ooRexxTry Reengineering

Seminar Paper

Markus Moldaschl

0751916

0502 Projektseminar aus Wirtschaftsinformatik
Ao. Univ.-Prof. Dr. Rony G. Flatscher
Institute for Management Information Systems
Vienna University of Economics and Business Administration

Table of Contents

List of Figures

List of Tables

1 Abstract

This paper discusses the functionality of the program `ooRexxTry` and describes the transformation from the original Windows bound version to a platform independent one which leverages Java GUI libraries. Furthermore, it gives a brief overview of ooRexx itself and important ooRexx concepts utilized in the project work.

In addition, the paper briefly introduces the very exciting, before the time of writing, undocumented feature of Java class specialization in ooRexx and points out some extensions made to the original `ooRexxTry` such as a code execution history and the ability to directly parse input from a file source.

2 ooRexxTry.rex

`ooRexxTry.rex` is a GUI based ooRexx script that can be used very much like `rexxtry.rex` that is also distributed with Open Object Rexx. However, they significantly vary in the way they run code and in their availability. rexxtry.rex runs a string entered in the command prompt when you press the `RETURN` key. `ooRexxTry.rex` starts the execution of your code when you click on the “Run” button on its GUI. Since `ooRexxTry.rex` leverages Windows libraries for building its GUI, `ooRexxTry.rex` is only available on the Windows platform. You may also supply arguments to your code by specifying them in the arguments section of the dialog (cf. figure 1). [Peed07]

ooRexxTry runs on every Windows distribution with a version of ooRexx 3.2 or higher installed.

2.1 ooRexx Basics

Open Object Rexx (**ooRexx**) is an Open Source project managed by Rexx Language Association (RexxLA) providing a free implementation of Object Rexx. **ooRexx** is distributed under Common Public License (CPL) v1.0. Object Rexx is an enhancement of classic Rexx; a powerful, full-featured programming language which has a human-oriented syntax. [ooRexx]

The following list points out the main characteristics and benefits of ooRexx:

- Natural language-like

  ooRexx uses commands full of meaningful English words and a set of self-explanatory instructions like SAY, IF … THEN … ELSE, DO … END, LEAVE and EXIT. That is why it is easy to learn and students may quickly become productive with it.

- Syntax rules

  A Rexx program is built from a series of clauses that are composed of zero or more whitespace characters, a sequence of tokens (e.g. literal

strings, symbols, …), zero or more whitespace characters and a semicolon (;) delimiter that the line end, certain keywords, or the colon (:) implies. Please see [FlatIntro] and subsequent slide sets for more information on the syntax of ooRexx.

Conceptually, each clause is scanned from left to right before processing, and the tokens composing it are identified. Instruction keywords are recognized at this stage, comments are removed, and sequences of whitespace characters (except within literal strings) are converted to single blanks. Whitespace characters adjacent to operator characters and special characters are also removed [RRef10].

Unlike languages like Java or C#, ooRexx uses the tilde character instead of a dot to invoke methods on objects (send messages to the object).

- Imposes few restrictions

  Programs can be coded in a format that emphasizes their structure, making them easier to read. For instance, a single instruction can span multiple lines. Since all characters of ooRexx statements are translated to uppercase, case sensitivity basically does not have to be observed, unless you use literal strings (strings delimited by quotation marks) for variable or label names. ooRexx also reduces the threat of name conflicts to a minimum as keywords are only reserved in context.

- Weak typing

  Variables do not have to be explicitly declared with a data type as data types are recognized from the context.

- Object-orientation

  ooRexx provides data encapsulation, polymorphism, an object class hierarchy, class-based inheritance of methods, and concurrency. It includes a number of useful base classes and allows you create new object classes of your own. [RRef10]

- Interpreted language

  Unlike non-interpreted languages, which compile program code into machine code before they can be run, ooRexx is run by an interpreter. The interpreter reads each line separately and runs it. That leads to faster debugging and reruns.

2.2 Functionality of ooRexxTry

`ooRexxTry` is not meant to be an IDE nor does it have all the bells and whistles of an editor. It is designed for experimentally testing code that you may not be ready to save as a file.

`ooRexxTry` uses its GUI for standard output instead of the command prompt. All `say` statements within your code are redirected to a specific area of the dialog. The same is true for any values returned by your code. Errors, information and returned values are also redirected to their specific area of the dialog. Standard input (keyboard input) required by `parse pull`, `lineIn` and `charIn` instructions still needs to be made at the command prompt. [Peed07]

Figure 1 shows the Graphical User Interface of the program. The GUI consists of two input areas for the main code and separate arguments and three output areas.

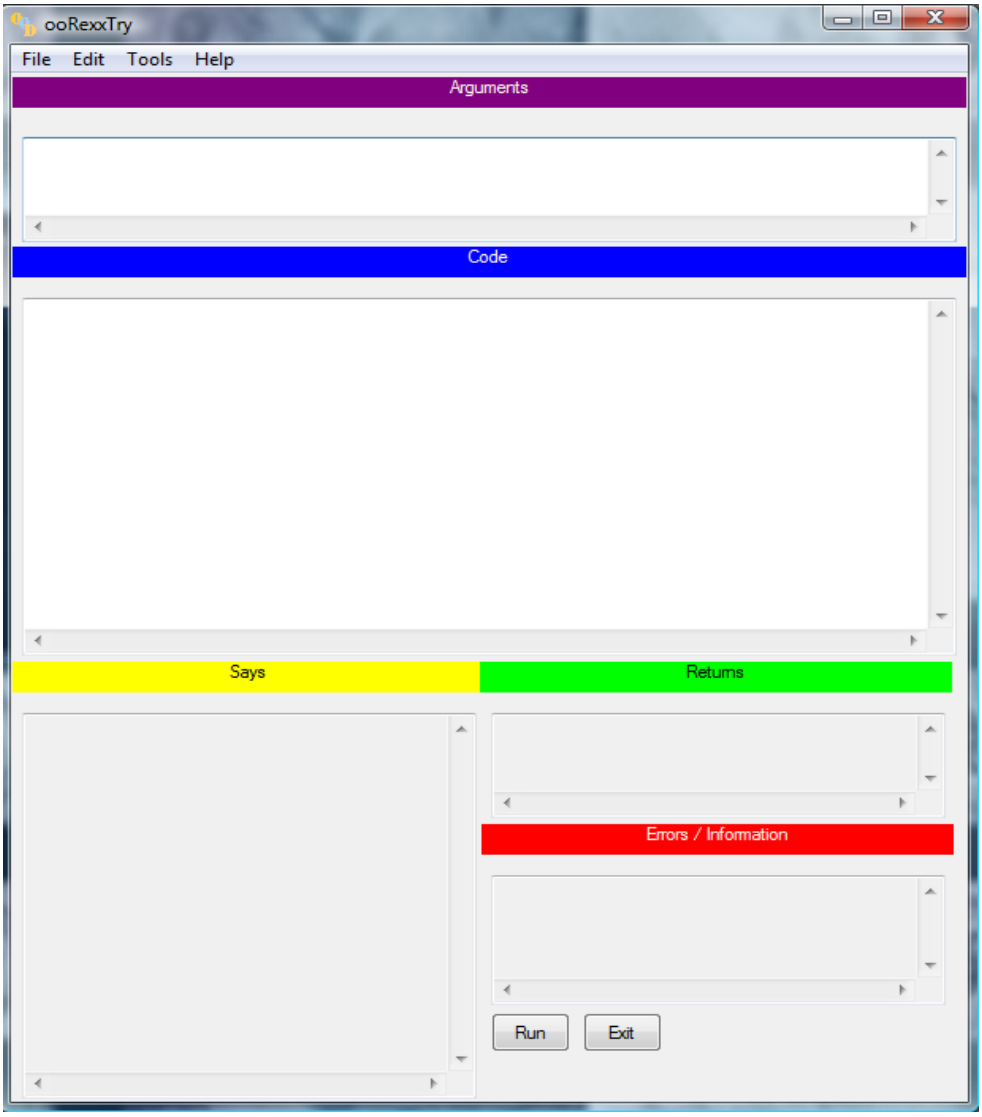

Figure 1: GUI of ooRexxTry

`ooRexxTry.rex` has the following characteristics [Peed07]:

