Sleep Developer's Guide

Part 0: About Sleep

This document is a guide for developers who want to integrate sleep into their application. Sleep is an embeddable scripting solution for Java applications. The language is heavily inspired by Perl with bits of Objective-C thrown in. The Sleep Java API allows the language to be extended with new constructs, operators, functions, and variable containers.

Sleep came from an inspired weekend of coding in April 2002. Since then Sleep has been developed in parallel with the Java IRC Client, jIRCii.

Sleep Language Features

A Perl inspired language

The Sleep language is very heavily inspired by Perl. The syntax and API are easy to pickup for novice programmers and familiar to those with Perl scripting experience. Scripters do not need to know Java to effectively extend an application that embeds Sleep.

Built-in Regular Expression support

Regular expressions are a powerful feature for parsing and extracting information from strings. Sleep provides full support for regular expressions built right into the language. Sleep also includes a transliteration function. These two features are considered by some to be the meat and potatoes of Perl scripting.

Powerful unified I/O API for scripters

The Sleep language includes a powerful unified I/O API for reading and writing from/to sockets, processes, and files all using the same interface. Sleep also includes powerful functionality for manipulating and extracting binary data.

Access, create, and query Java objects from Sleep scripts

Sleep 2.0 includes the HOES subsystem which provides the ability to access, instantiate, and manipulate Java objects. This means Sleep can interface directly with any Java API or API present in your application. This ability makes Sleep a very powerful glue language for the Java platform.

Flexible Datastructures

Sleep has three primary data structures built right into the language. Scalars are general purpose variables that can hold any type of data including ints, doubles, longs, strings, and even object references. The Sleep array data type can act as a stack and a list as well. Sleep includes extensive functionality for manipulating and sorting arrays. Sleep hashes are easy to use dictionary data structures that allow any key to map directly to a value. As an added bonus hashes and arrays can easily be combined into multidimensional data structures.

Closures and Pseudo Objects

Sleep closures are essentially blocks of code that contain their own lexical scope. These blocks of code can be passed as parameters, assigned to variables, and invoked. Within Sleep closures can be used to create pseudo object interfaces. Sleep closures can even be passed off as fake Java objects using the HOES subsystem. Closures bring to Sleep a method for organizing data, a way to implement Java interfaces, and they enable a simple functional programming within Sleep scripts.

The above is just a sampling of what the Sleep language has to offer. For more information read the Sleep Language Fundamentals document.

Bridges: Connecting Your Application to Sleep

A proper Sleep based solution brings more than syntactic sugar to an application. Sleep bridges allow you to provide a higher level and easier to use abstraction of existing APIs. Bridges are APIs for adding basic language elements to Sleep. By providing logical operations, predicates, functions, and constructs that map to an API, the Sleep language can be adapted to your application or to problem domains within your application.

You can easily add built in functions just by implementing a java interface. Classes implementing the function interface are known as function bridges. Function bridges can take arguments from sleep scripts and can return values as well.

Data in your application can be made available to scripters. Sleep provides a number of utility methods for wrapping your application data into scalar variables that user scripts can work with.

It is also possible to create built-in variables. This is accomplished with the Variable interface. Variable bridges let you create built-in variables that execute a java function and return a value whenever they are accessed.

Adding new comparison operators for use in if statements and loop constructs is easy as well. By implementing a java interface you can add new predicates to the sleep language. Operators for use within expressions can be added as well using the Operator interface.

Sleep also lets you build environment bridges. An environment bridge is a defined keyword that is associated with user script code. Whenever a defined keyword is encountered your own code can be called to handle it. You can use environment bridges to easily create event listeners, syntax for popup menus, or anything else you can imagine. Environment bridges offer a lot of power and flexibility towards integrating sleep in an easy to use way for your end-user scripters.

Most of sleeps basic language features were implemented using the same API's available to you. The package sleep.bridges.* is an immediate source code reference for developing useful sleep bridges.

Part 1: Introduction - The Most Basic Stuff

This introduction will serve as a basic overview for making your application scriptable. When making an application scriptable there are a few fundamental things you will want to know. These fundamentals include loading a script, adding your own built-in functions, reading and writing variables from/to the script environment, and calling functions. This introduction will cover how to do all of these things mostly with examples. A complete example based off of Part 1 is available in Appendix A.

Load a Script

To load a script from a file and run it:

```
ScriptLoader   loader = new ScriptLoader();
ScriptInstance script = loader.loadScript("script.sl");

script.runScript();
```

The above will load the file script.sl and then execute it immediately.

Add a Built-In Function

Just loading and executing a script isn't of much use. You want to make it so the script can interact with your application. One way to do this is to add functions that

scripters can take advantage of. An example of a basic function bridge is below:

```
public class FooFunction implements Function
{
   public Scalar evaluate(String name, ScriptInstance script, Stack args)
   {
      System.out.println("function foo has been called");
      return SleepUtils.getEmptyScalar();
   }
}
```

The above class implements the Function interface. The Function interface is used for creating built in functions. All functions available to scripters come from a class implementing the Function interface.

To use the above Function it must be installed into the script environment. To install a Function class into the script environment use:

```
script.getScriptEnvironment().getEnvironment().put("&foo", new FooFunction());
```

The above code puts an instance of FooFunction into the environment as the function named *&foo*. This means the ScriptInstance represented by the script variable will be able to call `foo()`. When `foo()` is called by a scripter the evaluate method in the FooFunction instance will be called.

The evaluate method is expected to return a Scalar object. If a function isn't returning anything then return the value returned by `SleepUtils.getEmptyScalar()`. The empty scalar is the sleep equivalent of null, undef, and nil in other languages. Sleep uses *$null* to represent the empty scalar in scripts.

To work with arguments passed to a built-in function:

```
public class MyAddFunction implements Function
{
   public Scalar evaluate(String name, ScriptInstance script, Stack args)
   {
      int arg1 = BridgeUtilities.getInt(args, 0);
      int arg2 = BridgeUtilities.getInt(args, 0);

      return SleepUtils.getScalar(arg1 + arg2);
   }
}
```

The function above takes two arguments. The arguments are passed in as a Stack object. The BridgeUtilities class contains methods for safely extracting parameters from the arguments stack. In the above example two integer parameters are extracted. The BridgeUtilities class allows a default value to be specified in case the stack is empty (i.e. not enough parameters were passed). I recommend you use the BridgeUtilities class.

