection

Other Web service systems that support some variant of session management commonly have some problematic shortcomings: Clicking the "back" button in the browser may lead to an obsolete page, which can confuse or annoy the client; bookmarks cannot be used to temporarily suspend and later resume a session (because selecting such a bookmarked URL would cause a

re-submission); and sessions cannot easily be migrated to another browser (if the session ID is encoded in a cookie, for instance).

The JWIG runtime system provides a unique solution to these problems. Instead of relying on cookies, URL rewriting, or hidden form fields, we associate a unique *session URL* to each session thread. The `sessionurl` field implicitly refers to a file named `index.html`, called the *reply file*, located in the session directory. (The name "`index.html`" depends on the `DirectoryIndex` directive in `httpd.conf`.) This URL functions as the ID of the session. At all times, the reply file contains the newest document produced in that particular session. When proceeding through a session, the client sees the same file again and again, but with different contents. This is implemented using the *moved temporarily* feature of HTTP. When a Web page has been produced for the client and written to the `index.html` file, the server sends a code "302 Moved Temporarily" message to the browser, which then retrieves the file. The overhead of this indirection is negligible, compared to the benefits: The history buffer in the browser is not filled with obsolete URLs, bookmarks can be used without problems, and a session can be moved to another browser just by copying the session URL - and the technique works transparently to the JWIG programmer.

This approach also makes it possible to produce *temporary replies*. If the server is able to predict that it will take a while to produce a response to a request, a message, such as, "please wait, we're working hard to serve your request", is written to the reply file. The file contains a `refresh` instruction (`<meta http-equiv="refresh" content="5" />`), causing the browser to reload the file every 5 seconds until the actual reply is ready. By receiving such temporary replies, the clients less likely become impatient and abandon the service. JWIG produces by default a generic temporary reply if it takes more than 5 seconds to produce the actual reply; the `setTemporaryReply` method allows more specialized messages to be produced.

File Naming Conventions

Extra files can be placed in the service directories, both in the thread directories and in the main service directories. To allow both private and public files to reside side-by-side and prevent clashes with special runtime system files, the following guidelines should be followed:

1. the names of files that are installed together with the service or created by the running service must contain a dot (`.`) or an underscore (_) but cannot end in "`.http`" or "`.fifo`" - this prevents name clashes, and
2. files whose names start with a dot or end with "`.class`", "`.jar`", "`.http`", or "`.lock`" and all files in directories whose names start with a dot are private, that is, they cannot be downloaded.

For example, a file named `style.css` or `help.html` is visible from the Web (unless other access restrictions are applicable), but requesting a file named `.htpasswd` will always result in a "404 Not Found" error.

Specifying HTTP Response for Files

JWIG services may associate special HTTP headers with public files. Using the method `setFileAttributes`, the Content-Type and Content-Encoding of a file can be set. Also, browser caching of a file can be disabled. (Browser caching is automatically disabled for the reply file `index.html`.)

Furthermore, cookies can be associated with a file, such that whenever it is downloaded, one or more cookies are sent along. As described above, cookies are *not* used for session management in JWIG, but they may still be used for other purposes. The method `addCookie` can be used to construct a cookie. Usually, cookies are associated with the `index.html` reply file, such that the client receives the cookies together with the normal server reply.

Environment Variables

The map `env` contains environment variables for the latest client interaction. The following variables are typically set:

`REMOTE_ADDR` - IP number of client

`REMOTE_PORT` - port number of client

`HTTP_USER_AGENT` - browser type

`HTTP_REFERER` - referring page

`HTTPS` - set only if the request was made through SSL

Log Files

The `mod_bigwig` module writes information about its behavior to the Apache Web server error log file (specified by the `ErrorLog` directive in `httpd.conf`).

Additionally, each service directory contains a file named `log` with log information from the Java part of the JWIG runtime system and from the service code. This log is available through the `servicelog` field.