- Size of the dialog is calculated on the basis of the screen resolution: minimum supported screen resolution is 800x600,
- .ini File used for start-up instructions (dialog position/size, …):
  Arguments must be entered on separate lines.
  Code input area can contain any valid ooRexx code (including directive instructions).
- Run code by using the menu option or the “Run” button
  If selection made only selected code fragment will be run
  Pressing enter key creates a new line but does not invoke the code execution procedure
- The menu bar provides commands for code saving/opening, font modification, accessing the system’s clipboard et cetera (cf. Figure 2)

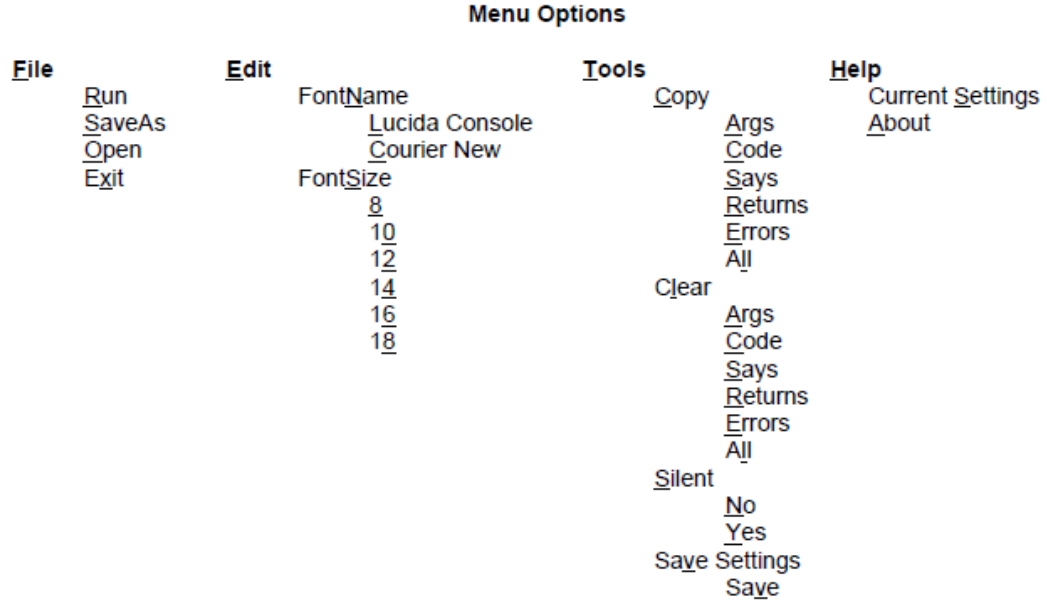

Figure 2: ooRexxTry Menu Structure

2.3 Library support

The original `ooRexxTry` leverages the functionality of the two classes `oodwin32.cls` and `winsystm.cls`, thus is tightly coupled to the Windows platform.

oodwin32.cls: implements the external library package ooDialog, which is a framework that aids ooRexx programmers in adding graphical elements to their Rexx progamms. The framework provides the base infrastructure, through a number of classes, that the programmer builds on to quickly produce Windows dialogs. [DRef10]

For instance, Figure 3 demonstrates the creation of the five input and output areas and the menu bar of the `ooRexxTry` GUI by Lee Peedin using `ooDialog`.

```
::class oort_dialog subclass userdialog inherit AdvancedControls Mes-
sageExtensions

::method DefineDialog
    expose u
    u = .dlgAreaU~new(self)
    if .nil \= u~lastError then
        call errorDialog u~lastError

---------- Arguments title & dialog area
    at = .dlgArea~new(0,0,u~w,10)
    self~addText(at~x,at~y,at~w,at~h,'Arguments','CENTER',17)
    ad = .dlgArea~new(0,at~y + 10,u~w,u~h('15%'))
    self~addEntryLine(12,'args_data',ad~x,ad~y,ad~w,ad~h, -
    'multiline hscroll vscroll')

---------- Code title & dialog area
    ct = .dlgArea~new(0,ad~y + ad~h,u~w,10)
    self~addText(ct~x,ct~y,ct~w,ct~h,'Code','CENTER',18)
    cd = .dlgArea~new(0,ct~y + ct~h,u~w,u~h('40%'))
    self~addEntryLine(13,'code_data',cd~x,cd~y,cd~w,cd~h, -
    'multiline hscroll vscroll')

---------- Says title & dialog area
    st = .dlgArea~new(0,cd~y + cd~h,u~w('50%'),10)
    self~addText(st~x,st~y,st~w,st~h,'Says','CENTER',19)
    sd = .dlgArea~new(0,st~y + st~h,u~w('50%'),u~h('43%'))
    self~addEntryLine(14,'say_data',sd~x,sd~y,sd~w,sd~h, -
    'notab readonly multiline hscroll vscroll')

---------- Returns title & dialog area
    rt = .dlgArea~new(sd~x + sd~w,cd~y + cd~h,u~w('50%'),10)
    self~addText(rt~x,rt~y,rt~w,rt~h,'Returns','CENTER',20)
    rd = .dlgArea~new(rt~x,st~y + st~h,u~w('50%'),u~h('15%'))
    self~addEntryLine(15,'results_data',rd~x,rd~y,rd~w,rd~h, -
    'notab readonly multiline hscroll vscroll')

---------- Errors/Information title & dialog area
    et = .dlgArea~new(rt~x,rd~y + rd~h,u~w('50%'),10)
```

```
    self~addText(et~x,et~y,et~w,et~h, -
    'Errors / Information','CENTER',21)
    ed = .dlgArea~new(rt~x,et~y + et~h,u~w('50%'),u~h('17%'))
    self~addEntryLine(16,'error_data',ed~x,ed~y,ed~w,ed~h, -
    'notab readonly multiline hscroll vscroll')

---------- Run & Exit buttons for easier execution
    self~AddButton(80,ed~x     ,ed~y + ed~h + 2,35,10,'&Run','RunIt')
    self~AddButton(81,ed~x + 40,ed~y + ed~h + 2,35,10, -
    'E&xit','Cancel')

----------
    self~createMenu
    self~AddPopupMenu('&File')
       self~addMenuItem('&Run'   ,22,     ,'RunIt')
       self~addMenuItem('&SaveAs',23,     ,'FileDialog')
       self~addMenuItem('&Open'  ,25,     ,'FileDialog')
       self~addMenuItem('E&xit'  ,24,'END','Cancel')

    self~addPopUpMenu('&Edit')
       self~addPopupMenu('Font&Name')
           self~addMenuItem('&Lucida Console',30,     -
           ,'onFontMenuClick')
           self~addMenuItem('&Courier New'   -
           ,31,'END','onFontMenuClick')
      self~addMenuSeparator
      self~addPopUpMenu('Font&Size','END')
           self~addMenuItem('&8' ,40,     ,'onFontMenuClick')
           self~addMenuItem('1&0',41,     ,'onFontMenuClick')
           self~addMenuItem('1&2',42,     ,'onFontMenuClick')
           self~addMenuItem('1&4',43,     ,'onFontMenuClick')
           self~addMenuItem('1&6',44,     ,'onFontMenuClick')
           self~addMenuItem('1&8',45,'END','onFontMenuClick')

   self~AddPopUpMenu('&Tools')
       self~addPopupMenu('&Copy')
           self~addMenuItem('&Args'   ,50,     ,'Clipboard')
           self~addMenuItem('&Code'   ,51,     ,'Clipboard')
           self~addMenuItem('&Says'   ,52,     ,'ClipBoard')
           self~addMenuItem('&Returns',53,     ,'ClipBoard')
           self~addMenuItem('&Errors' ,54,     ,'ClipBoard')
           self~addMenuItem('A&ll'    ,55,'END','ClipBoard')
           self~addMenuSeparator
       self~addPopupMenu('C&lear')
           self~addMenuItem('&Args'   ,60,     ,'ClearAll')
           self~addMenuItem('&Code'   ,61,     ,'ClearAll')
           self~addMenuItem('&Says'   ,62,     ,'ClearAll')
           self~addMenuItem('&Returns',63,     ,'ClearAll')
           self~addMenuItem('&Errors' ,64,     ,'ClearAll')
           self~addMenuItem('A&ll'    ,65,'END','ClearAll')
           self~addMenuSeparator
       self~addPopupMenu('&Silent')
           self~addMenuItem('&No'     ,66,     ,'Silent')
           self~addMenuItem('&Yes'    ,67,'END','Silent')
           self~addMenuSeparator
       self~addPopupMenu('Sa&ve Settings','END')
           self~addMenuItem('Sa&ve'   ,72,'END','SaveSettings')
   self~addPopupMenu('&Help','END')
       self~addMenuItem('Current &Settings',71,     ,'Settings')
       self~addMenuItem('&About'   ,70,'END','Help')
```

Figure 3: Input and Output Areas and Menu Bar Creation with ooDialog

winsystm.cls[1]: defines the `WindowsClipboard` class which provides methods to interact with a clipboard. Typically a clipboard is used to transfer data back and forth between different windows in a graphical user interface.