The last part of the above function is the return statement. Notice that the result of adding arg1 and arg2 is passed to `SleepUtils.getScalar()`. The SleepUtils class contains static methods for converting just about any type you can think of into a scalar usable by sleep scripts.

To install the MyAddFunction into a script environment:

```
script.getScriptEnvironment().getEnvironment().put("&myadd", new MyAddFunction());
```

The ScriptInstance represented by the script variable will now be able to call `myadd(3, 3)` to add two numbers together.

What do you need to know to add scripting to your application? At the most fundamental level a scriptable app needs a way of loading scripts, unloading scripts, and running a code snippet (from a file or some other place).

So far we've covered how to load a script and add built in functions. Next we'll discover how to make your applications data structures available to sleep scripts.

A Little About Scalars

Scalars are the fundamental data type in sleep. They are represented with the Scalar class. Scalars can contain an int value, long value,.double value, String value, and even an object reference.

The SleepUtils class contains static methods for easily wrapping data into scalars. For any type a Scalar can represent the SleepUtils class contains a getScalar(type) method that will return the appropriate scalar.

Reading and Writing Scalars

To write a value into the script environment:

```
script.getScriptVariables().putScalar("$test", SleepUtils.getScalar("sleep example"));
```

To read a variable from the script environment:

```
Scalar test  = script.getScriptVariables().getScalar("$test");
```

The first snippet puts a string scalar into the script environment with the name *$test*. The second snippet reads the scalar assigned to *$test* from the script environment.

Calling a Function

To call a function in a script:

```
Scalar value = script.callFunction("&konk", new Stack());
```

The above calls the function &konk from the ScriptInstance object contained by script. The second parameter to callFunction is a stack of arguments. Arguments are contained in the stack in a first in, last out fashion. Meaning the first value passed to the function should be the top item on the Stack.

A Better Way to Install Functions

Remembering to install every built-in function into every can be cumbersome. Sleep provides a Loadable bridge interface and the ScriptLoader class to help make this process easier.

To create a Loadable bridge that installs our two functions into a script environment:

```
public class SetupBridge implements Loadable
{
   public boolean scriptLoaded(ScriptInstance script)
   {
      Hashtable env = script.getScriptEnvironment().getEnvironment();

      env.put("&foo",   new FooFunction());
      env.put("&myadd", new MyAddFunction());

      return true;
   }

   public boolean scriptUnloaded(ScriptInstance si)
   {
      return true;
   }
}
```

By itself the above doesn't do much. However in conjunction with a ScriptLoader object the process of setting up each ScriptInstance when a script is loaded or unloaded is made much easier.

To create a script loader, install a loadable bridge, and load a script:

```
ScriptLoader   loader = new ScriptLoader();
loader.addSpecificBridge(new SetupBridge());

ScriptInstance script = loader.loadScript("script.sl");

script.runScript();
```

The above installs an instance of our loadable bridge, SetupBridge into the script loader loader. When script.sl is loaded via the loadScript method of loader the scriptLoaded method in SetupBridge will be called.

Conclusion

This section presented some of the basic concepts for interfacing with a scripting language. The code presented in this section is presented in Appendix A. Knowing how to load a script, add built in functions, read/write variables, and call functions are the most important concepts for interfacing an application with a scripting language.

We covered Loadable and Function bridges in this section. In the next part of this guide we will cover all of the bridging options in sleep.

Part 2: Bridge Writers Guide

The Bridge Architecture

Each loaded script has an associated environment with it. When a script attempts to call a function, check a condition, reference a variable etc. it refers to its environment for the appropriate bridge. If the bridge happens to be an application bridge, the application bridge will call an API in the application, any values returned by the application will be sent back to the script. Figure 1 highlights what the Sleep bridge architecture looks like.

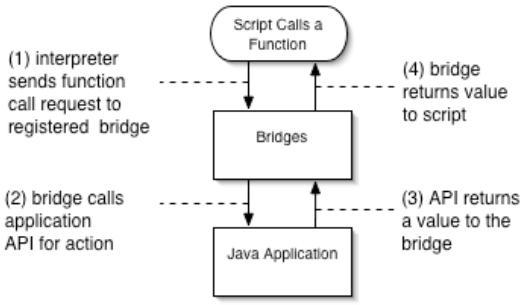

Figure 1. Sleep Bridge Architecture

Loadable Bridges

A loadable bridge is used to perform actions on scripts when they are loaded and unloaded. Loadable bridges by themselves do not add anything to the sleep language at all. In conjunction with a ScriptLoader loadable bridges make it easy to process the environment of scripts as they are loaded and unloaded.

A sleep loadable bridge is created through implementing the sleep.interfaces.Loadable interface. The loadable interface is as follows:

public boolean scriptLoaded (ScriptInstance script)

Called when a script is loaded. The loaded ScriptInstance is passed to this method. The script loading function can be used to install things to the environment (i.e. new Functions, Predicates etc). The return value should be true if the environment was loaded successfully.

public boolean scriptUnloaded (ScriptInstance script)

Called when a script is unloaded. The loaded ScriptInstance is passed to this method. The script loading function can be used to clean up script specific resources when the script is unloaded. The return value should be true if this bridge was unloaded successfully.

A loadable bridge is installed into the language by adding it to a script loader class. According to the ScriptLoader class there are two types of bridges. The two types are specific and global bridges.

A specific bridge is executed for each and every script on load, no matter what.

A global bridge is executed once for each script environment. If scripts are sharing an environment there is no sense in loading stuff into the environment more than once. This is why global bridges exist.

An example of adding a loadable bridge to a script loader:

```
ScriptLoader loader = new ScriptLoader()
loader.addSpecificBridge(new MyLoadableBridge());
```

Loadable bridges ARE used to install other bridges into a script environment. Using a loadable bridge is the easiest way to make sure actions are always performed on a script as it loads. Loadable bridges in conjunction with a script loader are used to perform cleanup actions when a script is unloaded.

