A concept for and an implementation of the Bean Scripting Framework for Rexx

Peter Kalender

sw1163@uni−essen.de

Table of contents

1. The basics

The 'Bean Scripting Framework' from IBM allows other programs to control Java applications with scripts written in Non−Java scripting languages. The goal of this seminar is to analyze the BSF and to design a possible concept for interfacing Rexx to the BSF and finally to write an implementation of that interface.

Before this problem can be tackled there are some basic questions that need to be answered first.

1.1 What is Rexx?

Rexx is a scripting language, that was developed by Mike Cowlishaw of IBM UK Laboratories in 1979. It is based on a language called EXEC 2. The first EXEC was not much more than a batch language, that was used to wrap up several (system−)commands into one.

EXEC 2 was an enhanced version of EXEC with a focus on writing macros for a wide variety of applications. The mayor drawback of this language was the rather complicated way it handled variables and language keywords. It was mainly designed to handle commands, but more complex macros needed variables and control structures.

So Mike Cowlishaw came up with a language, he called REX (because it sounded nice), in which he tried to unite the powerful command and string programming facilities of EXEC 2 with more classical syntax and semantics of languages like Pascal or PL/I. The first specification of this new language, which also included three sample programs, was released on 29 March 1979. At that time not a single line of implementation code had been written. The specification was first discussed and refined and in late 1979 a first version was released on IBM's world−wide internal network. It soon became very popular and a lot of suggestions for improvement were made over the net. This direct feedback was and still is one of Rexx's trademarks and it insures, that the language is as close to the user's needs as possible. REX went through some major changes over the years, one of them was adding an 'X', so it is now called Rexx, an abbreviation for **RE**structured e**X**tended e**X**ecutor.

In 1987 Rexx became IBM's Procedures Language for its Systems Application Architecture (SAA). The first Rexx compiler was developed at IBM's Vienna Laboratory in 1989. In 1990 the first international Rexx Symposium for Developers and Users was held. Rexx was included in all versions of OS/2 and could even handle multimedia devices.

In 1990 an object oriented version of Rexx became available and in 1996 NetRexx was introduced.

Rexx today is a very powerful scripting language. It was designed to be a macro language, so it is possible to introduce new functions to Rexx that enable it to control Non−Rexx programs. It is also very easy to use and very readable. Rexx is available for almost any platform.

1.2 What are Java Beans?

The Java Beans API allows users to create component software in Java. Components are small independent software units, that can be used to visually create new applications.

Beans expose their features to builder tools. The tool puts the bean into a toolbox. From there users are able to grab it, add it to an application, change it’s appearance and define it’s behavior if certain events occur. All beans are able to fire and handle events, through which they can communicate with each other. For example a button−bean might fire a ’clicked’−event and a text−field−bean might receive that event and react, by displaying a new text.

All these bean−features can be manipulated in a visual way, without having to write a single line of code.

Similar concepts can be found in Borland Delphi, or Microsoft Visual Basic.

1.3 What is the BSF? − A quick overview

The Bean Scripting Framework is an architecture that allows Java and scripting languages to interact. There are two ways to do this.

The first, would be to write a Java program and to use certain sub−commands or functions, that are written in a scripting language. For example, most scripting languages are very good at processing and evaluating strings, so instead of writing dozens of lines of Java code, a user might choose, to let a small script handle the string and return the result back to the Java program.

Java programs are able to call functions and get a return value. They can also evaluate, execute or even compile a piece of script, depending on the scripting engine.

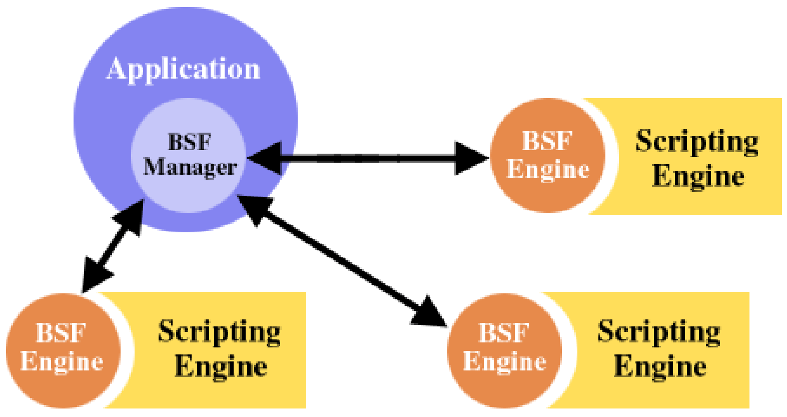


Figure 1: The BSF architecture

The second way would be to use the scripting language to provide the ’main’ program, that controls the Java Beans. For example, a user might want to use Java Beans to get a nice graphical interface for his Rexx program.

The core bean of the BSF architecture is the BSFManager. This bean provides all the services necessary to incorporate scripting into Java. It has a registry of scripting engines, that the BSF can handle. It also has a registry for beans, that, once registered, can be manipulated by the scripting engines.

The BSFEngine is the interface, that connects the BSFManager and the Scripting Engine. In most cases, this will be implemented in Java, but in some cases it is implemented in another language and connected to Java via JNI. The BSFEngine allows scripts to look up pre−

registered beans, to register new beans, to modify beans, register events and call bean functions.