Although ooRexx is by definition platform independent the GUI enhanced version of `ooRexxTry` cannot run on an operation system different than Windows due to the aforementioned restrictions regarding component resources. To achieve full platform independence some other GUI toolkit has to be leveraged. Now here is the point where Java comes into play.

[1] [ERef10]

3 ooRexxTry.rxj[2] Concepts

This chapter provides an overview and evaluation of the three most popular Java GUI toolkits and points out which of them serves the transformation towards a platform independent version of `ooRexxTry` the best. What is more, it briefly describes the BSF4ooRexx framework which enables Java support for ooRexx and points out important functionality of BSF4ooRexx incorporated in `ooRexxTry.rxj`.

3.1 Java GUI toolkits

Java GUI widget toolkits consist of libraries which provide building blocks for the graphical user interface of a program. The major benefit is that such a GUI will run unchanged with basically the same appearance on every computer with a Java Runtime Environment installed.

AWT

AWT (the Abstract Window Toolkit) was the first Java GUI toolkit, introduced with JDK 1.0 as one component of the Sun Microsystems Java platform. It is a very simple tool kit with limited GUI components, layout managers, and events. This is because Sun Microsystems decided to use a lowest-common denominator (LCD) approach for AWT. Only GUI components defined for all Java host environments would be used. [FeigSA]

AWT depends on host GUI peer controls for building graphical components. Basically it is a wrapper around native container and controls like dialogs, buttons and menus.

Most modern Java based GUIs are built on Swing the “successor” of AWT. Nevertheless AWT supplies the indispensable event model.

[2] ooRexx files with Java support use the file extension .rxj instead of .rex.

Swing

Swing was the next generation GUI toolkit introduced by Sun in J2SE 1.2. Swing was developed in order to provide a richer set of GUI components than AWT. Swing GUI elements are 100% Java with no native code: instead of wrapping native GUI components, Swing draws its own components by using Java2D to call low level operating system drawing routines. [FeigSA]

Most Swing components (`JComponent` and its subclasses) are emulated in pure-Java code. This means that Swing is naturally portable across all hosts [FeigSA]. Swing has the same look and feel regardless of the underlying operating system. Anyway it is possible to simulate the host's look and feel.

Swing uses the term heavyweight for peer-based components and lightweight for emulated components. Many swing classes inherit their behavior from AWT super classes as illustrated in Figure 4.

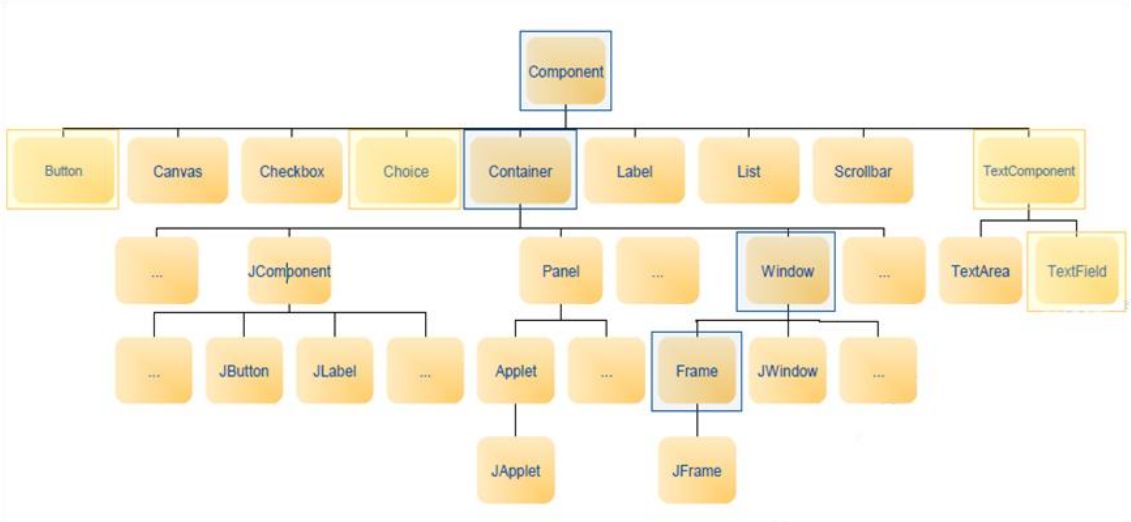

Figure 4: AWT-Swing Component Inheritance Hierarchy [Flat10b]

Swing components are not thread-safe. Invoking them from different threads risks thread interference or memory consistency errors. Java is designed to run most method invocations on realized visual components on a single thread, the event dispatch thread (EDT), and provides functionality for safely updating the graphical user interface. Tasks on the EDT must finish quickly. If a task is very computationally intensive it should not run on the EDT. Otherwise it may turn

the GUI unresponsive. GUI updates can be scheduled via `SwingUtilities.invokeLater`[3].

SWT

The Standard Widget Toolkit (SWT) was originally developed by IBM and is now maintained by the Eclipse Foundation and part of the Eclipse IDE. It does not come with a standard Java runtime or SDK but can be download separately from http://www.eclipse.org/swt/.

SWT is a lot like AWT in that it is based on a peer implementation. It overcomes the LCD problem faced by AWT by defining a set of controls adequate to make most office applications or developer tools. In cases where native platform GUI libraries do not support the functionality required for SWT, it implements (emulates) its own GUI code in Java, similar to Swing. In essence, SWT have a host look and feel and host performance. [FeigSA]

The bottom line is that the Swing framework is most suitable for the platform independence approach of `ooRexxTry.rxj` since its host peer dependence is minimal.

3.2 Leverage Java with BSF4ooRexx

BSF4ooRexx builds on the Bean Scripting Framework (BSF) maintained by the Apache Software Foundation. BSF is a set of Java classes which provides scripting language support within Java applications, and access to Java objects and methods from scripting languages. The current release version 2.4.0 of BSF builds on an API originally developed by IBM. BSF4ooRexx is supported with its own BSF engine and achieves to enable the use of ooRexx functionality in Java and vice versa.

---

[3] http://download.oracle.com/javase/6/docs/api/javax/swing/SwingUtilities.html#invokeLater%28java.lang.Runnable%29

In order to facilitate the interfacing to Java from invoked Rexx programs quite a comprehensive set of functions and sub functions have been devised. As a result of this endeavor all the functionality of Java as implemented in its class hierarchies can be regarded as a huge external Rexx function library which can be directly used! [Flat04]

The BSF.cls is the heart of the BSF4ooRexx package. It creates an object-oriented wrapper for allowing transparently accessing the Java objects registered with BSF.

This opens up the following possibilities [Flat01]:

- Java classes may get imported into the Object Rexx environment and can thereafter be used as if they were Object Rexx classes,
- creating instances of the imported Java classes creates Object Rexx objects serving as proxies for their Java counterparts,
- sending Object Rexx messages (terms right to the message operator “twiddle”, which is the character tilde) to these proxy objects invokes the methods on the Java side automatically.

RexxProxy

In order to work with Java GUI building blocks we need essentially two BSF functions, `.bsf~new` and `BsfCreateRexxProxy`. `.bsf~new` takes the fully qualified Java class name as first argument and Java class dependent constructor arguments subsequently. For instance,

```
.bsf~new("javax.swing.JTextField", '', 10)
```

returns the instance of the `JTextField` class with blank content and of 10 columns length.