Function Bridges

A function bridge is used to define a built-in function. Once a function bridge is installed into the script environment, it can be called from user created scripts.

A function bridge is created through implementing the interface sleep.interfaces.Function. The Function interface is as follows:

public Scalar evaluate(String name, ScriptInstance instance, Stack locals)

Called when a function of the specified name is to be evaluated. The script instance of the calling script is passed in. The java.util.Stack object contains the arguments passed to the function. The arguments are in the locals stack in the order they were passed in. The name for a function will always be prefixed with the & ampersand symbol. A Scalar value is returned by this interface. The returned value is passed back to the calling script.

To install a function into a script environment:

```
ScriptInstance script;           // assume
Function       myFunctionBridge; // assume

Hashtable environment = script.getScriptEnvironment().getEnvironment();
environment.put("&functionName", myFunctionBridge);
```

In the above code snippet the script environment is extracted from the ScriptInstance object script. The function name is the key with the instance of our Function bridge as the value. The function name must begin with & ampersand for sleep to know it is a function.

Function bridges are incredibly useful for making application functionality available in sleep.

Predicate Bridges

A predicate is an operator used inside of comparisons. Comparisons are used in if statements and loop constructs. Sleep supports two types of predicates. A unary predicate which takes one argument. The other type is a binary (normal) predicate which takes two arguments. In the example comparison a == b, a is the left hand

side, b is the right hand side, and == is the predicate. Predicate bridges are used to add new predicates to the language.

A predicate is created by implementing the interface sleep.interfaces.Predicate.

public boolean decide(String name, ScriptInstance instance, Stack locals)

The above function is called whenever the predicate name is called. The calling script instance is passed to the function. The java.util.Stack of locals contains the arguments passed into the predicate. The arguments are passed in reverse order. For example in a binary predicate the left hand side is the last item on the stack and the right hand side is the first item on the stack. The decide function returns a boolean value depending on the outcome of the predicate.

To install a predicate into a script environment:

```
ScriptInstance script;           // assume
Predicate      myPredicateBridge; // assume

Hashtable environment = script.getScriptEnvironment().getEnvironment();
environment.put("isPredicate", myPredicateBridge);
```

In the above code snippet the script environment is extracted from the script instance class.

A binary predicate can have any name. A unary predicate always begins with the - minus symbol. "isin" would be considered a binary predicate where as "-isletter" would be considered a unary predicate.

Additional built-in predicate keywords should be registered with the script parser before any scripts are loaded. This can be accomplished as follows:

```
ParserConfig.addKeyword("predicate");
```

Predicates are useful for implementing boolean functions.

Operator Bridges

An operator in sleep parlance is anything used to operate on two variables inside of an expression. For example 2 + 3 is the expression add 2 and 3. The + plus sign is the operator.

Creating an Operator class and installing it into the environment makes the operator available for use within expressions.

An operator is created by implementing the interface sleep.interfaces.Operator:

public Scalar operate(String name, ScriptInstance instance, Stack locals)

The operate function is called whenever operator name is to be applied. The calling script instance is passed to the method. The Stack locals contains the left hand side value and right hand side value of the expression in that order.

To install an operator into a script environment:

```
ScriptInstance script;           // assume
Operator       myOperatorBridge; // assume

Hashtable environment = script.getScriptEnvironment().getEnvironment();
environment.put("operator", myOperatorBridge);
```

New built-in operators should be registered with the script parser before any scripts are loaded. This can be accomplished as follows:

```
ParserConfig.addKeyword("operator");
```

The keyword registering practice is in place to clear up ambiguity when parsing scripts. Sleep when parsing scripts does not know what operators, functions, keywords it has. If you create an operator that follows the same naming rules as a function name, sleep might confuse left_hand_side operator (expression) for being a function call. This is due to operator (expression) looking the same as function (expression) to the parser.

Operator bridges probably won't be as common as other bridges. Operator bridges can be used for adding new math operators or new string manipulation operators.

Evaluation Bridge

A Sleep evaluation is a way to define how a `back quoted` string should work. In Perl any text inside of `back quotes` is fevaluated for embedded $scalar values and then executed as a shell command. The output of the executed command is collected into a perl array and returned as the resulting value of the `back quote` expression.

While executing commands in this way might be a useful abstraction, it seems more fun to allow application developer's to define what this syntax should do.

The sleep.interfaces.Evaluation interface consists of a single method:

public Scalar evaluateString(ScriptInstance script, String value)

Evaluates the specified string value

The following is an implementation of perl-like backquote behavior for Sleep:

```
import sleep.interfaces.Evaluation;

import sleep.runtime.Scalar;
```

```
import sleep.runtime.ScriptInstance;
import sleep.runtime.SleepUtils;

import java.io.*;

public class PerlLike implements Evaluation
{
   public Scalar evaluateString(ScriptInstance script, String value)
   {
      Scalar rv = SleepUtils.getArrayScalar();

      try
      {
         // execute our process and setup a reader for it

         Process proc  = Runtime.getRuntime().exec(value);
         BufferedReader reader = new BufferedReader(new InputStreamReader(proc.getInputStream()));

         // read each line from the process output, stuff it into our scalar array rv

         String text = null;
         while ((text = reader.readLine()) != null)
         {
            rv.getArray().push(SleepUtils.getScalar(text));
         }
      }
      catch (IOException ex)
      {
         script.getScriptEnvironment().flagError(ex.toString());
      }

      return rv;
   }
}
```

To install the perl-like backquote evaluator into the script environment:

```
public boolean scriptLoaded(ScriptInstance script)
{
   Evaluation perlStuff = new PerlLike();

   Hashtable environment = script.getScriptEnvironment().getEnvironment();
   environment.put("%BACKQUOTE%", perlStuff);

   return true;
}
```

The above code enables the following type of script to work:

```
@files = `ls -alh`; # execute the UNIX ls command

println("The files are:");
printAll(@files);
```

If no Evaluation bridge is installed (default) then a `backquoted string` will be treated like a normal parsed literal.