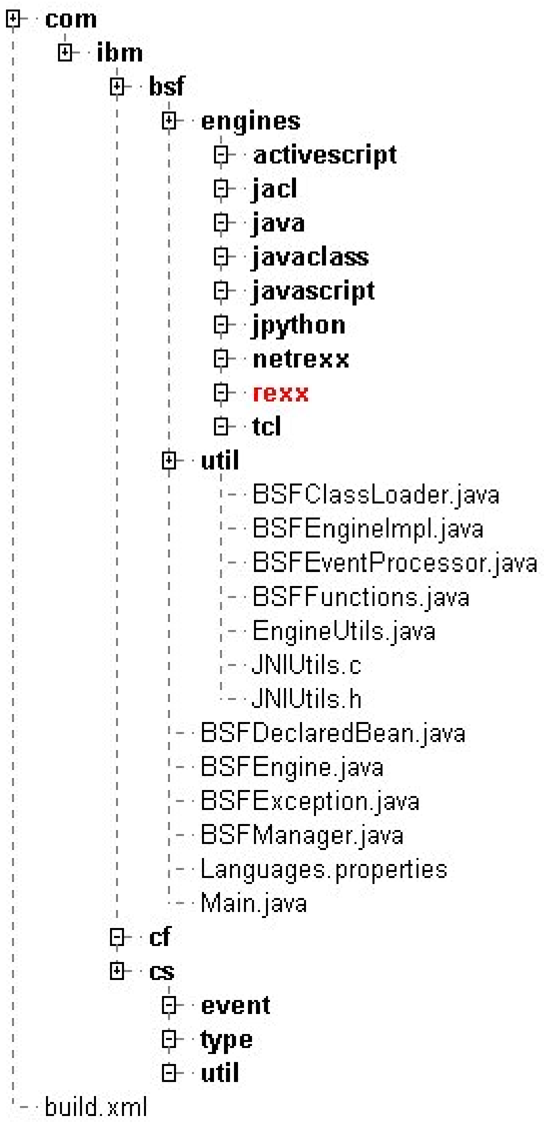
The term *Bean* Scripting Framework is also a bit misleading, because in fact the framework can be used to access all Java Objects, not just beans.

The current version is 2.1 and it supports the following languages:

- BML
- Jacl
- JavaScript
- Jpython
- NetRexx
- TCL
- XSL/T
- On Win32: VBScript, Jscript and PerlScript

2. A closer look at the Bean Scripting Framework

Now it's time to take a closer look at all the components, that make up the Bean Scripting Framework. Figure 2 shows the main files and directories.



```
com
  ibm
    bsf
      engines
        activescript
        jacl
        java
        javaclass
        javascript
        jpython
        netrexx
        rexx
        tcl
      util
        BSFClassLoader.java
        BSFEngineImpl.java
        BSFEventProcessor.java
        BSFFunctions.java
        EngineUtils.java
        JNIUtils.c
        JNIUtils.h
      BSFDeclaredBean.java
      BSFEngine.java
      BSFException.java
      BSFManager.java
      Languages.properties
      Main.java
    cf
    cs
      event
      type
      util
build.xml
```

Figure 2: The main BSF files

2.1 The 'bsf' directory

The 'bsf' directory contains all the 'core' files, the 'cf' and 'cs' directories contain just 'helper' classes.

All the engines are located in sub−directories of the 'engines' directory. In most of these there is just one file present, implementing all engine functions relevant to that scripting engine. The 'activescript' directory contains several files, implementing all the Win32 scripting engines.

Some very useful utilities can be found in the ’utils’ directory. There are some ’JNIUtils’ for those scripting languages that can’t be directly accessed from Java and have to use the Java Native Interface to call C Functions. The ’BSFClassLoader’ can load classes from a temporary directoy. The ’BSFEventProcessor’ is used to bind scripts to certain events. The ’BSFFunctions’ contain a ’bsf’ class, that can be used as an object in some scripting languages. These languages then use the methods this object provides to access their beans. ’EngineUtils’ contains several methods to create beans, call functions and create event listeners.

The ’BSFDeclaredBean’ is used internally to pass information between the BSFManager and it’s scripting engines. The ’BSFException’ class defines six different bsf−exceptions that can be thrown. The ’Languages.properties’ file lists all the different scripting languages that BSF supports, the names and locations of their engines and the file extensions used by scripts written in a certain language.

2.2 ’BSFEngine’ & ’BSFEngineImpl’

The ’BSFEngine’ is just an interface, that every scripting engine needs to implement. To make things easier, there already is a basic implementation of the ’BSFEngine’, ’BSFEngineImpl’. Most of the scripting engines choose to extend this class.

But what’s in there? First of all, there is an initialize and a terminate method, doing all the usual stuff, that can be expected from methods of that name. (For more details see Chapter 3).

Then there are several methods to handle scripts. The ’call’ method is used to call functions belonging to an object. ’apply’ tries to invoke an anonymous function, that is a value returning piece of script. The ’eval’ method evaluates a piece of script, in most cases a string and may return a value. ’exec’ executes a piece of script, but doesn’t return a value. ’compileApply’, ’compileExpr’ and ’compileScript’ compile pieces of script and dump the resulting code in the code buffer. ’compileApply’ is for anonymous functions, ’compileExpr’ is for value returning expressions and ’compileScript’ is for compiling pieces of scripts.

Of course not all engines use all of these functions. For example many scripts can’t be compiled, they are just interpreted.

Finally, there are ’declareBean’ and ’undeclareBean’ methods. (See chapter 2.3 for more details on these.)

2.3 ’BSFManager’

The ’BSF’Manager’ is the central class of the whole BSF. A Java application wanting to use scripts has to include a ’BSFManager’.

The ’BSFManager’ offers several methods to handle scripting engines. The engines are stored in a hash−table. There are methods to register and unregister engines and there is a method to that looks up if an engine is registered or not. File extensions are also kept in a hash−table and can be looked up to find out to which scripting engine a certain file belongs. There is also a method to load scripting engines. All engines contained in the ’Languages.properties’ file are automatically registered on startup.

There are several methods to handle debugging and some to handle class loaders. An object registry and a temporary directory are defined.

Beans can be registered and unregistered. Registered beans are kept in the object registry and can be accessed by name and be manipulated. Beans can also be declared and undeclared. Declared beans are automatically registered as well, but declared beans are also supposed to be made pre−available to scripting engines. In object oriented scripting languages, that would mean to create objects in the scripting language that corresponding to the declared beans. Furthermore there is a lookup function, that returns a handle to a given bean.

Finally, there are all the methods for handling scripts, which call their counterparts, of the needed engines.

2.4 ’Main’

If an application is to be written in a scripting language and that application wants to use Java objects, then it needs to use the BSF. But in most cases you won’t be able to start up the BSF from a scripting language, so a little trick is needed. That’s what this ’Main’ class was made for.

It implements a ’BSFManager’, offering all the BSF functions. Then it tries to run a script specified by command line arguments. Said script is then able to use all BSF functions, thus remote controlling Java objects.

2.5 The ’cf’ & ’cs’ directories

The ’cf’ directory contains a code−formatter−bean, which can be used to format raw Java code.