The interactivity of a GUI is realized by the controller holding various event listeners. But, e.g. an `ActionListener` interface needs its abstract methods to be implemented in the context of the program. Hence, creating an instance of an `ActionListener` and invoking methods on it is not enough. It is necessary to implement the needed interface method(s) with an ooRexx genuine class and

then give an instance of it to `BsfCreateRexxProxy` which creates and returns a proxy object that can be utilized as event listener.

Figure 5 demonstrates the use of the class `KeyObserver` as a proxy to listen for `keyPressed` events. The first argument of `BsfCreateRexxProxy` is the instance of the proxy class. The second argument is an optional directory which in this case holds important objects the proxy class needs access to. The last argument is the name of the Java interface. Note, that the interface can be addressed by the local name `.KeyListener` because it was previously imported via

```
call bsf.importClass 'java.awt.event.KeyListener', 'KeyListener'
```

Every time Java notices a key event on the specified area the event is propagated to the `RexxProxy` object.

```
hlist~addKeyListener(BsfCreateRexxProxy(.KeyObserver~new, -
userData, .KeyListener))

::class KeyObserver
::method keyPressed
  use arg eventObject, slotDir

  userData = slotDir~userData
  comp = eventObject~getComponent
  key = eventObject~getKeyCode
  if key == .KeyEvent~VK_ENTER then
    do
     …

    end
```

Figure 5: Implement Java Interface Using BsfCreateRexxProxy

4 ooRexxTry.rxj in Detail

This section covers a complete list of Java classes utilized for the implementation of the platform independent version of `ooRexxTry`. What is more, it describes core functionality, highlights selected alterations and discusses interesting improvements and extensions.

The GUI of `ooRexxTry.rxj` features an additional input area which substitutes the command-line when it comes to user terminal input in case of "`parse pull`" instructions and the like. This input area shown in figure 6 is connected to a stream which serves for primary inputs[4].

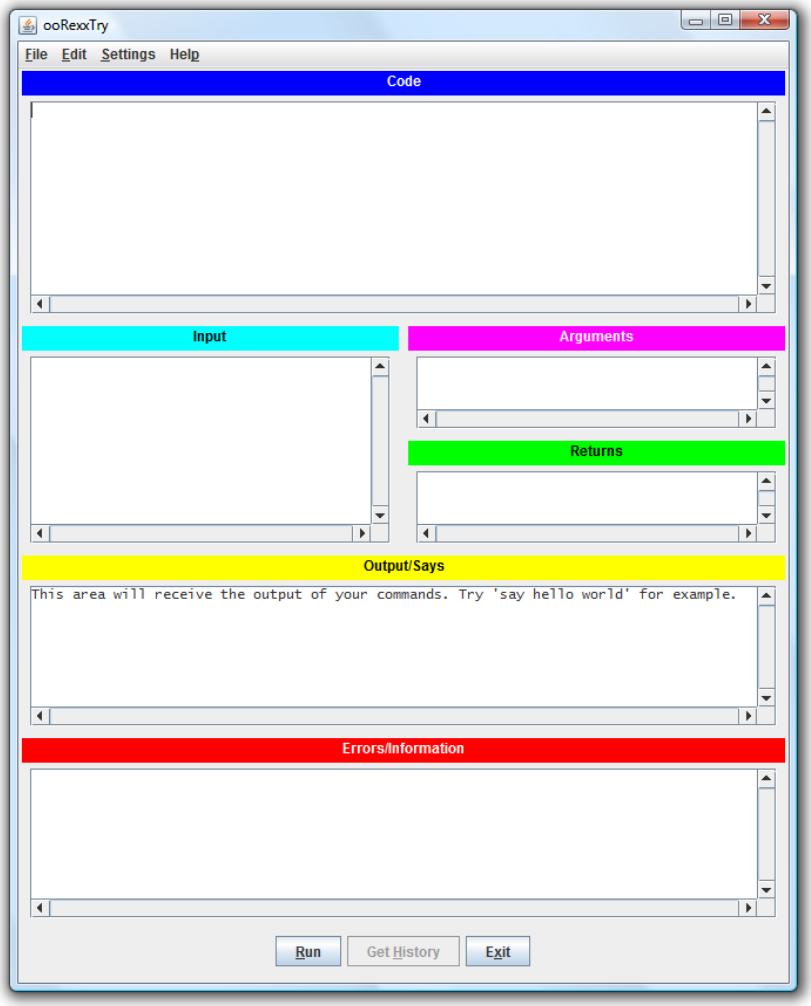

Figure 6: ooRexxTry.rxj GUI featuring "Input" area

[4] For further information on standard input see chapter 4.2.4

4.1 Leveraged Java Classes

In the course of reproducing the functionality of the original program at an operation system neutral level the classes and their methods from basically four different packages were used.

The earlier mentioned java.awt package is mostly leveraged for layout tasks and event handling (cf. table 1). The GUI the user interacts with is essentially built from the `javax.swing` package with visual components like `JFrame`, `JDialog`, `JList` and `JTextArea` (cf. table 2).

Package: java.awt [JSEAPI]

Java Class	Description/Usage in Program
java.awt.BorderLayout	Lays out a container, arranging and resizing its components to fit in five regions: north, south, east, west, and center. Each region may contain no more than one component. Most outer layout on the GUI.
java.awt.Color	The `Color` class is used to encapsulate colors in the default sRGB color space or colors in arbitrary color spaces.
java.awt.Cursor	A class to encapsulate the bitmap representation of the mouse cursor. Transform cursor to visualize busy state.
java.awt.datatransfer.StringSelection	Creates a `Transferable` capable of transferring the specified `String`.
java.awt.Dimension	The `Dimension` class encapsulates the width and height of a component in a single object.
java.awt.event.ActionListener	The listener interface for receiving action events. The class that is interested in processing an action event implements this interface, and the object created with that class is registered with a component, using the component's `addActionListener` method. When the action event occurs, that object's `actionPerformed` method is invoked.
java.awt.event.WindowListener	The listener interface for receiving window events. cf. ActionListener

java.awt.FlowLayout	A flow layout arranges components in a left-to-right flow, much like words of text in a paragraph. Used on the dialog windows.
java.awt.Font	The `Font` class represents fonts, which are used to render text in a visible way.
java.awt.Frame	A `Frame` is a top-level window with a title and a border. In the context of the program only used to get access to its constant values.
java.awt.GridBagConstraints	The `GridBagConstraints` class specifies constraints for components that are laid out using the `GridBagLayout` class.
java.awt.GridBagLayout	The `GridBagLayout` class is a flexible layout manager that aligns components vertically and horizontally, without requiring that the components be of the same size. Used for most of the components.
java.awt.Insets	It specifies the space that a container must leave at each of its edges.
java.awt.KeyEvent	This low-level event is generated by a component object (such as a text field) when a key is pressed, released, or typed. The event is passed to every `KeyListener` or `KeyAdapter` object which registered to receive such events using the component's `addKeyListener` method.
java.awt.KeyListener	The listener interface for receiving keyboard events (keystrokes). Used for the “Input” area and the “History” dialog.
java.awt.MouseListener	This class processes mouse events. Used to be aware of double clicks at the “History” dialog.
java.awt.Toolkit	Binds various platform-independent components to particular native toolkit implementations. Used to get the screen size, to get access to the system clipboard and used for shortcut making of the host system.