Variable Bridges

A variable bridge is a container for storing scalars. A variable bridge is nothing more than a container. It is possible to use a new variable container to alter how scalars are stored and accessed. All scalars, scalar arrays, and scalar hashes are stored using this system.

A Variable bridge is created by implementing the sleep.interfaces.Variable interface.

public boolean scalarExists(String variable)

Return true if a scalar variable exists in this container. Called whenever a scalar is requested by a script. This method is used to check if the scalar exists or not.

public Scalar getScalar(String variable)

If the scalar does exist (as determined by scalarExists()) this method will be called to request the scalar itself. The reference of the scalar can be returned. Passing scalars by value (and not by reference) is taken care of by the Scalar class.

public Scalar putScalar(String key, Scalar value)

If a scalar does exist (as determined by scalarExists()) this method will be called when a script wants to store a scalar value.

public void removeScalar(String key)

If a scalar exists and a script wants to remove a scalar this method will be called.

public Variable createLocalVariableContainer()

This method is called to return a class that implements Variable for maintaining local variables. This method is only called on the Variable class used as the global variable container.

public Variable createInternalVariableContainer()

This method is called to return a class that implements Variable for maintaining the internal variable context. Internal variables are variables that are global to a single Script Instance only. This method is only called on the Variable class used as the global variable container.

A Variable bridge is installed by creating a new script variable manager with the new variable bridge. The variable manager is then installed into a given script.

```
ScriptVariables variableManager = new ScriptVariable(new MyVariable());
script.setScriptVariables(variableManager);
```

Sleep scripts can share variables by using the same instance of ScriptVariables. A Variable bridge can be used to create built in variables. Every time a certain scalar is accessed the bridge might call a method and return the value of the method as the value of the accessed scalar.

Environment Bridges

Blocks of code associated with an identifier are processed by their environment. An example of an environment is the subroutine environment. To declare a subroutine in sleep you use:

```
sub identifier { commands; }
```

When sleep encounters this code it looks for the environment bound to the keyword "sub". It passes the environment for "sub" a copy of the script instance, the identifier, and the block of executable code. The environment can do anything it wants with this information. The subroutine environment simply creates a Function object with the block of code and installs it into the environment. Thus allowing scripts to declare custom subroutines.

In general a block of code is associated with an environment using the following syntax:

```
keyword identifier { commands; } # sleep code
```

The interface for an environment is:

public void bindFunction(ScriptInstance instance, String keyword, String identifier, Block commands)

Called when the keyword for the environment is encountered i.e. keyword identifier { commands; }. The identifier is passed in as a string. Identifiers can also be enclosed within " double or ' single quotes. The block object contains all of the executable code for the commands associated with this identifier.

Script environment bridge keywords should be registered with the script parser before any scripts are loaded. This can be accomplished as follows:

```
ParserConfig.addKeyword("keyword");
```

To install a new environment into the script environment:

```
ScriptInstance script;              // assume
Environment    myEnvironmentBridge; // assume

Hashtable environment = script.getScriptEnvironment().getEnvironment();
environment.put("keyword", myEnvironmentBridge);
```

The Block object passed to the environment can be executed using:

```
SleepUtils.runCode(commands, instance.getScriptEnvironment());
```

Environment bridges are great for implementing different types of paradigms. I've used this feature to add everything from event driven scripting to popup menu structures to my application. Environments are a very powerful way to get the most out of integrating your application with the sleep language.

Filtered Environment Bridges

Filtered environments are similar to normal keyword environments except they also allow a parameter specified by the user. The identifier and parameter are both sent to the bridge when a block of code is bound to a particular filtered environment keyword.

In general the syntax for binding a filtered environment is:

```
keyword identifier "parameter" { code; }
```

The filtered environment interface is pretty similar to a normal environment interface:

public void bindFilterFunction(ScriptInstance script, String keyword, String parameter, Block body)

This method is called when a filter environment is to be bound. The script instance of the calling script is passed in. The keyword for the environment is passed in as well. Something worth noting is that the parameter string is passed in as-is from the source file. i.e. if the user specified a "string", the value of parameter will be "string". If a user script specified an expression i.e. (2 + 2) the parameter will be the string "(2 + 2)". The parameter string does have to adhere to sleep syntax rules though. i.e. no mismatched parentheses, mismatched quotes etc.

The parameter keyword is passed in unparsed to allow you, the bridge writer, choice in the matter of what to do with the parameter. Depending on the purpose of the bridge some may want to evaluate the parameter when the block of code is first bound. Others may want to evaluate the parameter each time the bridge carries out its actions. In any case to evaluate the parameter string as a sleep expression:

```
ScriptEnvironment environment = script.getScriptEnvironment();
Scalar value = environment.evaluateExpression(parameter);
```

Filter environment bridge keywords should be registered with the script parser before any scripts are loaded. This can be accomplished as follows:

```
ParserConfig.addKeyword("keyword");
```

To install a new filter environment into the script environment:

```
ScriptInstance    script;              // assume
FilterEnvironment myEnvironmentBridge; // assume

Hashtable environment = script.getScriptEnvironment().getEnvironment();
environment.put("keyword", myEnvironmentBridge);
```

Filter environments are really just an extension of the normal environment bridges. In particular Filter environments can be used to implement event listener mechanisms or as an expansion for normal environment bridges that might require a parameter.

Predicate Environment Bridges

Predicate environments are similar to normal keyword environments except instead of binding commands to an identifier they are bound to a predicate condition.

In general the syntax for declaring a predicate environment is:

```
keyword (condition) { commands; }
```

The predicate environment interface looks like:

public void bindPredicate(ScriptInstance instance, String keyword, Check condition, Block commands)

> This method is called when a predicate is to be bound. The script instance of the calling script is passed in. The keyword for the environment is passed in as well. The condition for whether or not to execute this code is passed in as a Check instance. The commands to execute are defined within the Block instance.

Predicate environment bridge keywords should be registered with the script parser before any scripts are loaded. This can be accomplished as follows:

```
ParserConfig.addKeyword("keyword");
```

To install a new predicate environment into the script environment:

```
ScriptInstance          script;              // assume
PredicateEnvironment    myEnvironmentBridge; // assume

Hashtable environment = script.getScriptEnvironment().getEnvironment();
environment.put("keyword", myEnvironmentBridge);
```

Predicate environments are a powerful way to create environments that are triggered selectively. Predicate environments can also be used to add new constructs to the sleep language such as an unless (comparison) { } construct.