The ’cs’ directory contains some tools. The ’event’ directory contains several classes, that can set up event handlers, file them in a registry and handle incoming events. In the ’type’ directory there is a type converter, that is able to convert an object from one class to another. The ’util’ directory contains utilities for many different tasks. For example, there are string−utilities, code−buffer−utilities, etc.

3. Rexx and the BSF

First things first. The BSFManager needs to know, that it will soon support Rexx scripts. So the 'Languages.properties' file has to be edited and the line

```
rexx = com.ibm.bsf.engines.rexx.RexxEngine, rexx
```

needs to be added. The left part is the language descriptor, that the BSFManager uses, the middle part is the engine class and on the right are the file extensions, that scripting engine's files use. In case of Rexx, there is just one extension.

3.1 Beans in Rexx

One of the biggest problems when implementing a BSF scripting engine for Rexx is how to represent objects in a non−object−oriented language. The only 'objects' Rexx knows, are strings. But you can't call methods belonging to a string (because there aren't any) and you can't add event listeners to strings.

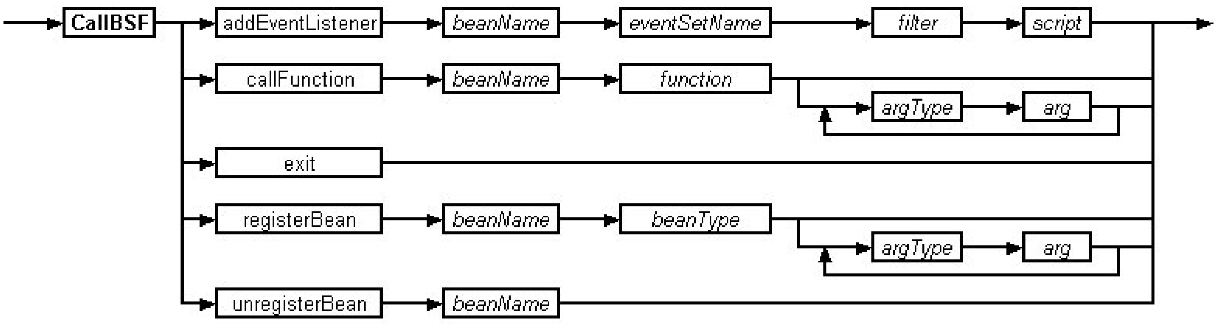


Figure 3: The CallBSF function

Fortunately beans don't have to be kept in the scripting language. The BSFManager does that instead. So the only thing Rexx needs to be able to do is to call the BSFManager and tell it what to do with a certain bean. To this end the CallBSF function was designed.

The CallBSF function is actually five functions in one. The first parameter specifies, which function is to be called.

If there are no pre−registered beans available, Rexx first needs to register some beans of it's own. To do so, CallBSF is used, with the first parameter being 'registerBean'. The second parameter has to be the bean's name. This can be any name and is later used to identify that bean. The third parameter specifies the type of that bean. A very basic type would be 'java.awt.frame', an empty window. The following parameters are optional constructor arguments.

These arguments are a bit tricky, because Rexx can only provide strings, where Java expects integers, booleans and several other types. So somewhere on their way from Rexx to the constructor method a type conversion has to take place. To let the converter know into which type an argument has to be converted the arguments have to come in pairs of argument type and argument value.

Possible values for argType are:

- B : boolean
- I : integer
- L : long
- F : float
- D : double
- O : bean (object)
- S : string

In case of a ’java.awt.frame’ object a string containing the window title is expected so the command would look something like this:

```
call CallBSF ’registerBean’, ’Window’, ’java.awt.frame’, ’S’,
’Title’
```

If a bean is no longer needed, it may be unregistered with CallBSF. The first parameter has to be ’unregisterBean’ and the second has to be the name of the bean.

To call a certain function belonging to a certain bean, the first parameter of CallBSF has to be ’callFunction’, the second, as always, is the name of the bean and the third is the name of the function. Again, the arguments for the function being called come in pairs of argument type and argument value, so they can be converted into the correct type.

In order for beans to interact with each other, event listeners have to be added. To do this, CallBSF is used with ’addEventListener’ as the first parameter. The second parameter is the name of the bean firing the event. The third parameter specifies an event set. Event sets group events by categories. For example, there are ’WindowListener’ events, ’MouseListener’ events, ’ActionListener’ events and so on. The ’filter’ parameter specifies the exact event from a set and the fifth parameter is the script to be run, if the event fires.

For example, in case of a ’java.awt.frame’ something should happen if the user tries to close the window. In that situation it may be a good idea, to terminate the program as well. To do this, a command like this might be used:

```
call CallBSF ’addEventListener’, ’Window’, ’window’,
’windowClosing’, ’exit’
```

But unfortunately, this won’t work, because it will only terminate the Rexx programm, but the Java program providing the BSFManager will keep on running and the frame won’t be closed. To handle this situation, the CallBSF function was extended, to include an exit function, that terminates the Java application running the script. So to make the above example work, it would have to look like this:

```
call CallBSF ’addEventListener’, ’Window’, ’window’,
’windowClosing’, ’CallBSF(’exit’)’
```

3.2 Connecting Java and Rexx

The other big problem is how to connect Java and Rexx to each other. Rexx can be accessed by other programming languages via the Rexx SAA API. Unfortunately the Rexx SAA API uses a C header file, so it would be best to have a C or C++ application handling the access to Rexx. And C functions can be called from Java through the JNI.