The `LogLevel` set in `httpd.conf` is used both by `mod_bigwig` and `bigwigd` and also as the initial log level of `servicelog` in the running JWIG services. If the log level is set to `DEBUG`, very detailed information is produced.

Suspending a JVM

If a JVM is running in a service directory where there have been no active threads for a certain period of time (by default 10 minutes), the JVM will store its state to disk and terminate itself. When a request is received again, a new JVM will start, restore the state, and handle the

request. This is convenient for servers that host many different services that run simultaneously but perhaps only being active a few times an hour each.

The suspension is transparent to the clients, except that there can be a small delay when restarting the JVM. The JWIG service programmer must ensure that all shared service state is serializable - the next section describes this in more detail.

This suspension feature can be disabled by changing the `BigwigJava` parameter `600` to `0` in `bigwig.conf`.

The Security Manager

In each service directory, the JVM runs in a sandbox to protect the rest of the server. This means that, unless the `jwig.policy` file is modified at installation, JWIG services are restricted in the following ways:

- files outside the service directory cannot be accessed (except the Java and JWIG system files)
- native code cannot be used (again, except the JWIG runtime system)
- the security manager cannot be modified

The JWIG security manager can be globally disabled using the `--disable-secure` configuration option during installation.

Garbage Collection

When the Apache Web server is running with the `mod_bigwig` module, the `bigwigd` daemon is automatically started. This daemon periodically looks in all directories where JWIG services have been installed and removes the thread directories that are no longer in use, either because the threads are finished or because they have been abandoned by the clients. A thread directory is kept for at least the number of seconds specified by the `terminated_timeout` and `show_timeout` fields (default: 600 seconds).

More information about `runwig` can be found in the manual pages for `mod_bigwig` and `bigwigd`.

8 Serialization of Shared Data

All fields in the service class are automatically *shared* between all running threads in the service. This provides a simple alternative to full-scale databases for non-critical data. Shared data fields are read and written just as any other variable, and synchronization and other concurrency control issues can be handled by the standard Java mechanisms, such as `synchronized` methods and statement blocks.

Example:

```
public class Game extends Service {
    protected int visitors = 0;

    synchronized int hit() {
        return ++visitors;
    }
    ...
}
```

In this example, a shared field `visitors` is declared. Only one instance of this field exists for the entire service. The method `hit`, which writes to the field, is `synchronized` to avoid race conditions with other concurrently executing threads.

For critical data, external databases can be used, for instance via JDBC[4], as in any other Java program. However, much data in typical Web services does not require the high performance and stability of a large database, and the extra complexity of the service code for building database requests can be significant.

Transactions and Serialization

To increase robustness of the services, JWIG contains a primitive transaction system, which allows the shared state to be stored to disk at well-defined places during the execution. The default mechanism is based on Java's built-in serialization mechanism.

Using the `checkpoint` and `rollback` methods, the programmer can decide when to take a snapshot of the shared state and store it on disk and also to restore the state as it was at the last snapshot:

class `dk.brics.jwig.runwig.Service`

`public static void` **`checkpoint`**`()`
`throws java.io.IOException`

Serializes all shared service data and stores it in jvm.state.

Throws:
`java.io.IOException` - if an I/O error occurred

[4] `http://java.sun.com/products/jdbc/`

class `dk.brics.jwig.runwig.Service`

`public static void` **`rollback`**`()`
`throws java.io.IOException,`
`java.lang.ClassNotFoundException`

Restores all shared service data from jvm.state.

Throws:
`java.io.IOException` - if an I/O error occurred
`java.lang.ClassNotFoundException` - if the class of a serialized object cannot be found

This approach requires all shared state to be declared as `Serializable`. All data that is transitively reachable from the service object is stored, except for transient and static fields.