Generating Runtime Warning Messages

Sleep has a built-in mechanism for notifying the scripter of runtime warning messages. Typically if an error occurs inside of a bridge it is helpful to make any relevant information available to the scripter. In general any uncaught exceptions will be caught by the Sleep runtime and formatted into a Sleep error message with the appropriate Sleep script line number/file isolated.

As a convention though, two exceptions are useful for bridge writers. These include *java.lang.IllegalArgumentException* and *java.lang.RuntimeException*. Both of these exceptions are formatted cleanly without the exception message name. Keep in mind an exception really should only be thrown for an exceptional case. When a bridge throws an exception Sleep will jump out of the current executing block of code. So use them sparingly and only when the script has encountered an unrecoverable error.

Bridge Design Patterns

A convenient way to design a bridge is to group bridges of a certain type together. Create each bridge function, predicate, or operator as a static inner class. Then make the enclosing outer class a loadable bridge that instantiates each inner class and adds each item to the script environment.

Below is an example of this design pattern:

```
public class MyBridge implements Loadable
{
   public boolean scriptLoaded(ScriptInstance script)
   {
      Hashtable environment =
            script.getScriptEnvironment().getEnvironment();
      environment.put("&function", new MyFunction());
      environment.put("-predicate", new MyPredicate());
   }

   public boolean scriptUnloaded(ScriptInstance script)
   {
      return true;
   }
```

```
    private static class MyFunction implements Function
    {
       public Scalar evaluate(String name, ScriptInstance si, Stack args)
       {
           // code for MyFunction
       }
    }

    private static class MyPredicate implements Predicate
    {
       public boolean decide(String name, ScriptInstance si, Stack args)
       {
           // code for MyPredicate
       }
    }
}
```

The above Loadable bridge with all of its API's contained within can then be installed into the script loader:

```
ScriptLoader   loader = new ScriptLoader();
loader.addSpecificBridge(new MyBridge());
```

Now each time a script is loaded using loader, &function and -predicate will be installed into the script's environment. I've found this pattern very useful for making my own bridges.

Bridge Utilities Class

The Bridge Utilities class is a in the sleep.bridges package. It contains many static methods for extracting java types from a java.util.Stack of Scalar arguments. Use of this class is recommended for extracting arguments as it provides a safe way to do it.

Multithreaded Bridges

Sleep is safe to use in a multi-threaded application. This is accomplished by internal synchronization that allows only one block of sleep code to be in the process of execution at any time regardless of the number of threads. I should say one block of Sleep code per shared environment. A shared environment is defined as scripts sharing the same variable set. Sleep internally synchronizes on the *sleep.runtime.ScriptVariables* object held by each *ScriptEnvironment* object.

Bottom line: if a sleep function is executing, any attempt to execute a sleep function in another thread will block until the executing function returns.

The recommended way of dealing with 'blocking' code is to build bridges that accept a callback function. In the Sleep IO API the `&read` function takes an IO source and a callback function as a parameter. When data is available from the IO source the callback function is called. Underneath the hood, the `&read` function spins off a new thread. Any blocking that occurs by trying to read from the specified IO source is contained in this new thread. When its time for a script to do something (i.e. start executing, process the read data, stop executing) the callback function is executed.

Sleep scripts can happily interact in a multithreaded context. The trick is to force any blocking/waiting to occur within Java itself. Once the blocking/waiting is done just call a preregisted block of sleep code to respond to the event.

Multithreading with Fork

The Sleep function `&fork` implementation creates a new script environment entirely separate from the currnet one. Forked code can run in multiple threads without blocking. To create a forked script instance simply call the *fork()* method on a ScriptInstance object. This will share the script environment without sharing any variables.

Data can be returned from a fork if your implementation uses *sleep.bridges.io.IOObject*. A thread contained by an IOObject can be waited on within a Sleep script using the `&wait($handle)` function. To setup an IOObject for waiting simply call *setThread(java.lang.Thread t)* on an IOObject instance. To pass a return value from a fork use the *setToken(Scalar s)* on the IOObject instance containing the thread.

Part 3: Extending Sleep Input/Output

Nearly everything built into Sleep is implemented via Sleep bridges. Period. The Input/Output system is no exception. The Sleep IO library provides functionality for reading ascii-based text data and binary data. By default Sleep provides facilities for accessing files, sockets, and processes. What makes the interface fun is that it is all unified and easily extensible.

This chapter presents the architecture of the IO system and describes the implementation of the socket piece. The goal is to thoroughly explain a complete Sleep API implementation.

The Input/Output World

The Sleep I/O World really consists of 3 pieces. The first piece is some sort of class that defines an IO source. This IO source class inherits from *sleep.bridges.io.IOObject*. The IOObject class provides all of the compatability with Sleeps current array of IO functions. The last piece is the IO Bridge itself. The IO Bridge is just a normal set of Sleep bridges that implement functions, predicates, and operators that make the functionality of the IO Source and the IOObject accessible to scripters.

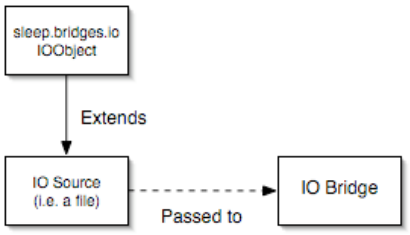

Figure 2. Sleep IO Architecture

The IOObject Class

The IOObject is the parent class for all IO Source object that are compatible with Sleep's I/O API.

Methods

public DataInputStream getReader()

Returns the binary data reader

public DataOutputStream getWriter()

Returns the binary data writer

public void openRead(InputStream in)

Initializes a binary reader (a DataInputStream) and a text reader (a BufferedReader) against this input stream.

public void openWrite(OutputStream out)

Initializes a binary writer (a DataOutputStream) and a text writer (a PrintWriter) against this input stream.

public void sendEOF()

Closes down the output streams effectively sending an end of file message to the reading end.