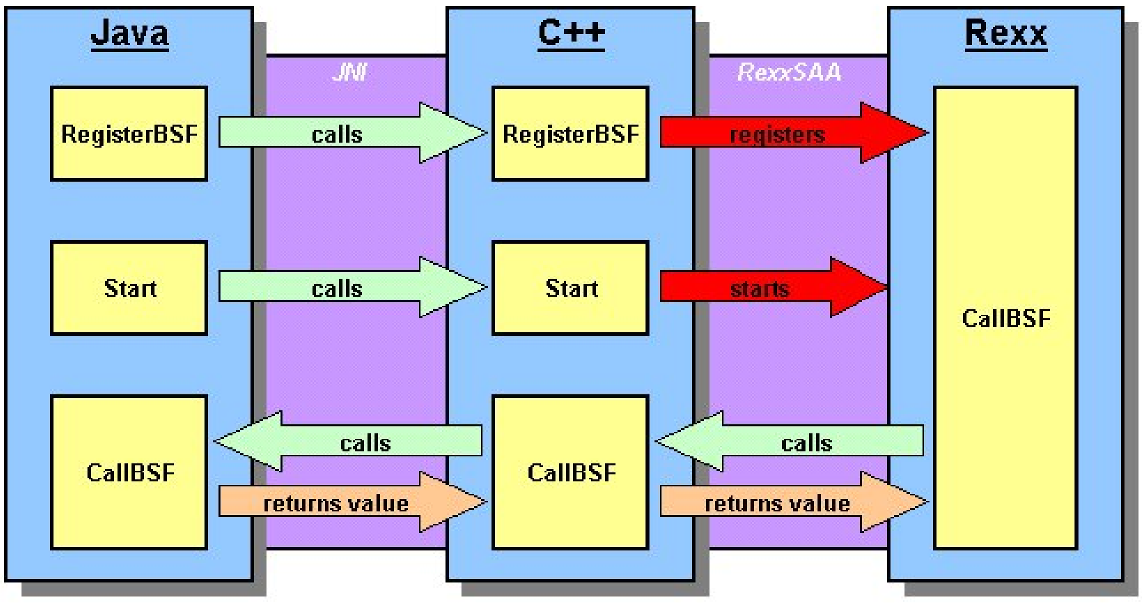


Figure 4: Connecting Java and Rexx

3.2.1 Rexx SAA API

The **Rexx** **S**ystems **A**pplication **A**rchitecture **A**pplication **P**rogramming **I**nterface is Rexx's port to the outside world. Through it Rexx can communicate with other high level programming languages. In most cases this will be C or C++.

The Rexx SAA API functionality can be divided into several areas:

- Subcommand handlers
  can be used to register subcommands written in another language than Rexx. Every call to such a command is trapped and executed by the external application. Subcommands can either be part of that application or they might be part of a dynamic library. Subcommands don't have any arguments, but they may return a value.

- External function handlers
  are quite similar to subcommand handlers. They can be registered with Rexx and are either part of an external application or a dynamic library. The main difference is, that external functions may receive any number of arguments. They also may return a value.

- Interpreting
  allows non−Rexx applications to execute pieces of Rexx code.

- Variable interface
  enables applications to set, retrieve and drop variables in the Rexx interpreter.

- System exits
  enable an application to trap certain situations, like Rexx terminating, or Rexx accessing I/O streams. If a predefined situation arises, the application may take over and handle it itself.

To create a link between BSF and Rexx, only external function handlers and interpreting of Rexx scripts will be needed.

3.2.2 JNI

The **J**ava **N**ative **I**nterface is Java's port to other programming languages.

The JNI's functionality can be divided into several areas:

- Native methods
  enable Java to access functions written in Non−Java code. To do so, a Java method has to be declared as being 'native'. It is then compiled normally, using javac. Next a header file has to be created with javah. That header can then be included in other programming languages. The native code has to be compiled into a shared library, so the Java application can use it.

- Callbacks
  allow native methods to call Java methods. To do so, the native method has to get a method ID. To do so, the name and argument types of the Java method have to be known.

- Mapping
  is a way of accessing Java types from a native language. Especially strings and arrays need to be converted, before they can be use in a language like C++.

- Invoking the Java Virtual Machine
  enables a Non−Java application to embed a JVM and use it to run Java applets.

To create a link between Java and Rexx, native methods, callbacks and mapping will be needed.

3.3 Implementing the Rexx engine for BSF

The actual implementation of the Rexx engine was in many parts inspired by the Jacl engine. For example, the Jacl engine uses something quite similar to the CallBSF method.

The first thing that is needed is the engine itself. It is an extension of the BSFEngineImpl and thus has to implement at least an ’initialize’ method and a ’call’ method. Some method to run a script would also be nice.

3.3.1 Initializing the engine

The ’initialize’ method calls it’s parents initalizer and creates a new JavaRexxSAA object. This object uses JNI to call C++ functions, that call Rexx SAA API functions. Furthermore the RegisterBSF method is called.

RegisterBSF is a native method written in C++. Because the JNI adds the package name to the function name, in C++ the function is called ’Java_com_ibm_bsf_engines_rexx_JavaRexxSAA_RegisterBSF’. All it does is to register a new function called ’CallBSF’ with the Rexx interpreter.

3.3.2 Running a Rexx script

The BSF offers several ways to run scripts, but because Rexx scripts can’t be compiled, so all the ’compile*’ methods wouldn’t make sense. ’apply’ returns an object, that Rexx can’t handle and ’exec’ doesn’t return anything at all. So the most appropriate way to run Rexx scripts seems to be the ’eval’ method.

A ’call’ method has also to be present, because it’s part of the BSFEngine interface. But this method is used to call a method belonging to an object, which Rexx can’t provide, so this method was left empty. The only thing it does is returning ’null’.

The ’eval’ method of the Rexx engine is used in two ways. The first is to run a normal piece of Rexx script that is stored on disc in some file. The second way is to run a script as a reaction to some event. In most cases the ’main’ Rexx script will already have terminated when an event fires. That’s because some Java objects, like the ’java.awt.frame’ will keep on running until they are explicitly terminated. In that case an event script trying to call a subroutine or function defined in the ’main’ Rexx program won’t be able to find it anymore. So a ’dirty trick’ is used.

The main script is stored in the ’mainScript’ string for future reference. Whenever an event script is called, ’eval’ attaches ’mainScript’ to the event script. That way all the subroutines and functions the event script might need are present. To keep the Rexx interpreter from running the ’main’ script again a ’return’ command is inserted between the event script and the ’main’ script.

To run a piece of script the engine uses the ’Start’ method, written in C++. Start converts the Java string containing the source script into a Rexx string and calls the RexxStart method.

3.3.3 Calling the BSF from Rexx

If a Rexx application wants to call the BSF, it uses the CallBSF function. There are actually two functions of that name, but the one Rexx calls is the one implemented in C++. This function converts all the arguments stored in an array of Rexx strings into an array of Java strings and then calls the Java function. Any return value is converted to a Rexx string and passed back to Rexx.

The main problem here is that a JNI environment pointer is needed to call back to a Java function. No pointer of that kind is present in this function and normally this would be a dead end.