Table 1: Classes from the java.awt Package Utilized in ooRexxTry.rxj

Package: javax.swing [JSEAPI]

Java Class	Description/Usage in Program
javax.swing.ButtonGroup	This class is used to create a multiple-exclusion scope for a set of buttons. Arranges the monitor configuration buttons.
javax.swing.DefaultListModel	Provides methods components like JList use to get the value of each cell in a list and the length of the list.
javax.swing.filechooser.FileFilter	A `FileFilter`, once implemented, can be set on a `JFileChooser` to keep unwanted files from appearing in the directory listing. Implemented with a custom file filter class in ooRexx.
javax.swing.JButton	An implementation of a "push" button.
javax.swing.JCheckBoxMenuItem	This class represents a check box that can be included in a menu.
javax.swing.JComponent	The base class for all Swing components except top-level containers. Used to allow certain arrow key navigation when the “History” dialog is focused.
javax.swing.JDialog	The main class for creating a dialog window. Used to display meta data and the code execution history.
javax.swing.JFileChooser	`JFileChooser` provides a simple mechanism for the user to choose a file. Used for opening/saving data from/to an external file.
javax.swing.JFrame	An extended version of `java.awt.Frame` that adds support for the JFC/Swing component architecture. Unlike a `Frame`, a `JFrame` has some notion of how to respond when the user attempts to close the window. Main container which holds most of the GUI components.
javax.swing.JLabel	A display area for a short text string or an image, or both.
javax.swing.JList	A component that allows the user to select one or more objects from a list. A separate model, `ListModel`, represents the contents of the list. Visual container of the code execution history.
javax.swing.JMenu	An implementation of a menu -- a popup window containing `JMenuItem`s that is displayed when the user selects an item on the `JMenuBar`.
javax.swing.JMenuBar	You add `JMenu` objects to the menu bar to construct a menu.

javax.swing.JMenuItem	A menu item is essentially a button sitting in a list.
javax.swing.JPanel	`JPanel` is a generic lightweight container.
javax.swing.JRadioButton	Used with a ButtonGroup object to create a group of buttons in which only one button at a time can be selected.
javax.swing.JScrollPane	Provides a scrollable view of a lightweight component. Most of the input and output areas are put into one.
javax.swing.JTabbedPane	A component that lets the user switch between a group of components by clicking on a tab with a given title and/or icon. See Settings -> Current Settings.
javax.swing.JTextArea	A `JTextArea` is a multi-line area that displays plain text. Represents the main input and output areas.
javax.swing.JTextField	`JTextField` is a lightweight component that allows the editing of a single line of text.
javax.swing.ListSelectionModel	Represents the current state of the selection for any of the components that display a list of values with stable indices.
javax.swing.KeyStroke	A KeyStroke represents a key action on the keyboard, or equivalent input device. KeyStrokes can correspond to only a press or release of a particular key. Used to enable Shortcuts.
javax.swing.UIManager	This class keeps track of the current look and feel and its defaults. Ensures system familiar look and feel of the GUI.

Table 2: Classes from the javax.swing Package Utilized in ooRexxTry.rxj

The use of the `Runnable` and the `Thread` class from `java.lang` package allow simultaneously handling the code execution and socket connections without making the GUI unresponsive (cf. table 3).

Package: java.lang [JSEAPI]

Java Class	Description/Usage in Program
Runnable	The `Runnable` interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called `run`.

Thread	A *thread* is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently. Starting the thread invokes the `run` method of the `Runnable` object.

Table 3: Classes from the java.lang Package Utilized in ooRexxTry.rxj

The `java.io` package provides functionality for reading and writing data when communicating with a client via socket connection (cf. table 4).

Package: java.io [JSEAPI]

Java Class	Description/Usage in Program
BufferedReader	Read text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.
File	An abstract representation of file and directory pathnames. Used to set the current path.
InputStreamReader	An InputStreamReader is a bridge from byte streams to character streams. Each invocation of the read() method may cause one or more bytes to be read from the underlying byte-input stream. Used to read inbound data from socket connection
PrintWriter	Print formatted representations of objects to a text-output stream. Used to write outbound data to a socket communication partner.

Table 4: Classes from the java.io Package Utilized in ooRexxTry.rxj

4.2 Important Functionality and Classes

The .rxj version written with the support of Java features two major extensions to the original `ooRexxTry.rex`. The first one is a code execution history which remembers previously run code and displays code items at a dialog where the user can select one to restore. The second feature allows the user to dynamically switch the destination of the three monitors of ooRexx. As a result, the program can directly parse input from a file source or write output or error related data automatically to a file instead of the assigned GUI areas.

`ooRexxTry.rxj` also incorporates a very exciting undocumented feature of BSF4ooRexx, the possibility to specialize Java classes in ooRexx. But first let's have a close look at the code execution procedure and how `ooRexxTry.rxj` manages access to certain configuration data.

4.2.1 Execute Code

To start the execution of some code simply click the "`Run`" button or click menu `File > Run` or use `ALT-R` or `CTRL-R` (on Windows) or `META-R` (on Mac), if you prefer shortcuts (cf. figure 7).

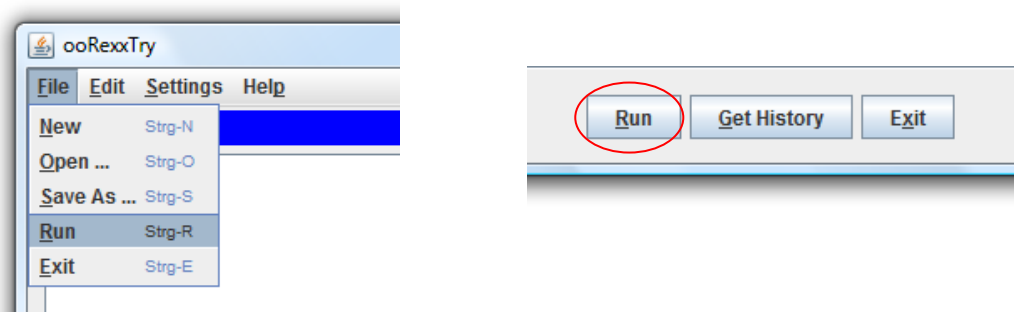

Figure 7: Access to Code Execution

The program first parses the input from the "`Arguments`" area and from the "`Code`" area and makes an array out of it while observing syntactical correct arguments and looking for directive instructions (cf. C1 in the appendix). It is necessary to structure the initial arguments string in way that gives access to every single argument spanning just one line of the string. This arguments array allows identifying syntactically incorrect arguments on interpretation and gives

access to them. If no exception was thrown, the member items of the arguments array are passed as arguments to the `run` method of the `Object` class (super class of the Executor class) later on. The code array follows the same principle the arguments array does. The subsequent step is to create an instance of the `ExecuteThread` class with the formatted code and arguments handed over. The `ExecuteThread` class implements the `java.lang.Runnable` interface. In order to run the literal code execution in a separate thread – not interrupting GUI operations – the `Runnable` objects gets passed to the instance of a new thread.

On thread start (calling the `run` method of the `ExecuteThread` class) the `Executor` class gets instantiated and creates an executable representation of the code (cf. C2 in the appendix).

Next the program calls the `run` method of the `Executor` class to run the created code representation. This is accomplished in a final step by invoking the earlier mentioned `run` method of the `Object` class with the code and arguments passed.

If an error occurs during code execution, the program raises a condition trapped by “RunSyntax” (cf. C2 in the appendix.) Otherwise the successfully run code is admitted to the history by calling the `storeData` method of the `Store` class. (See chapter 4.2.3 for more information on the history.) Possibly returned values are appended to the “`Returns`” area on the GUI. The results of “`say`”, “`charout`” or “`lineout`” instructions are appended to the “`Output/Says`” area on the GUI by the `GUIOutputStream` (cf. figure 8).