The `checkpoint` method should be invoked only at times when the shared state is consistent. The runtime system ensures that the checkpoint is performed atomically. If the JVM is suspended, as described in the previous section, a `checkpoint` is automatically performed, and when it resumes, a `rollback` is performed. Also, if the server or the JVM should crash, a `rollback` is automatically performed when the server is up again and a new thread is started by a client.

Serialization using XML

As an alternative to the standard serialization mechanism described above, JWIG also allows the shared state to be stored using an XML representation. Using such a representation has the benefit that other tools straightforwardly can read and modify the data, if the need should arise. A service class which implements the XMLSerializable interface is serialized and unserialized using the methods `toXML` and `fromXML`, which the programmer must implement. The XML documents are represented using JDOM[5] in the program code:

class `dk.brics.jwig.runwig.XMLSerializable`

`public org.jdom.Element` **`toXML`**`()`

Creates XML representation of this object. References to objects must be followed manually.

Returns:
JDOM XML tree

[5] `http://www.jdom.org/`

class `dk.brics.jwig.runwig.XMLSerializable`

`public void` **`fromXML`**`(org.jdom.Element e)`

Restores the state of this object according to the given XML representation.

Parameters:
`e` - JDOM XML tree

Example:

```
public class MyService extends Service implements XMLSerializable {
    class Person {
        String login;
        String password;
        String name;

        Person(String login, String password, String name) {
            this.login = login;
            this.password = password;
            this.name = name;
        }
    }

    HashMap people = new HashMap();

    public Element toXML() {
        Element e = new Element("people");
        Iterator i = people.values().iterator();
        while (i.hasNext()) {
            Person p = (Person) i.next();
            Element f = new Element("person").
                addContent(new Element("login").addContent(p.login)).
                addContent(new Element("password").addContent(p.password)).
                addContent(new Element("name").addContent(p.name));
            e.addContent(f);
        }
        return e;
    }

    public void fromXML(Element e) {
        people = new HashMap();
        Iterator i = e.getChildren().iterator();
        while (i.hasNext()) {
            Element f = (Element) i.next();
            Person p = new Person(f.getChildText("login"),
                                  f.getChildText("password"),
                                  f.getChildText("name"));
```

```
            people.put(p.login, p);
        }
    }

    synchronized void addPerson(String login, String password, String name) {
        people.put(login, new Person(login, password, name));
        log(Log.INFO, "added user: "+login);
        try {
            checkpoint();
        } catch (IOException e) {
            log(Log.ERR, "unable to checkpoint");
        }
    }
    ...
}
```

In this example, the `people` field contains a set of `Person` objects, which are shared between all service threads. The `toXML` methods constructs an XML tree containing all the information from the `people` set, and `fromXML` replaces the current shared state by the data in the given XML tree. Also note that the `addPerson` method is `synchronized` to avoid concurrency problems.

At runtime, the serialized state is stored in a file named `jvm.state.`*`CLASS`* located in the service directory, where *`CLASS`* is an ASCII encoding of the service class name.

9 Updating a Running Service

The `jwig` tool has special support for updating running services. This is a nontrivial task for several reasons:

- The state defined by the class files may have changed. Classes and fields may have been added, modified, or removed by the update, and the existing state should not be corrupted.
- Threads may be running at the same time as the update is performed, either executing service code or showing documents and waiting for client response. Again, to avoid corruption of shared state, old and new code should never be run simultaneously.

The command

```
jwig update dir files
```

ensures that the service files are updated atomically and only at well-defined places in the thread execution. The arguments are as for the `jwig install` command.

Before the update is performed, the tool checks the serialVersionUID for each Serializable class in the service. If this number have changed for one or more files, the update is aborted to avoid unserialization errors and corruption of the service state. See Sun's serialization guide[6] for a description of using serialization with evolution.