But another ’dirty trick’ is used. Because it is assumed that all Rexx programs calling ’CallBSF’ will be run from the ’Start’ function, the JNI environment that function receives from Java can be used. To this end the ’JNIenv’ pointer is stored in a global variable. The same goes for the object pointer. Now ’CallBSF’ can use these to call back to Java. If the Rexx program calling ’CallBSF’ is not run from the ’Start’ function this pointer will point nowhere and the program will crash.

The Java version of the ’CallBSF’ function does all the ’real’ BSF stuff. It checks, which function of the five avaible is to be used. Next it checks if the correct number of arguments was used and converting all that need conversion. Then it calls the appropriate BSF function, making heavy use of the ’EngineUtils’. These can handle most of the tasks, from calling bean functions to adding event listeners.

For more detailed information about the implementation, see Appendix C.

4. Conclusion

As it turned out, creating a concept for a BSF Rexx engine implementation was quite interesting and not too hard, although at first it didn’t seem that way at all. "Objects in a non−oo−language, impossible!" was my first thought. Well it’s possible after all.

In the end, the biggest problem I had was my C compiler, who just didn’t seem to like me.

For future versions of this engine, the C part of the engine should be made to throw exceptions if something goes wrong. And the argument type conversion in ’CallBSF’ might be put in a separate function. Maybe some better way of handling subroutines written in Rexx could be found, the ’dirty trick’ of always attaching the whole source code to event scripts works, but it may slow bigger programs down a lot.

And maybe the engine could be extended for Object Rexx. In that case beans might be properly declared and even the ’call’ function could be realized. Perhaps. But that’s not part of this seminar.

Appendix A − Directory of figures

Appendix B − Bibliography

Rexx

http://www.RexxLA.org
http://wwwi.wu−wien.ac.at/Studium/LVA−Unterlagen/poolv/folien/

Rexx history

http://www2.hursley.ibm.com/rexx/rexxhist.htm
http://www2.hursley.ibm.com/rexx/faq4.htm

Regina Rexx interpreter

http://www.lightlink.com/hessling/Regina/

BSF

http://oss.software.ibm.com/developerworks/opensource/bsf/
http://www.javaworld.com/javaworld/jw−03−2000/jw−03−beans.html

Java

http://java.sun.com/

JNI

http://java.sun.com/docs/books/tutorial/
http://developer.java.sun.com/developer/onlineTraining/Programming/JDCBook/index.html

Appendix C − Source codes

RexxEngine.java

```
package com.ibm.bsf.engines.rexx;

import java.util.*;
import java.io.*;
import com.ibm.bsf.*;
import com.ibm.bsf.util.BSFEngineImpl;
import com.ibm.bsf.engines.rexx.JavaRexxSAA;

/**********************************************************
 * This is the interface to use (Regina) Rexx with the BSF
 **********************************************************/

public class RexxEngine extends BSFEngineImpl {

   private JavaRexxSAA Rexx;  //the link to the Rexx interpreter
   private String mainScript;     //used to store the script, needed
                                  //for calls to Rexx functions

   /**********************************************************
    * 'initialize' creates a link to the Rexx interpreter and
    * registers the 'callBSF' function with Rexx
    **********************************************************/
   public void initialize (BSFManager mgr, String lang,
                   Vector declaredBeans) throws BSFException {
      super.initialize (mgr, lang, declaredBeans);
      Rexx = new JavaRexxSAA(mgr, this);
      Rexx.RegisterBSF();
   }

   /**********************************************************
    * 'eval' runs a piece of Rexx script
    * if the script is marked as beeing an '<event−script>'
    * the main script is added, so that any subcommands defined
    * there can be run appropriately
    **********************************************************/
   public Object eval (String source, int lineNo, int columnNo,
                 Object oscript) throws BSFException {
      String script = oscript.toString ();
      if(source.equals("<event−script>")){
         script = script+"\nReturn;\n"+mainScript;
      } else {
         mainScript = script;
      }
      Rexx.Start(script);
      return null;
   }
   /*************************************************************
    * 'call' has to be present, because it's part of the
    * BSFEngine interface, however, there is no way to implement
    * calls to Objects in a non−oo−language
    *************************************************************/
   public Object call (Object object, String method, Object[] args)
                   throws BSFException {
      return null;
   }
}
```