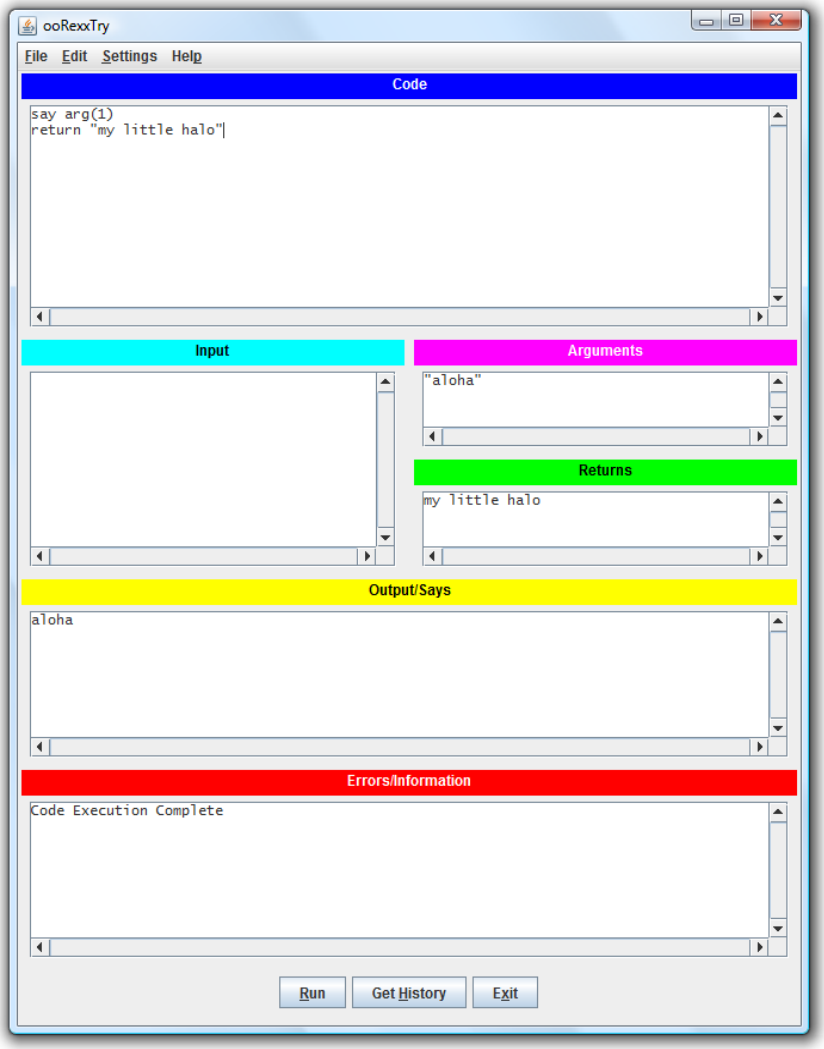

Figure 8: Result of Code Execution

Error Handling

`ooRexxTry` contains three condition traps (`ArgSyntax`, `ExecSyntax`, `RunSyntax`) throwing exceptions in case of a syntax-based or logical error occurred. Figure 9 demonstrates the result of a syntax-based error trapped by `ExecSyntax`.

An appropriate message is appended at the "`Errors/Information`" area with detailed information of the error like the error code/number, error type and source code line involved (depending on the type of error). What is more, the faulty code line gets highlighted on the GUI.

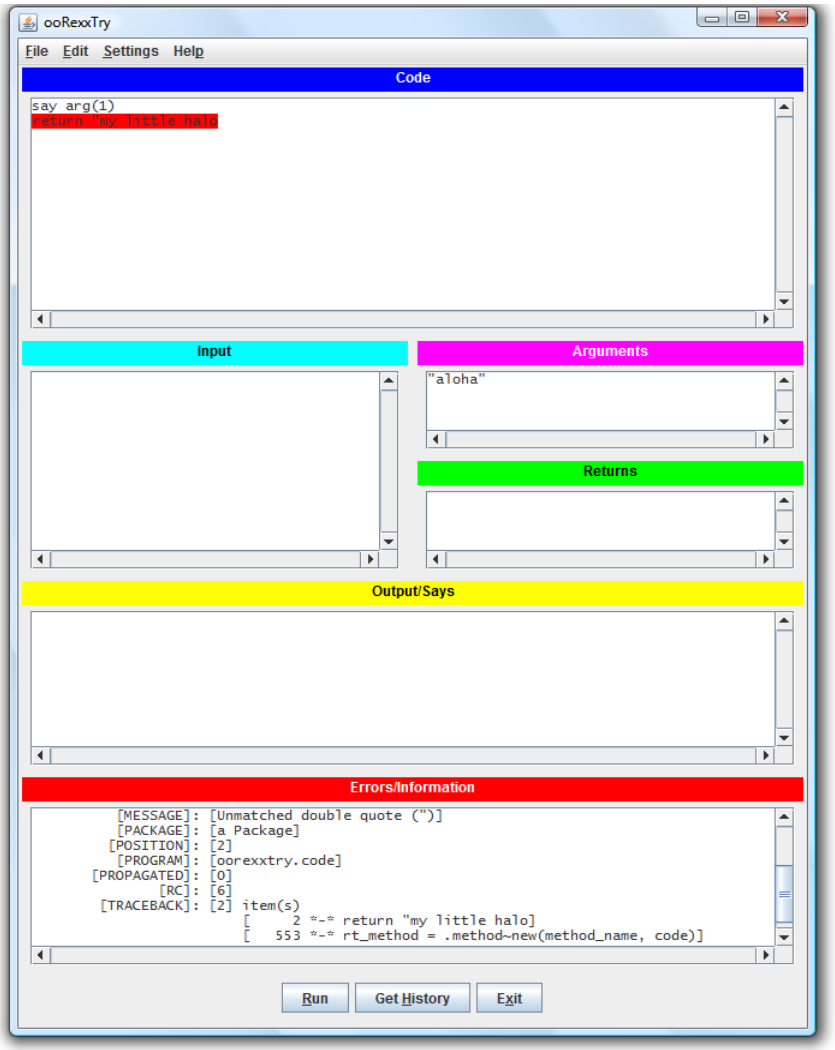

Figure 9: Trapped Syntax Error

4.2.2 Configuration Data Management

The original `ooRexxTry` uses the function `SysIni` from the `RexxUtil` package for saving certain variables. However `SysIni` requires the use of a Windows OS. Fortunately ooRexx offers the `Properties` class for processing application option values. Figure 10 demonstrates the use of `SysIni` for loading the font name, font size and the current value of silent into the program while figure 11 shows in extracts the entirely platform independent way.

```
-- If the .ini file is present, use it for font/silent variables
    .local~fontname = SysIni('oorexxtry.ini','oorexxtry','fn')
    .local~fontsize = SysIni('oorexxtry.ini','oorexxtry','fs')
    .local~silent   = SysIni('oorexxtry.ini','oorexxtry','sl')

-- Else use some defaults
    if .fontname = 'ERROR:' | .useDefault then
        .local~fontname = 'Lucida Console'
    if .fontsize = 'ERROR:' | .useDefault then
        .local~fontsize = 12
    if .silent   = 'ERROR:' | .useDefault then
        .local~silent   = .false
```

Figure 10: Using SysIni to Load Configuration Data

`ooRexxTry.rxj` offers a greater variety of fonts and incorporates 17 modifiable settings including a changeable look & feel. The default font is chosen depending on whether you use a Windows, Linux or Mac operating system. Additional fonts are provided after the program checked if certain fonts are installed on your system.

```
-- Load configuration data from the .rc file if present

if .mm.dir~useDefault then
  do
   if .mm.dir~opSys="W" then -
   .local~mm.dir~fontname = 'Lucida Console'
   else if .mm.dir~opSys="L" then .local~mm.dir~fontname = 'Monospace'
   else if .mm.dir~opSys="M" then .local~mm.dir~fontname = 'Monaco'
   else .local~mm.dir~fontname = 'Courier'
   .local~mm.dir~fontsize = 12
   .local~mm.dir~historySize = 20
   .local~mm.dir~inputMonitorDestination = 'GUIInputStream'
   .local~mm.dir~outputMonitorDestination = 'GUIOutputStream'
   .local~mm.dir~errorMonitorDestination = 'GUIErrorStream'
   .local~mm.dir~host = 'localhost'
   .local~mm.dir~port = 8888
   .local~mm.dir~enableInputSocket = .false
   .local~mm.dir~enableOutputSocket = .false
   .local~mm.dir~enableErrorSocket = .false
   .local~mm.dir~useSystemLookAndFeel = .false
   .local~mm.dir~silent = .false
   return
  end

.local~mm.dir~props = .properties~load("ooRexxTry.rc")
props = .mm.dir~props
.local~mm.dir~fontname = props~getProperty("FontName")
if .mm.dir~fontname == .nil then
  do
   if .mm.dir~opSys="W" then .local~mm.dir~fontname = 'Lucida Console'
```

```
   else if .mm.dir~opSys="L" then .local~mm.dir~fontname = 'Monospace'
   else if .mm.dir~opSys="M" then .local~mm.dir~fontname = 'Monaco'
   else .local~mm.dir~fontname = 'Courier'
  end
.local~mm.dir~fontsize = props~getProperty("FontSize", 12)
.local~mm.dir~historySize = props~getProperty("HistorySize", 20)
.local~mm.dir~inputMonitorDestination = -
props~getProperty("Input_Monitor_Destination", 'GUIInputStream')
.local~mm.dir~outputMonitorDestination = -
props~getProperty("Output_Monitor_Destination", 'GUIOutputStream' )
.local~mm.dir~errorMonitorDestination = -
props~getProperty("Error_Monitor_Destination", 'GUIErrorStream')
.local~mm.dir~host = props~getProperty("Host", 'localhost')
.local~mm.dir~port = props~getProperty("Port", 8888)
.local~mm.dir~enableInputSocket = -
props~getProperty("Enable_Input_Stream_On_Socket", .false)
.local~mm.dir~enableOutputSocket = -
props~getProperty("Enable_Output_Stream_On_Socket", .false)
.local~mm.dir~enableErrorSocket = -
props~getProperty("Enable_Error_Stream_On_Socket", .false)
.local~mm.dir~useSystemLookAndFeel = -
props~getProperty("Use_System_Look_And_Feel", .false)
.local~mm.dir~silent = props~getProperty("silentMode", .false)

...
```