An IO Source Implementation

IO Sources extend the sleep.bridges.io.IOObject class (detailed above). The IO Source class is responsible for providing 'starter' methods that initialize the streams correctly. The starter methods can be whatever the developer likes. They are later tied explicitly to a sleep function via a Function bridge.

Starter methods have the responsibility of initializing the IO source and setting up the IO streams. IO streams are intialized by calling the openRead() and openWrite() methods of the parent IOObject class.

The following example is the Sleep implementation of the Socket IO Source:

```
package sleep.bridges.io;

import java.io.*;
import java.net.*;
import sleep.runtime.*;

public class SocketObject extends IOObject
{
   protected Socket socket;

   public void open(String server, int port, ScriptEnvironment env)
   {
      try
      {
         socket = new Socket(server, port);
         socket.setSoLinger(true, 5);

         openRead(socket.getInputStream());
         openWrite(socket.getOutputStream());
      }
      catch (Exception ex)
      {
         env.flagError(ex.toString());
      }
   }

   public void listen(int port, int timeout, Scalar data, ScriptEnvironment env)
   {
      try
      {
         ServerSocket server = new ServerSocket(port);
         server.setSoTimeout(timeout);
```

```
            socket = server.accept();
            socket.setSoLinger(true, 5);

            data.setValue(SleepUtils.getScalar(socket.getInetAddress().getHostAddress()));

            openRead(socket.getInputStream());
            openWrite(socket.getOutputStream());
         }
         catch (Exception ex)
         {
            env.flagError(ex.toString());
         }
      }

      public void close()
      {
         super.close();

         try
         {
            socket.close();
         }
         catch (Exception ex) { }
      }
   }
```

Notice that two 'starter' methods are provided. A 'open' method which connects to a server and the 'listen' method which waits for a connection. Once a connection has been established both of these start methods call openRead and openWrite with the appropriate input/output streams.

Having an appropriate IO source class is great however it does not provide anything new for the scripter. The next step is to tie the starter methods into a sleep function.

The IO Bridge

To make the socket functionality accessible, the relevant sleep function bridges have to be created. In the case of the socket API there are 2 unique and 1 overriden functions in Sleep:

closef($handle)
: closes the IO for $handle

connect("host", port)
: connects to the specified host:port and returns a $handle.

listen(port, [timeout], [$host])
: listens for a connection on the specified port.

The &closef function is implicit to all IO objects. When &closef is called, the close() method of the IO source is called.

The sleep functions &connect and &listen instantiate a SocketObject and call the appropriate starter method. As an example, the implementation of the &connect bridge is below:

```
   public class connect implements Function
   {
      public Scalar evaluate(String func, ScriptInstance script, Stack args)
      {
         String host = BridgeUtilities.getString(args, "");
         int    port = BridgeUtilities.getInt(args);

         SocketObject socket = new SocketObject();
         socket.open(host, port, script.getScriptEnvironment());

         return SleepUtils.getScalar(socket);
      }
   }
```

Note that the arguments are extracted from the argument stack. The SocketObject is created. The appropriate starter method is called and finally the socket object is wrapped into a scalar and returned.

Finally, the function needs to be made available in the script environment. This is typically done at script load time using a sleep.interfaces.Loadable bridge. Loadable bridges are registered with a script loader. Whenever a script is loaded or unloaded the script loader class calls the scriptLoaded/scriptUnloaded methods of all registered Loadable bridges.

The relevant pieces of a Loadable bridge to install the &connect function are:

```
   public boolean scriptLoaded (ScriptInstance script)
   {
      Hashtable env = script.getScriptEnvironment().getEnvironment();
      env.put("&connect",  new connect());
   }
```

Part 4: Working with Scalars

What is a Scalar

Scalars have been mentioned in this document many times. Scalars are just sleep variables. Scalars can be strings, numbers, or even a reference to an object. Scalars are represented by the java class sleep.runtime.Scalar.

Scalar objects maintain a reference to an instantiated scalar type. A scalar type represents data of a certain type. A scalar type also contains logic to convert the stored data to any of the primitive types used by sleep. Sleep primitive types include double, int, long, object, and String.

When a scalar is instantiated its type is determined. Example:

$x = 3.4;

When the above is executed 3.4 is determined to be a double. 3.4 is then created as a scalar type double, this scalar type double is stored in the scalar *$x*. Any attempt to use *$x* as a string would result in the value of "3.4". Any attempt to use *$x* as an integer would result in the value of 3.

The following table below illustrates the scalar types and how the data would be converted to other types if needed.

Type	**Example**	**Double**	**Int**	**Long**	**Object**	**String**
Double	3.3	3.3	3	3L	new Double(3.3)	"3.3"
Integer	3.3	4	4.0	4L	new Integer(4)	"4"
Long	5	5.0	5	5L	new Long(5)	"5"
Object *	ref	hashcode	hashcode	hashcode	ref	ref.toString()
String **	"3rd place"	0	0	0L	"3rd place"	"3rd place"
$null		0.0	0	0L	null	""

** The word ref within the Object column refers to a reference to an object.*
*** If a string value contains a number i.e. "42" then the string can be used as a number*

To obtain the string value of a Scalar:

```
String value = scalar.toString();
```

To obtain the integer value of a Scalar:

```
int value = scalar.intValue();
```

For more information on querying data from a Scalar see the Java API Documentation for sleep.runtime.Scalar.

Instantiating a Scalar

Instantiating a Scalar is most easily done using the sleep.runtime.SleepUtils class. The SleepUtils class contains several static methods for creating a Scalar object from data.

The general pattern for this is a `SleepUtils.getScalar(data)` method. There are static getScalar() methods that take a double, int, long, Object, or a String as a parameter.

There are even methods for wrapping java data structures into a scalar array or scalar hash. Methods also exist to copy data from one scalar into another new scalar.

Examples:

```
Scalar anInt   = SleepUtils.getScalar(3); // create an int scalar
Scalar aDouble = SleepUtils.getScalar(4.5); // create a double scalar
Scalar aString = SleepUtils.getScalar("hello"); // string scalar
Scalar anArray = SleepUtils.getArrayWrapper(new LinkedList(); // array scalar
```

For more information see the sleep.runtime.SleepUtils Java API Documentation.