JavaRexxSAA.java

```
package com.ibm.bsf.engines.rexx;

import java.util.*;
import java.io.*;
import com.ibm.bsf.*;
import com.ibm.bsf.util.*;
import com.ibm.bsf.BSFManager;
import com.ibm.bsf.util.BSFEngineImpl;

class JavaRexxSAA {
   BSFManager mgr;
   BSFEngine rengine;

/**********************************************************
 * This class uses JNI to call a dynamic library written in C++
 * that access Rexx through the Rexx SAA API
 **********************************************************/
   JavaRexxSAA(BSFManager mgr, BSFEngine rengine) {
      this.mgr = mgr;              //the current BSFManager
      this.rengine = rengine;      //the current RexxEngine
   }

   /**********************************************************
    * 'RegisterBSF' − a native function to register the 'CallBSF'
    * function with the Rexx interpreter
    **********************************************************/
   public native long RegisterBSF();

   /**********************************************************
    * 'Start' − a native function to start evaluating a piece
    * of Rexx script
    **********************************************************/
   public native long Start(String script);

   /**********************************************************
    * a static method to load the dynamic library 'JaReSAA'
    **********************************************************/
   static {
      System.loadLibrary("JaReSAA");
   }

    /**********************************************************
     * The 'CallBSF' function is called by Rexx to access the BSF
     * First parameter has to be a function name
     *       − addEventListener
     *       − callFunction
     *       − exit
     *       − registerBean
     *       − unregisterBean
     *   Second paramter is a bean name
     *   The following parameters are the 'real' parameters
     *   for the called function
     *
     * in some cases argTypes are used these can be:
     * − B: boolean
     * − I: integer
     * − L: long
     * − F: float
     * − D: double
     * − O: bean (object)
     * − S: string
     ************************************************************/
   public String CallBSF(String args[]) throws BSFException{
      /**********************************************************
       * CallBSF addEventListener beanName eventSetName filter script
       *
       * − beanName: the bean that fires the event
       * − eventSetName: the set the event belongs to
       * − filter: the actual event
       * − script: the script to be run, if the event fires
       **********************************************************/
      if(args[0].equalsIgnoreCase("addEventListener")) {
        if(args.length!=5){
          throw new BSFException (BSFException.REASON_EXECUTION_ERROR,
            "invalid # of args; usage: CallBSF " +
            "addEventListener beanName eventSetName filter script)");
        }
        Object bean = mgr.lookupBean(args[1]);
        if(bean==null){
          throw new BSFException (BSFException.REASON_EXECUTION_ERROR,
            "Bean ’"+args[1]+"’ not registered");
        }
        args[3] = args[3].equals("") ? null : args[3];
        try {
          EngineUtils.addEventListener(bean, args[2], args[3],
                                rengine, mgr, "<event−script>",
                                0, 0, args[4]);
        } catch (BSFException e){
          e.printStackTrace ();
          System.out.println("got BSF exception: " + e.getMessage ());
        }
      }
      /**********************************************************
       * CallBSF callFunction beanName function (argType arg)
       *
       * − beanName: the bean whos function is to be called

       * − function: the function’s name
       * − argType: the type of an argument
       * − arg: the argument’s value
       **********************************************************/
      if(args[0].equalsIgnoreCase("callFunction")) {
        if((args.length<3) | ((args.length−3)%2!=0)){
          throw new BSFException (BSFException.REASON_EXECUTION_ERROR,
            "invalid # of args; usage: CallBSF " +
              "callFunction beanName function (argType arg)");
        }
        Object bean = mgr.lookupBean(args[1]);

        if(bean==null){
          throw new BSFException (BSFException.REASON_EXECUTION_ERROR,
            "Bean '"+args[1]+"' not registered");
        }
        //convert function−arguments to the correct type
        Object funcArgs[] = new Object[(args.length−3)/2];
        if(args.length>3){
          for(int i=3, j=0; i<args.length; i+=2, j++){
            switch(args[i].charAt(0)){
              case 'B':
                funcArgs[j] = new Boolean(args[i+1]);
                break;
              case 'I':
                funcArgs[j] = new Integer(args[i+1]);
                break;
              case 'L':
                funcArgs[j] = new Long(args[i+1]);
                break;
              case 'F':
                funcArgs[j] = new Float(args[i+1]);
                break;
              case 'D':
                funcArgs[j] = new Double(args[i+1]);
                break;
              case 'O':
                funcArgs[j] = mgr.lookupBean(args[i+1]);
                if(funcArgs[j]==null){
                  throw new BSFException (BSFException.REASON_EXECUTION_ERROR,
                    "Bean '"+args[i+1]+"' not registered");
                }
                break;
              case 'S':
                funcArgs[j] = args[i+1];
                break;
              default:
                throw new BSFException (BSFException.REASON_EXECUTION_ERROR,
                  "argType '"+args[i]+"' unknown");
            }
          }
        }
        //call the function
        Object result;
        try {
          result = EngineUtils.callBeanMethod(bean, args[2], funcArgs);

          if(result==null) {return null;}
          else{
            return result.toString();
          }
        } catch (BSFException e){
          e.printStackTrace ();
          System.out.println("got BSF exception: " + e.getMessage ());
        }
      }
```

```
/*****************************************************
 * CallBSF exit
 * terminates the application
 *****************************************************/
if(args[0].equalsIgnoreCase("exit")) {
   System.exit(0);
}
/*****************************************************
 * CallBSF registerBean beanName beanType (argType arg)
 *
 * − beanName: the bean’s name
 * − beanType: the bean’s type
 * − argType: the type of a constructor argument
 * − arg: the argument’s value
 *****************************************************/
if(args[0].equalsIgnoreCase("registerBean")) {
   if((args.length<3) | ((args.length−3)%2!=0)){
      throw new BSFException (BSFException.REASON_EXECUTION_ERROR,
         "invalid # of args; usage: CallBSF " +
            "registerBean beanName beanType (argType arg)");
   }
   //convert arguments to the correct type
   Object funcArgs[] = new Object[(args.length−3)/2];
   if(args.length>3){
      for(int i=3, j=0; i<args.length; i+=2, j++){
         switch(args[i].charAt(0)){
            case ’B’:
               funcArgs[j] = new Boolean(args[i+1]);
               break;
            case ’I’:
               funcArgs[j] = new Integer(args[i+1]);
               break;
            case ’L’:
               funcArgs[j] = new Long(args[i+1]);
               break;
            case ’F’:
               funcArgs[j] = new Float(args[i+1]);
               break;
            case ’D’:
               funcArgs[j] = new Double(args[i+1]);
               break;
            case ’O’:
               funcArgs[j] = mgr.lookupBean(args[i+1]);
               if(funcArgs[j]==null){
                  throw new BSFException (BSFException.REASON_EXECUTION_ERROR,
                     "Bean ’"+args[i+1]+"’ not registered");
               }
               break;
            case ’S’:
               funcArgs[j] = args[i+1];
               break;
            default:
               throw new BSFException (BSFException.REASON_EXECUTION_ERROR,
                  "argType ’"+args[i]+"’ unknown");
         }
```

```
            }
        }
        //create the bean
        try {
            Object bean = EngineUtils.createBean(args[2],funcArgs);
            mgr.registerBean(args[1], bean);
        } catch (BSFException e){
            e.printStackTrace ();
            System.out.println("got BSF exception: " + e.getMessage ());
        }
    }
    /*******************************************************
     * CallBSF unregisterBean beanName
     *
     * − beanName: the bean to unregister
     *******************************************************/
    if(args[0].equalsIgnoreCase("unregisterBean")) {
        if(args.length!=2){
            throw new BSFException (BSFException.REASON_EXECUTION_ERROR,
                "invalid # of args; usage: CallBSF " +
                "unregisterBean beanName");
        }
        mgr.unregisterBean(args[1]);
    }
    return null;
  }
}
```

JaReSAA.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <jni.h>
#include "com_ibm_bsf_engines_rexx_JavaRexxSAA.h"
#define INCL_RXSHV
#define INCL_RXFUNC
#include <rexxsaa.h>

#ifdef _MSC_VER
# pragma warning(disable:4100)
#endif

#define DLLNAME "JaReSAA"
#define FUNCTION1 CallBSF
#define NAME_FUNCTION1 "CallBSF"

RexxFunctionHandler CallBSF;