Figure 11: Using ooRexx's Property Class to Load Configuration Data

Figure 12 stores the frame specifications, the font name, font size and the like at `ooRexxTry.rc`.

```
::class SaveSettings
::method actionPerformed

    self~saveSettings

::method saveSettings

    -- Write out the configuration data like
    -- size,position,fontname,fontsize, & silent to the .rc file
    frame = .mm.dir~frame
    props = .mm.dir~props
    if frame~getExtendedState \= .Frame~iconified & -
    frame~getExtendedState \= .Frame~maximized_both then
        do
            props~setProperty("Xpos",frame~getX)
            props~setProperty("Ypos",frame~getY)
            props~setProperty("Width",frame~getWidth)
            props~setProperty("Height",frame~getHeight)
        end
    props~setProperty("FontName",.mm.dir~fontname)
    props~setProperty("FontSize",.mm.dir~fontsize)
    props~setProperty("HistorySize",.mm.dir~historySize)
    props~setProperty("Input_Monitor_Destination", -
    .mm.dir~inputMonitorDestination)
    props~setProperty("Output_Monitor_Destination", -
```

```
    .mm.dir~outputMonitorDestination)
    props~setProperty("Error_Monitor_Destination", -
    .mm.dir~errorMonitorDestination)
    props~setProperty("Host",.mm.dir~host)
    props~setProperty("Port",.mm.dir~port)
    if .mm.dir~enableInputSocket then -
    props~setProperty("Enable_Input_Stream_On_Socket","yes")
    else props~setProperty("Enable_Input_Stream_On_Socket","no")
    if .mm.dir~enableOutputSocket then -
    props~setProperty("Enable_Output_Stream_On_Socket","yes")
    else props~setProperty("Enable_Output_Stream_On_Socket","no")
    if .mm.dir~enableErrorSocket then -
    props~setProperty("Enable_Error_Stream_On_Socket","yes")
    else props~setProperty("Enable_Error_Stream_On_Socket","no")
    if .mm.dir~useSystemLookAndFeel then -
    props~setProperty("Use_System_Look_And_Feel","yes")
    else props~setProperty("Use_System_Look_And_Feel","no")
    if .mm.dir~silent then props~setProperty("silentMode","yes")
    else props~setProperty("silentMode","no")
    props~save("ooRexxTry.rc")
    .mm.dir~errorsArea~setText("Settings Saved To ooRexxTry.rc")
    return
```

Figure 12: SaveSettings Class Stores Configuration Data

The file extension .rc refers to a “settings” file containing startup instructions for an application program. Nevertheless any other text file can be used. All properties are saved as name-value-pairs (cf. figure 13).

```
Xpos=0
HistorySize=20
Height=840
Ypos=0
Enable_Error_Stream_On_Socket=no
Error_Monitor_Destination=GUIErrorStream
Input_Monitor_Destination=GUIInputStream
Width=672
FontName=Lucida Console
FontSize=12
Output_Monitor_Destination=GUIOutputStream
Host=localhost
Port=8888
Enable_Input_Stream_On_Socket=no
Enable_Output_Stream_On_Socket=no
Use_System_Look_And_Feel=no
silentMode=no
```

Figure 13: Content of ooRexxTry.rc

4.2.3 Code Execution History

The Code Execution History – one of the extensions to the original `ooRexxTry` – is a convenient way to get access to previously run code.

This tool basically consists of a circular queue and a list model that together constitute the logic and a `JList` where each item gets rendered in as a `JTextArea`. The `Store` class from figure 15 creates a circular queue. An object is appended at the end of the queue and may replace earlier entries.

Every successfully run command queues the code input along with a time stamp. Subsequently the model of the `JList` gets cleared and refreshed with the current queue elements.

The size of the history can be adjusted at any time via `Settings -> Set History Capacity …` (cf. figure 14)

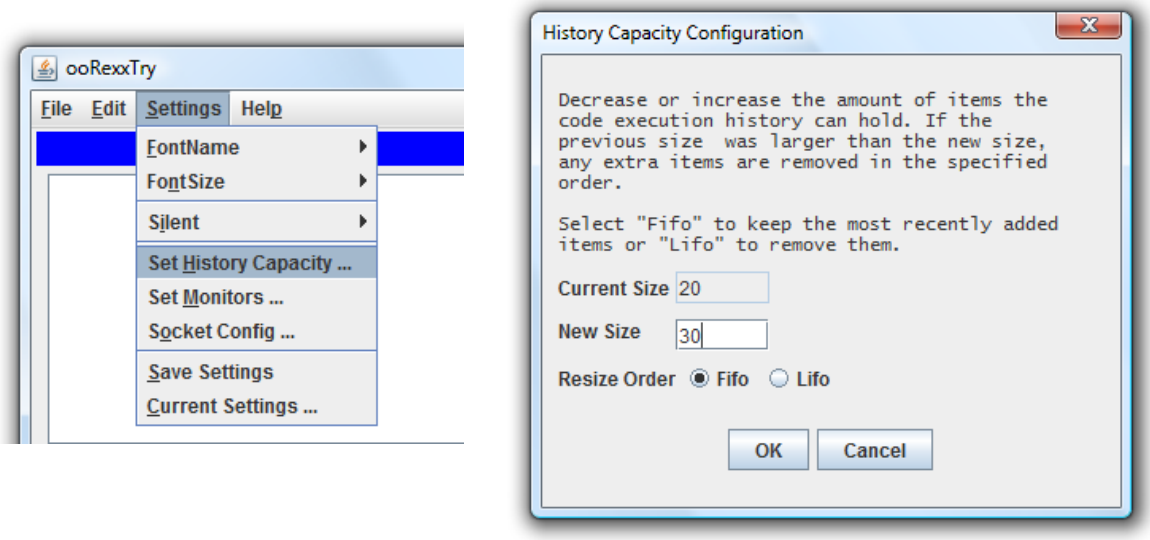

Figure 14: Access to History Capacity Configuration and Respective Dialog

The list will be extended or shortened observing the resize order specified. Subsequently the `makeQueue` method of the Store class is called which resizes the circular queue and updates the model of the `JList` (cf. `else` block of `makeQueue` method in figure 15)

```
/*Creates a circular queue and stores code execution data from the
JList's list model in it.*/
::class Store
::method makeQueue
   use arg items, order
-- Storing object for x elements (modifiable at any time);
-- newly items get inserted at the end of the queue
-- and replace earlier entries
   if \self~circq~isInstanceOf(.circularQueue) then  -
   self~circq = .circularQueue~new(items)

   else
     do
       listmodel = .mm.dir~listmodel
       listmodel~clear
-- Resize storing object according to new settings
       self~circq~resize(items, order)
       circqarray = self~circq~makeArray('F')
       do i=1 to circqarray~size
          listmodel~addElement(circqarray~at(i))
       end
     end

::method circq attribute

::method storeData
    use arg data

    listmodel = .mm.dir~listmodel
    item = time()||'0a'x||'-
'~copies(40)||'0a'x||data||'0a'x||'='~copies(40)  -- time stamp
    self~circq~queue(item)       -- Adds item at the end of the queue
-- Creates an array out of the queue elements
    circqarray = self~circq~makeArray('F')

    /* Clear list model and fill it with the current items of the cir-
cular queue*/
    listmodel~clear
    do i=1 to circqarray~size
        listmodel~addElement(circqarray~at(i))
    end
    .mm.dir~historybutton~setEnabled(.true)
```

Figure 15: Store Class Deposits Executed Code

Clicking on the "`Get History`" Button as visualized in figure 16 opens a dialog with all recently run code items (elements of the `JList`).