Working with Array and Hashtable Scalars

To add a value to a Scalar array:

```
Scalar arrayScalar = SleepUtils.getArray(); // empty array
arrayScalar.getArray().add(SleepUtils.getScalar("value"), 0);
```

To iterate through all of the values in a Scalar array:

```
Iterator i = arrayScalar.getArray().scalarIterator();
while (i.hasNext())
```

```
{
    Scalar temp = (Scalar)i.next();
}
```

For more information see the sleep.runtime.ScalarArray Java API Documentation.

To add a value to a Scalar hashtable:

```
Scalar hashScalar = SleepUtils.getHashScalar(); // blank hashtable
Scalar temp = hashScalar.getHash().getAt(SleepUtils.getScalar("key"));
temp.setValue(SleepUtils.getScalar("value"));
```

The second line obtains a Scalar for "key". The returned Scalar is just a container. It is possible to set the value of the returned scalar using the setValue method.

Internally scalar values in sleep are passed by value. Methods like setValue inside of the Scalar class take care of copying the value. Externally though Scalar objects are passed by reference. When you call getAt() in the ScalarHash you are obtaining a reference to a Scalar inside of the hashtable. When you change the value of the Scalar you obtained, you change the value of the Scalar in the hashtable.

For more information see the sleep.runtime.ScalarHash Java API Documentation.

Creating a Scalar Type

If wrapping data into an already existing scalar type isn't enough. It is possible to create a new scalar type that performs its own conversions and everything.

To create a new type of scalar create a class that implements the sleep.runtime.ScalarType interface. The interface is self explanatory. Simply implement several methods that ask for the stored value as a certain primitive type.

To store a custom scalar type in a scalar:

```
Scalar temp = SleepUtils.getScalar(); // returns an empty scalar.
temp.setValue(new MyScalarType());
```

In the above example MyScalarType is an instance that implements the ScalarType interface.

Creating a Scalar Array Implementation

It is possible to create a scalar array with your own backend implementation of the scalar array interface. This allows for an easy way to make your data structures in your application available to scripters.

To create a new type of scalar array create a class that implements the sleep.runtime.ScalarArray interface. The scalar array interface asks for methods that define all of the common operations on sleep arrays.

To instantiate a custom scalar array:

```
Scalar temp = SleepUtils.getArrayScalar(new MyScalarArray());
```

In the above example MyScalarArray is the class name of your new scalar array implementation.

Creating a Scalar Hash Implementation

It is also possible to create a hashtable scalar with your own backend implementation of the scalar hash interface. This allows for an easy way to make your data structures in your application available to scripters.

To create a new type of scalar hash: create a class that implements the sleep.runtime.ScalarHash interface. The scalar hash interface asks for methods that define all of the common operations on sleep hashes.

To instantiate a custom scalar hash:

```
Scalar temp = SleepUtils.getHashScalar(new MyHashScalar());
```

In the above example MyHashScalar is the class name of your new scalar hash implementation.

Part 5: How-to Guide

This section describes some of the miscellaneous stuff one might need to know.

Install an Escape Constant

In sleep a character prefixed by a \ backslash within a "double quoted" string is said to be escaped. Typically an escaped character is just skipped over during processing. It is possible in sleep to add meaning to different characters by installing an escape. For example to add the escape \r to mean the new line character one would do the following:

```
sleep.parser.ParserConfig.installEscapeConstant('r', "\n");
```

Once the above code is executed the value "blah\r" inside of sleep would be equivalent in java to "blah\n".

Force Scripts to Share Information

By default sleep scripts are isolated from each other. Isolated scripts do not share variable or function information. It is possible to get scripts to share this information:

Functions and other environment information are typically stored in a java.util.Hashtable. Each loadScript() method inside of the Script Loader has a similar method that takes a java.util.Hashtable argument. Passing the same Hashtable to all of your loaded scripts will force those scripts to share function information.

```
Hashtable environment = new Hashtable();

ScriptLoader loader = new ScriptLoader();
ScriptInstance a = loader.loadScript("script1.sl", environment);
ScriptInstance b = loader.loadScript("script2.sl", environment);
```

In the above example the Script Instance's a and b are both sharing the same function environment. If a subroutine is declared in script a it will be available in script b. The script loader is also smart about shared environments.

When installing a loadable bridge you have the choice of installing a bridge as a global or a specific bridge. The two methods for this are addGlobalBridge() and addSpecificBridge() in sleep.runtime.ScriptLoader. Specific bridges are processed for every single script that is loaded whether they are sharing environments or not. Global bridges will only be processed if they have not already been loaded into the specified environment. This saves some overhead when loading multiple scripts sharing the same environments.

Variables in sleep are managed by the class sleep.runtime.ScriptVariables. To force scripts to share variables among multiple instances:

```
ScriptVariables variables = new ScriptVariables();

ScriptLoader loader = new ScriptLoader();

ScriptInstance a = loader.loadScript("script1.sl");
a.setScriptVariables(variables);

ScriptInstance b = loader.loadScript("script2.sl");
b.setScriptVariables(variables);
```

In the above example the Script Instance's a and b are both sharing the same variable information.

Execute a Block of Code

To execute a block of code:

```
Block             code; // assume
ScriptEnvironment env  = script.getEnvironment();

Scalar value = SleepUtils.runCode(code, env);
```

Evaluate Code from a String

To evaluate an expression:

```
Scalar value = env.evaluateExpression("2 + 2");
```

To evaluate a predicate expression:

```
boolean condition = env.evaluatePredicate("2 == 2");
```

To evaluate a statement or series of statements:

```
env.evaluateStatement("while ($x < 100) { $x++; }");
```

Catch a Syntax Error when loading a Script

Syntax errors are a reality of programming. Any time a syntax error occurs when attempting to load a script the exception YourCodeSucksException will be raised. [yes, the exception name is staying]

To catch a YourCodeSucksException:

```
try
{
   ScriptInstance script;
   script = loader.loadScript("name", inputStream);
}
catch (YourCodeSucksException ex)
{
```

```
      Iterator i = ex.getErrors().iterator();
      while (i.hasNext())
      {
         SyntaxError error = (SyntaxError)i.next();

         String description = error.getDescription();
         String code        = error.getCodeSnippet();
         int    lineNumber  = error.getLineNumber();
      }
   }
```

Catch a Runtime Script Error

Runtime errors are caught by sleep. Examples of a runtime error include calling a function that doesn't exist, using an operator that doesn't exist, or causing an exception in the underlying java code. Whenever any of these events occurs the event is isolated and turned into a ScriptWarning object. The Script Warning object is then propagated to all registered warning watchers.