JNIEnv *GlobEnv;        //used by CallBSF to call back to Java
jobject GlobObj;        //used by CallBSF to call back to Java

/**********************************************************
 * 'CallBSF' calls back to the Java method 'CallBSF'
 **********************************************************/
APIRET APIENTRY
FUNCTION1(PSZ name,
                ULONG argc,
                PRXSTRING argv,
                PSZ queuename,
                PRXSTRING retstr) {
        //get Java method 'CallBSF'
        jclass cls = (*GlobEnv).GetObjectClass(GlobObj);
        jmethodID mid = (*GlobEnv).GetMethodID(cls, "CallBSF",
                                        "([Ljava/lang/String;)Ljava/lang/String;");
        if (mid == 0) return −1;

        //convert arguments array to Java format
        jobjectArray arr;
        arr = (*GlobEnv).NewObjectArray(argc, (*GlobEnv).FindClass("java/lang/String"),
                                (*GlobEnv).NewStringUTF(""));
        for(ULONG i=0; i<argc; i++) {
              (*GlobEnv).SetObjectArrayElement(arr, i, (*GlobEnv).NewStringUTF(argv[i].strptr));
        }

        //call Java method and create Rexx return value
        jstring ReturnValue = (jstring)(*GlobEnv).CallObjectMethod(GlobObj, mid, arr);
        if(ReturnValue!=NULL) {
                const char *str = GlobEnv−>GetStringUTFChars(ReturnValue, 0);
                retstr−>strptr = new char[strlen(str)];
                strcpy(retstr−>strptr, str);
                retstr−>strlength = strlen(str);
        } else {
                retstr->strptr = new char[0];
                retstr->strlength = 0;
        }
        return 0;
}

/**********************************************************
 * 'RegisterBSF' registers the 'CallBSF' method with the
 * Rexx interpreter
 **********************************************************/
JNIEXPORT jlong JNICALL
Java_com_ibm_bsf_engines_rexx_JavaRexxSAA_RegisterBSF(JNIEnv *env,
                                                      jobject obj) {
        LONG ReturnValue;
        ReturnValue = RexxRegisterFunctionDll("CallBSF", "JaReSAA.dll", "CallBSF");
        return ReturnValue;
}

/**********************************************************
 * 'Start' runs a piece of Rexx script
 **********************************************************/
JNIEXPORT jlong JNICALL
Java_com_ibm_bsf_engines_rexx_JavaRexxSAA_Start(JNIEnv *env,
                                                jobject obj,
                                                jstring source) {
        //set global variables for callbacks to Java
        GlobEnv = env;
        GlobObj = obj;

        //convert Java string containing script to Rexx string
        const char *str = env->GetStringUTFChars(source, 0);
        char *sourceStr;
        sourceStr = new char[strlen(str)];
        strcpy(sourceStr, str);
        PRXSTRING Instore;
        RXSTRING RXsourceStr[2];
        MAKERXSTRING(RXsourceStr[0], sourceStr, strlen(sourceStr));
        MAKERXSTRING(RXsourceStr[1], NULL, NULL);

        //start Rexx
        SHORT ReturnCode;
        LONG ReturnValue;
        char ResultBuffer[250];
        RXSTRING Result;
        MAKERXSTRING(Result, ResultBuffer, sizeof(ResultBuffer));
        Instore = &RXsourceStr[0];
        PSZ     ProgramName = "";
        ReturnValue = RexxStart(
                        0,                      //LONG          ArgCount
                        NULL,                   //PRXSTRING ArgList
                        ProgramName,            //PSZ           ProgramName
                        Instore,                        //PRXSTRING  Instore
                        NULL,                   //PSZ           EnvName
                        RXSUBROUTINE,           //LONG          CallType
                        NULL,                   //PRXSYSEXIT Exits

                        &ReturnCode,            //PUSHORT       ReturnCode
                        &Result                       //PRXSTRING Result
                        );
        GlobalFree(&RXsourceStr[1]);
        env−>ReleaseStringUTFChars(source, str);
        return ReturnValue;
}
```

Appendix D − Sample programs

Hello World!

This small program creates a new window and puts a button in it. When either the window is closed or the button is pressed, the program terminates.

```
call CallBSF ’registerBean’, ’Window’, ’java.awt.Frame’, ’S’, ’Hello World!’
call CallBSF ’addEventListener’, ’Window’, ’window’, ’windowClosing’, ’CallBSF(’exit’)’

call CallBSF ’registerBean’, ’Button’, ’java.awt.Button’, ’S’, ’Press me!’
call CallBSF ’addEventListener’, ’Button’, ’action’, ’’, ’CallBSF(’exit’)’

call CallBSF ’callFunction’, ’Window’, ’add’, ’O’, ’Button’
call CallBSF ’callFunction’, ’Window’, ’pack’
call CallBSF ’callFunction’, ’Window’, ’show’
call CallBSF ’callFunction’, ’Window’, ’toFront’

return
```

The ouput of this program can be seen in figure 5.

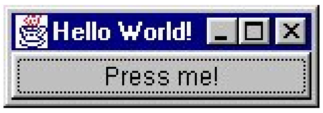


Figure 5: A small "Hello world" programm

Show Size

This program creates a window containing a label and a button. Whenever the button is pressed, the label is updated to show the current size of the frame.

```
call CallBSF ’registerBean’, ’Window’, ’java.awt.Frame’, ’S’, ’Show size’
call CallBSF ’addEventListener’, ’Window’, ’window’, ’windowClosing’, ’CallBSF(’exit’)’

call CallBSF ’registerBean’, ’Button’, ’java.awt.Button’, ’S’, ’Press me!’
call CallBSF ’addEventListener’, ’Button’, ’action’, ’’, ’call ShowSize’

call CallBSF ’registerBean’, ’Label’, ’java.awt.Label’
call CallBSF ’callFunction’, ’Label’, ’setAlignment’, ’I’, ’1’

call CallBSF ’callFunction’, ’Window’, ’add’, ’S’, ’Center’, ’O’, ’Label’
call CallBSF ’callFunction’, ’Window’, ’add’, ’S’, ’South’, ’O’, ’Button’
call CallBSF ’callFunction’, ’Window’, ’pack’
call CallBSF ’callFunction’, ’Window’, ’show’
call CallBSF ’callFunction’, ’Window’, ’toFront’
return