The update is performed as follows: First, the service directory is locked (using `jvm.lock`) such that new requests to start service threads are blocked ("503 Service Unavailable") Then, the tool waits until there are no running threads, that is, they are all either terminated or waiting for client response. All waiting threads are then terminated. (This should give a well-defined behavior since clients may never respond anyway; the clients that run these threads are met with a "session has terminated" error). The files are then copied to the service directory, overwriting the existing files. Finally, the service lock is removed to accept new requests.

10 PowerForms - Declarative Form Input Validation

JWIG incorporates the PowerForms language for making validation of form input easier. Often with traditional programming languages, substantial amounts of the service source code is used for checking that the users have filled out the forms correctly, but those languages provide no particular support for this aspect of Web service development. To provide immediate and user friendly feedback to the user, client-side JavaScript is typically applied. However, since JavaScript execution can be bypassed, a server-side double check is always necessary. This means that, in addition to being required to master JavaScript - which can be surprisingly difficult because of the many different variants that the browsers understand - the Web service programmers must essentially write the same code twice, first in JavaScript for the user friendly client-side validation, and then in a different language for the double check on the server.

PowerForms is a small domain-specific language for declarative specification of form input validity requirements. Using an XML notation, *formats* and help messages can be specified for individual fields. A format is essentially a regular expression defining a set of valid values for the field. Additionally, complex interdependencies between different fields can be specified, such that the format of one field may depend on the values of other fields. A *PowerForms document* concisely specifies validity requirements for one or more whole forms that appear in an XHTML document that is shown to the user. Given such an XHTML document and a PowerForms document, JWIG automatically inserts JavaScript code into the XHTML document before being shown, such that form input validation is performed incrementally as the user fills out the form. Furthermore, code for performing the server side double check is also automatically generated. With PowerForms, JWIG programmers can easily add advanced form input validation to a Web service - without writing a single line of JavaScript code.

A PowerForms document is an `XML` object. As for XHTML documents, it can be built using gaps and plug operations. A variant of the `show` operation takes a PowerForms document as an extra argument P:

```
show D powerforms P;
```

[6] `http://java.sun.com/j2se/1.4/docs/guide/serialization/`

JWIG guarantees that execution will not continue after this operation unless all specified form input requirements are satisfied.

Example:

```
import dk.brics.jwig.runtime.*;

public class PowerFreebie extends Service {

    public class HowMany extends Session {

static final int MAX = 5;

XML templateAsk = [[
             <html><head><title>PowerFreebie</title></head><body><form>
               How many free T-shirts do you want?
               <input name="amount" type="text"/>
               <input name="continue" type="submit"/>
             </form></body></html>
        ]];

        XML templateReply = [[
             <html><head><title>PowerFreebie</title></head><body>
               You will receive <[amount]> k00l T-shirts any day now...
             </body></html>
        ]];

        XML format = [[
             <powerforms xmlns="http://www.brics.dk/powerforms/2.0">
               <constraint field="amount">
                 <match>
                   <interval low="1" high=[high]/>
                 </match>
               </constraint>
             </powerforms>
        ]];

public void main() {
    show templateAsk powerforms format<[high=MAX];
    int amount = Integer.parseInt(receive amount);
    exit templateReply<[amount=amount];
}
    }
}
```

With this Web service, users can order a number of T-shirts, but the PowerForms document specifies that at most `MAX` can be requested. If the user's browser supports JavaScript, an error window will pop up if the user attempts to order too many. Such violations are caught on the server if the JavaScript code is somehow bypassed. Note that the construction of the XHTML

documents is not affected by the introduction of form input validation. Thus, PowerForms can easily be added gradually to a JWIG service.

The JWIG program analyzer can be used to check that the PowerForms documents are always valid - even though they are dynamically generated as in the example above.

See the PowerForms section of the online tutorial *Interactive Web Services with Java* [6] for further description of PowerForms. The full grammar for the PowerForms language is available from the PowerForms home page[7]. The PowerForms language was introduced in the research paper *PowerForms: Declarative Client-side Form Field Validation* [2] in the context of the `<bigwig>` language.