To create a runtime warning watcher:

```
public class Watchdog implements RuntimeWarningWatcher
{
   public void processScriptWarning(ScriptWarning warning)
   {
      String message = warning.getMessage();
      int    lineNo  = warning.getLineNumber();
      String script  = warning.getNameShort(); // name of script
   }
}
```

To register a warning watcher (assume script is a ScriptInstance object):

```
script.addWarningWatcher(new Watchdog());
```

Integrate the Sleep Console

The sleep console is an interactive console for working with sleep scripts. The console includes commands for loading scripts, running scripts, and dumping an abstract syntax tree of parsed scripts. Integrating the sleep console consists of building a console proxy that provides input/output services for the actual Sleep Console.

An example console proxy (using STDIN/STDOUT) is below:

```
import sleep.io.*;
import sleep.console.ConsoleProxy;

public class MyConsoleProxy implements ConsoleProxy
{
   protected BufferedRead in;

   public MyConsoleProxy()
   {
      in = new BufferedReader(new InputStreamReader(System.in));
   }

   public void consolePrint(String message)
   {
      System.out.print(message);
   }

   public void consolePrintln(Object message)
   {
      System.out.println(message.toString());
   }

   public String consoleReadln()
   {
      try
      {
         return in.readLine();
      }
      catch (IOException ex)
      {
         ex.printStackTrace();
         return null;
      }
   }
}
```

To instantiate the Sleep Console and install a custom console proxy:

```
// assume ScriptEnvironment environment; ScriptVariables variables; ScriptLoader loader;

ConsoleImplementation console;
console = new ConsoleImplementation(environment, variables, loader);

console.setProxy(new MyConsoleProxy());
console.rppl(); // starts the console
```

The above will instantiate a sleep console with the specified function environment, script variables, and script loader. Once an implementation of the ConsoleProxy interface is installed the sleep console is ready to use. Any application taking advantage of the sleep console should instantiate it before scripts are loaded. This is necessary as the sleep console installs itself as a loadable bridge into the script loader.

Glossary:

Bridge

A piece of code that brings application functionality together with sleep scripts. Called a bridge because it bridges the gap between an application and the sleep scripting language.

Block

A block of code in sleep is a ready to execute block of parsed sleep code. Blocks do not have any variable or environment information associated with them. They are simply parsed sleep code.

Environment

Environment is a little bit of an overloaded term in sleep. There is the script environment class which contains methods for accessing the data stack, return value of a function, and the real script environment. In sleep the real script environment is a simple java.util.Hashtable. A hashtable stores all of the bridged information, all of the scripted subroutines, and all of the scalar values. It is possible in Sleeps API's to have all scripts be isolated from each other (i.e. they have there own Hashtable environment). Scripts can share subroutines and variables simply by sharing a Hashtable reference.

Scalar

A scalar is the universal data type for sleep variables. Scalars can have numerical values of integer, double, or long. Scalars can have a string value. Scalars can also contain a reference to a scalar array, scalar hash, or a generic Java object.

Script Instance

A script instance in sleep is a loaded script. It contains all of the environment information for the loaded script. It contains the Block of executable sleep code. The script instance class contains several utility methods for checking if that script instance represents a loaded script, calling functions, and querying the script for information.

Appendix A: Example Code

Contents of ScriptExample.java:

```
import sleep.interfaces.Function;
import sleep.interfaces.Loadable;

import sleep.runtime.ScriptLoader;
import sleep.runtime.ScriptInstance;
import sleep.runtime.SleepUtils;
import sleep.runtime.Scalar;

import sleep.error.YourCodeSucksException;
import sleep.bridges.BridgeUtilities;

import java.io.IOException;
import java.util.Stack;

public class ScriptExample
{
   public static void main (String args[]) throws IOException, YourCodeSucksException
   {
       // initialize the script loader and add a bridge
       ScriptLoader   loader = new ScriptLoader();
       loader.addSpecificBridge(new SetupBridge());

       ScriptInstance script = loader.loadScript("script.sl"); // loads the script
```

```
        script.runScript(); // evaluates the script

        // put a scalar into the environment
        script.getScriptVariables().putScalar("$test", SleepUtils.getScalar("sleep example"));

        // retrieve a scalar from the environment
        Scalar test  = script.getScriptVariables().getScalar("$test");

        // call a function
        Scalar value = script.callFunction("&konk", new Stack());

        // unload the script
        loader.unloadScript(script);
    }

    private static class FooFunction implements Function
    {
        public Scalar evaluate(String name, ScriptInstance script, Stack parameters)
        {
            System.out.println("function foo has been called");
            return SleepUtils.getEmptyScalar();
        }
    }

    private static class MyAddFunction implements Function
    {
        public Scalar evaluate(String name, ScriptInstance script, Stack parameters)
        {
            int arg1 = BridgeUtilities.getInt(parameters, 0);
            int arg2 = BridgeUtilities.getInt(parameters, 0);

            return SleepUtils.getScalar(arg1 + arg2);
        }
    }

    private static class SetupBridge implements Loadable
    {
        public boolean scriptLoaded(ScriptInstance script)
        {
           Hashtable env = script.getScriptEnvironment().getEnvironment();

           env.put("&foo",   new FooFunction());
           env.put("&myadd", new MyAddFunction());

           return true;
        }

        public boolean scriptUnloaded(ScriptInstance script)
        {
           return true;
        }
    }
}
```

Contents of script.sl

```
sub konk
{
   println("Konk has been called: \$test is $test");
}

foo();
println("Calling built in: " . myadd(6, 36));
```