ShowSize:
call CallBSF ’callFunction’, ’Window’, ’getSize’
parse var result "width=" width ",height=" height "]"
call CallBSF ’callFunction’, ’Label’, ’setText’, ’S’, "(" width "," height ")"
return
```

The ouput of this program can be seen in figure 6.

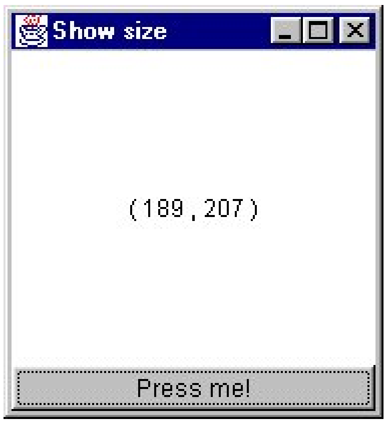


Figure 6: The "ShowSize" program

Greetings

This program demonstrates how a dialog window might be created with Rexx. The window contains a list box and two text fields. The user is asked to select his/her gender and to enter his/her first name and surname. The program will then greet the user appropriately.

```
call CallBSF ’registerBean’, ’Label1’, ’java.awt.Label’, ’S’, ’Gender: ’
call CallBSF ’registerBean’, ’Label2’, ’java.awt.Label’, ’S’, ’First Name: ’
call CallBSF ’registerBean’, ’Label3’, ’java.awt.Label’, ’S’, ’Surname: ’
call CallBSF ’registerBean’, ’DropDown’, ’java.awt.Choice’
call CallBSF ’callFunction’, ’DropDown’, ’addItem’, ’S’, ’male’
call CallBSF ’callFunction’, ’DropDown’, ’addItem’, ’S’, ’female’
call CallBSF ’registerBean’, ’Surname’, ’java.awt.TextField’
call CallBSF ’registerBean’, ’Name’, ’java.awt.TextField’
call CallBSF ’registerBean’, ’OK’, ’java.awt.Button’, ’S’, ’OK’
call CallBSF ’addEventListener’, ’OK’, ’action’, ’’, ’call Greetings’
call CallBSF ’registerBean’, ’Cancel’, ’java.awt.Button’, ’S’, ’Cancel’
call CallBSF ’addEventListener’, ’Cancel’, ’action’, ’’, ’CallBSF(’exit’)’

call CallBSF ’registerBean’, ’Background’, ’java.awt.Color’, ’I’, ’150’, ’I’, ’150’, ’I’, ’250’

call CallBSF ’registerBean’, ’GLayout’, ’java.awt.GridLayout’, ’I’, ’4’, ’I’, ’2’
call CallBSF ’registerBean’, ’Window’, ’java.awt.Frame’, ’S’, ’Greetings!’
call CallBSF ’addEventListener’, ’Window’, ’window’, ’windowClosing’, ’CallBSF(’exit’)’

call CallBSF ’callFunction’, ’Window’, ’setLayout’, ’O’, ’GLayout’
call CallBSF ’callFunction’, ’Window’, ’add’, ’O’, ’Label1’
call CallBSF ’callFunction’, ’Window’, ’add’, ’O’, ’DropDown’
call CallBSF ’callFunction’, ’Window’, ’add’, ’O’, ’Label2’
call CallBSF ’callFunction’, ’Window’, ’add’, ’O’, ’Name’
call CallBSF ’callFunction’, ’Window’, ’add’, ’O’, ’Label3’
call CallBSF ’callFunction’, ’Window’, ’add’, ’O’, ’Surname’
call CallBSF ’callFunction’, ’Window’, ’add’, ’O’, ’OK’
call CallBSF ’callFunction’, ’Window’, ’add’, ’O’, ’Cancel’
call CallBSF ’callFunction’, ’Window’, ’setSize’, ’I’, ’200’, ’I’, ’200’
call CallBSF ’callFunction’, ’Window’, ’setBackground’, ’O’, ’Background’
call CallBSF ’callFunction’, ’Window’, ’pack’
call CallBSF ’callFunction’, ’Window’, ’show’
call CallBSF ’callFunction’, ’Window’, ’toFront’
return

Greetings:
call CallBSF ’registerBean’, ’Window2’, ’java.awt.Frame’
call CallBSF ’addEventListener’, ’Window2’, ’window’, ’windowClosing’,
’CallBSF(’exit’)’
call CallBSF ’registerBean’, ’Greet’, ’java.awt.Label’
call CallBSF ’registerBean’, ’Bye’, ’java.awt.Button’, ’S’, ’Bye!’
call CallBSF ’addEventListener’, ’Bye’, ’action’, ’’, ’CallBSF(’exit’)’

call CallBSF ’callFunction’, ’Window2’, ’add’, ’S’, ’North’, ’O’, ’Greet’
call CallBSF ’callFunction’, ’Window2’, ’add’, ’S’, ’South’, ’O’, ’Bye’
call CallBSF ’callFunction’, ’Window2’, ’pack’
call CallBSF ’callFunction’, ’Window2’, ’show’
call CallBSF ’callFunction’, ’Window2’, ’toFront’

call CallBSF ’callFunction’, ’DropDown’, ’getSelectedItem’
parse var Result Text1
call CallBSF ’callFunction’, ’Name’, ’getText’
parse var Result Text2
call CallBSF ’callFunction’, ’Surname’, ’getText’
parse var Result Text3
if Text1=="male" then call CallBSF ’callFunction’, ’Greet’, ’setText’, ’S’, "Hello Mr." Text2
Text3;
else call CallBSF ’callFunction’, ’Greet’, ’setText’, ’S’, "Hello Mrs." Text2 Text3;
return
```

The ouput of this program can be seen in figure 7.

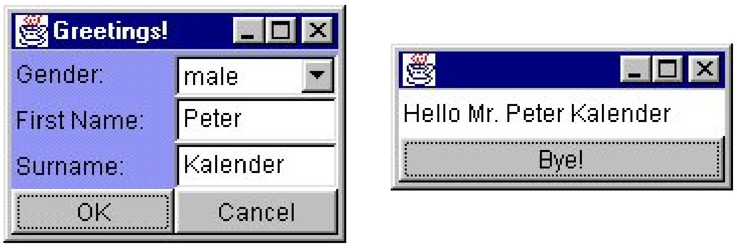


Figure 7: The Greetings program