11 SSL Encryption and HTTP Authentication

To ensure authentication and confidentiality of the communication between the server and the clients, JWIG supports HTTP Authentication and SSL[8] (Secure Sockets Layer).

The JWIG distribution contains a simple example service `Authentication.jwig`, which uses both SSL and HTTP Authentication.

HTTP Authentication

The methods listed below define security requirements, either *locally* for the current thread or *globally* for the entire service.

The `makeUserFile` method can be used to create a file containing usernames and passwords:

[7] `http://www.brics.dk/~ricky/powerforms/`
[8] `http://www.netscape.com/eng/ssl3/`

class dk.brics.jwig.runwig.Service.Session

```
public void makeUserFile(boolean local,
                         java.lang.String userfile,
                         java.util.List usernames,
                         java.util.List passwords,
                         boolean encrypt)
                  throws java.io.IOException
```

Writes a ".htpasswd" file.

Parameters:
- `local` - if true, write to local thread directory - if false, write to shared service directory (ignore if userfile is non-null)
- `userfile` - file name for authorized names/passwords - must be absolute (non-relative) path (if null, use .htpasswd)
- `usernames` - list of user names
- `passwords` - list of passwords
- `encrypt` - if true, passwords are encrypted

Throws:
- `java.io.IOException` - if I/O error occurs

The `setAccessControl` method can be used to write the "`.htaccess`" file for enabling client authentication and also for requiring SSL encryption (the name "`.htaccess`" depends on the `AccessFileName` directive in `httpd.conf`):

class dk.brics.jwig.runwig.Service.Session

```
public void setAccessControl(boolean local,
                             java.lang.String userfile,
                             java.lang.String realm,
                             boolean require_ssl)
                      throws java.io.IOException
```

Writes a ".htaccess" file. This file defines authentication requirements for client access.

Parameters:
- `local` - if true, write to local thread directory - if false, write to shared service directory
- `userfile` - file name for authorized names/passwords - must be created before, and must be absolute (non-relative) path (if null, use .htpasswd)
- `realm` - realm name (if null, no authentication check)
- `require_ssl` - if true, set SSLRequireSSL

Throws:
- `java.io.IOException` - if I/O error occurs

The `removeAccessControl` method removes the ".htaccess" file to disable authentication and SSL requirements:

class `dk.brics.jwig.runwig.Service.Session`

`public boolean` **`removeAccessControl`**`(boolean local)`

Removes ".htaccess" file.

Parameters:
`local` - if true, remove from local thread directory - if false, remove from shared service directory
Returns:
true if file successfully deleted.

SSL Encryption

To use SSL, mod_ssl[9] must be installed in your Apache Web server, together with a server certificate. The configuration of mod_ssl is managed through Apache - independently of the JWIG system.

The `HTTPS` environment variable (available in the `env` map) is set if the last interaction was made through SSL.

The `enableAccessControl` method described above should always be used to set the `SSLRequireSSL` flag such that SSL cannot be bypassed.

The `enableSSL` method can be used to change the HTTP protocol to `https` in subsequently generated URLs:

class `dk.brics.jwig.runwig.Service.ServerThread`

`public void` **`enableSSL`**`()`

Enables SSL. The protocol part of serverurl is set to https. This assumes that the standard ports are used (80 for http and 443 for https).

The `disableSSL` method can be used to change the HTTP protocol to `http` in subsequently generated URLs:

class `dk.brics.jwig.runwig.Service.ServerThread`

`public void` **`disableSSL`**`()`

Disables SSL. The protocol part of serverurl is set to http. This assumes that the standard ports are used (80 for http and 443 for https).

[9] `http://www.modssl.org/`

If more advanced control over SSL or HTTP Authentication is required, the Apache Web server configuration files, in particular `.htaccess`, should be written manually.

12 Sending Emails

The JWIG API contains a simple technique for sending emails from the server:

class `dk.brics.jwig.runwig.Service`

```
public boolean sendMail(java.lang.String to,
                        java.util.Map header,
                        java.lang.String body,
                        java.lang.String enc)
```

Sends email.

Parameters:
- `to` - receiver email address
- `header` - map from String to String containing header fields
- `body` - email contents
- `enc` - encoding (e.g. iso-8859-1)

Returns:
- false if an error occured

Example:

```
HashMap header = new HashMap();
header.put("MIME-Version", "1.0");
header.put("Content-Transfer-Encoding", "8bit");
header.put("Content-Type", "text/plain; charset=iso-8859-1");
header.put("To", p.email);
header.put("Subject", "JWIG password");
header.put("From", root.email);
header.put("Reply-To", root.email);
header.put("Errors-To", root.email);
String body =
    "Your login for the JWIG meta-service is: "+p.login+"\n"+
    "and the password is: "+p.password+"\n\n"+
    "Go to "+getURL(Run.class)+" to log in.";
boolean ok = sendMail(p.email, header, body, "iso-8859-1");
```

With the default installation configuration, the `sendmail` program with the options `-n -i -t` is used to send the email. This can be changed during installation of the JWIG system using the `--with-sendmail` and `--with-sendmail-args` configuration options.

13 API - Overview of Classes, Methods, and Fields

The JWIG Application Programming Interface provides functionality for defining Web services.

In addition to this API, JWIG adds a set of language constructs to the Java language for constructing and showing XML documents and receiving form input.

A brief overview of the central parts of the API:

- `abstract class dk.brics.jwig.runtime.Service`
  - base class for JWIG services
  - `void setOutputEncoding`
    - sets output encoding (default: ISO-8859-1)
  - `Log servicelog`
    - service log file
  - `void log(int level, String msg)`
    - writes message to service log
  - `String formatHTTPTime(Date d)`
    - formats Date according to RFC 822/1123
  - `String urlEncode(String s)`
    - URL encodes string using UTF-8
  - `String urlDecode(String s)`
    - URL decodes string in UTF-8
  - `String getURL(Class c)`
    - returns URL for creating a thread in the current service
  - `void checkpoint()`
    - writes shared service data to disk
  - `void rollback()`
    - restores shared service data from disk
  - `boolean setExcludeLock()`
    - locks service for new requests
  - `void releaseExcludeLock()`
    - removes service lock
  - `boolean sendMail(String to, Map header, String body, String enc)`
    - sends an email
  - `abstract class ServerThread`
    - base class for all server threads
    - `List arglist`
      - interaction field arguments (list of `Arg`) in order of occurence
    - `Map argmap`
      - interaction field arguments (map from `String` to list of `Arg`)

- `Map env`
  - interaction environment (map from `String` to `String`)
- `Map cookies`
  - incoming cookies (map from `String` to `String`)
- `String out_cookies`
  - raw string containing outgoing cookies
- `int reply_timeout`
  - time-out (seconds) for waiting for service reply and refresh interval
- `int show_timeout`
  - time-out at 'show'
- `int terminated_timeout`
  - time-out for garbage collecting completed sessions
- `String serverdir`
  - Apache/mod_bigwig server home directory
- `String servicedir`
  - service home directory
- `String sessiondir`
  - session home directory
- `String serverurl`
  - server URL (protocol and host)
- `String serviceurl`
  - URL to service home directory (excluding protocol and host)
- `String sessionurl`
  - URL to session home directory (excluding protocol and host)
- `String intkey`
  - interaction access key
- `boolean validate_xhtml`
  - perform runtime XHTML validation if true (default: false)
- `void addCookie(String name, String value, String expires, String path, String domain, boolean secure)`
  - adds cookie to `out_cookies`
- `void enableSSL()`
  - sets protocol to HTTPS
- `void disableSSL()`
  - sets protocol to HTTP
- `void setFileAttributes(boolean local, boolean remove, String path, String mime, String encoding, boolean nocache, boolean sendcookies)`
  - writes "`.http`" attribute file

- `abstract class Session`
  - base class for session threads, extends `ServerThread`
  - `abstract void main()`
    - main method

    - `String continueURL()`
      - returns URL for continuing the current session
    - `void setTemporaryReply(XML doc)`
      - writes temporary reply page
    - `void makeUserFile(boolean local, String userfile, List usernames)`
      - writes `.htpasswd` file for HTTP Authentication
    - `void setAccessControl(boolean local, String userfile, String realm, boolean require_ssl)`
      - writes `.htaccess` file for HTTP Authentication
    - `boolean removeAccessControl(boolean local)`
      - removes `.htaccess` file
  - `abstract class Page`
    - base class for page threads, extends `ServerThread`
    - `abstract void main()`
      - main method
  - `abstract class Seslet`
    - base class for seslet threads, extends `ServerThread`
    - `abstract void main(InputStream in, OutputStream out)`
      - main method

- `class dk.brics.jwig.runwig.Log`
  - logging facility
  - `void setLogLevel(int level)`
    - sets logging verbosity level
  - `void setStream(PrintStream s)`
    - changes print stream
  - `void write(int level, String msg)`
    - writes message to log
  - `void write(String msg, Throwable e)`
    - writes `Throwable` message to log

- `class dk.brics.jwig.runwig.Arg`
  - HTTP input field, used for file uploads
  - `String name`
    - field name
  - `String value`
    - field value
  - `String filename`
    - filename attribute, if file upload
  - `String contenttype`
    - contenttype attribute, if file upload

  - `String charset`
    - charset attribute, if file upload

- `interface dk.brics.jwig.runwig.XMLSerializable`
  - for XML serialization of shared data

  - `Element toXML()`
    - creates JDOM XML representation of the shared state
  - `void fromXML(Element e)`
    - restores shared state from JDOM XML representation

The full javadoc API specification for the BRICS JWIG implementation is available online[10]. (Note, however, that the javadoc currently exposes many low-level methods and fields that should not be exploited directly - in a future version, we will hide those parts.)

References

[1] K. ARNOLD, J. GOSLING, AND D. HOLMES, *The Java Programming Language*, Addison-Wesley, 3rd ed., June 2000.

[2] C. BRABRAND, A. MØLLER, M. RICKY, AND M. I. SCHWARTZBACH, *PowerForms: Declarative client-side form field validation*, World Wide Web Journal, 3 (2000), pp. 205–314. Kluwer.

[3] C. BRABRAND, A. MØLLER, A. SANDHOLM, AND M. I. SCHWARTZBACH, *A runtime system for interactive Web services*, Computer Networks, 31 (1999), pp. 1391–1401. Elsevier. Also in Proc. 8th International World Wide Web Conference, WWW8.

[4] A. S. CHRISTENSEN, A. MØLLER, AND M. I. SCHWARTZBACH, *Extending Java for high-level Web service construction*, ACM Transactions on Programming Languages and Systems, 25 (2003), pp. 814–875.

[5] A. S. CHRISTENSEN, A. MØLLER, AND M. I. SCHWARTZBACH, *Precise analysis of string expressions*, in Proc. 10th International Static Analysis Symposium, SAS '03, vol. 2694 of LNCS, Springer-Verlag, June 2003, pp. 1–18.

[6] A. MØLLER AND M. I. SCHWARTZBACH, *Interactive Web services with Java*, April 2002. BRICS, Department of Computer Science, University of Aarhus, Notes Series NS-02-1. Available from `http://www.brics.dk/~amoeller/WWW/`.

[7] S. PEMBERTON ET AL., *XHTML 1.0: The extensible hypertext markup language*, January 2000. W3C Recommendation. `http://www.w3.org/TR/xhtml1`.

[10] `http://www.brics.dk/JWIG/doc/`