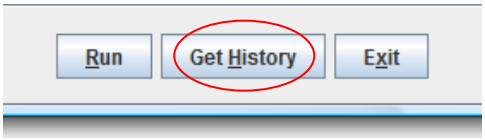

Figure 16: Access to History Dialog

By default a `JList` renders all elements from its list model with `JLabels`. A `JLabel` is only capable of displaying its content in a single line. This is obviously not suitable for displaying code. So the default single-line `JLabels` are replaced by multi-line `JTextAreas`. This is accomplished by the `CellRenderer` class, which implements the `ListCellRenderer` interface and returns a `JTextArea` as visual component for list items as shown in figure 17 and 18.

```
/*Custom list cell renderer; renders the elements from the listmodel
with seperate JTextAreas in a JList on the screen*/
::class CellRenderer
::method getListCellRendererComponent
        use arg list, value, index, isSelected, cellHasFocus

    listarea = .JTextArea~new
    listarea~setFont(.Font~new(.fontname, .Font~plain, "12"))
    listarea~setText(value~makeString)
    listarea~setLineWrap(.true)

    if isSelected = .true then
        do
            listarea~setBackground(listarea~getSelectionColor)
            listarea~setForeground(.Color~white)
            .rb~setEnabled(.true)
        end
    laScrollPane = .JScrollPane~new(listarea, -
    .JScrollPane~vertical_scrollbar_as_needed, -
    .JScrollPane~horizontal_scrollbar_never)
    return laScrollPane
```

Figure 17: CellRenderer Class

With the Code History Dialog from figure 18 the user can skim through previous code fragments and select what should be restored.

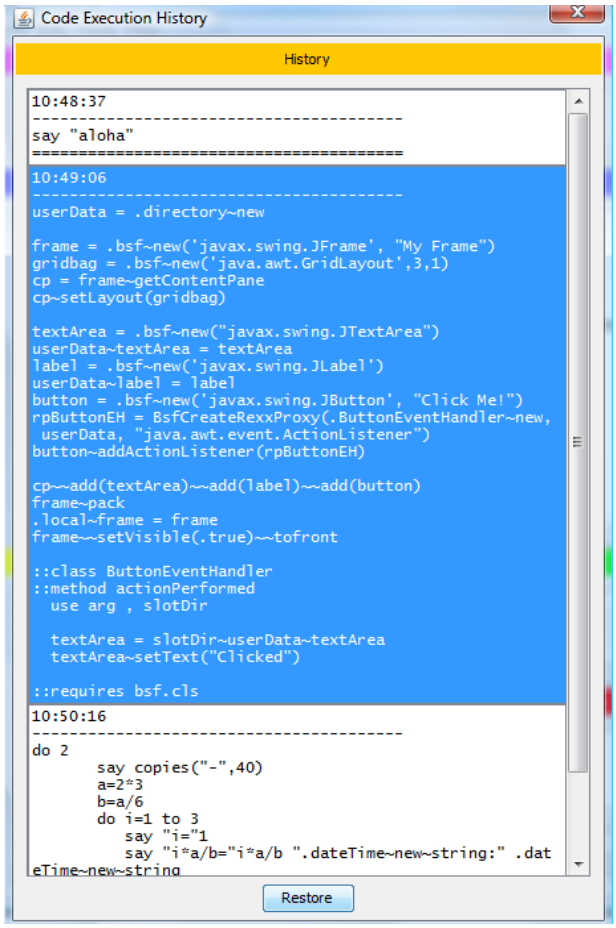

Figure 18: Code History Dialog

The `restoreData` method from the `Restore` class is called when an item from the list is selected and the "`Restore`" button is clicked or the user double clicks on the selected item or presses `ENTER`. After some formatting the code is restored to the GUI's "`Code`" area (cf. figure 19).

```
/*Retrieve code fragments from the execution histroy and display them
at the code input area again.*/
::class Restore
::method actionPerformed
    use arg , slotDir

    self~restoreData(slotDir~userData)

::method restoreData
    use arg userData

    hlist = userData~hlist
    listitem = hlist~getSelectedValue
    strippeditem = listitem~strip(, '=')
    timestamp = strippeditem~substr(1,8)
-- Modify the title bar to reflect the version currently working with
    .mm.dir~frame~setTitle(.mm.dir~title|| -
    " --- Code Version of "||timestamp)
    code_string = strippeditem~substr(51)
```

```
    code_string_alt = code_string~substr(1, code_string~length-1)
    .mm.dir~codeArea~setText(code_string_alt)
    .mm.dir~errorsArea~setText("Previous Code Restored")
```

Figure 19: Restore Class Retrieves Previously Executed Code

4.2.4 Monitor configuration

Another new feature of `ooRexxTry.rxj` is the possibility to configure the `.input`, `.output` and the `.error` monitor by changing the destination of each one. With the original `ooRexxTry.rex` the monitors read/write data from/to their standard streams which are `.stdin`, `.stdout` and `.stderr`. For instance, the result of a `say` statement is displayed at the command line, provided that `.stdout` is used. (In fact, `oorexxTry.rex` switches the output monitor each time `.stdout` was invoked to display output in the "`Says`" area.)

Now, `ooRexxTry.rxj` makes the command prompt obsolete when it comes to program input and output. All three monitors are connected via custom streams to respective frame areas. For instance, `.error` writes error related data (including trace protocols) to the `GUIErrorStream` that displays such data at the "Errors/Information" area of the GUI as a result.

Figure 20 illustrates in extracts the `GUIInputStream` class, responsible for managing keyboard input.

```
/* A stream that simulates the behavior of the standard input stream
and thus substitutes it
   as destination object of the .input monitor.
   Allows to gather keyboard input at the "input" area of the GUI in-
stead of the command prompt.*/
::class GUIInputStream subclass stream
::method init
-- control variable; set to .true when the Return key was pressed
   expose vk_enterPressed
   vk_enterPressed = .false

::method setControlVar
   expose vk_enterPressed
   use arg cv

   vk_enterPressed = cv
```

```
::method lineIn
    expose vk_enterPressed
    use arg line, count

    -- look for bad arguments
    if \line~equals('LINE') then raise syntax 93.958
    if count == 0 then return '0a'x
    if count < 0 then raise syntax 88.907 array -
    (2, 0, 4294967295, count)
    inputArea = .mm.dir~inputArea
    if count == 1 | count~equals('COUNT') then
      if .mm.dir~clientConnected & .mm.dir~enableInputSocket then
        do
          .mm.dir~socketInputRequired = .true
          guard on when vk_enterPressed \= .false
          input = inputArea~getText
          inputArea~setText('')
          .mm.dir~socketInputRequired = .false
          self~setControlVar(.false)
          return input
        end
      else
        do
          text = inputArea~getText
          if text~length > 0 then
            do
              input = text
              inputArea~setText('')
              return input
            end
          else
            do
-- move the focus for keyboard input to the "input" area
              inputArea~~setEditable(.true)~~requestFocus
-- Halt program flow until control variable is set to .true
              guard on when vk_enterPressed \= .false
-- Read the text content from the "input" area
              input = inputArea~getText
              inputArea~setText('')
              self~setControlVar(.false)
              inputArea~setEditable(.false)
              return input
           end
        end
   else raise syntax 93.0
```

Figure 20: GUIInputStream Class Substituting the Standard Input Monitor

In order to simulate the .stdin stream and the behavior of the command prompt in the course of keyboard input, it is necessary to overwrite the charIn and lineIn methods of the stream class and to incorporate some mechanism to temporary halt the program. For this purpose, the program uses a guard on instruction. Now, guard has great application spectrum but in the context of the

`GUIInputStream` it is used to halt the program flow until the user has made some input at the “Input” area and completed it with the `ENTER` key.

But with `ooRexxTry.rxj` the monitors are not statically bound to the GUI. The user can change the destination object of each monitor at `Settings -> Set Monitors …` as visualized in figure 21. This opens up the possibility to directly read input from an external file source or write output and error related data to one.

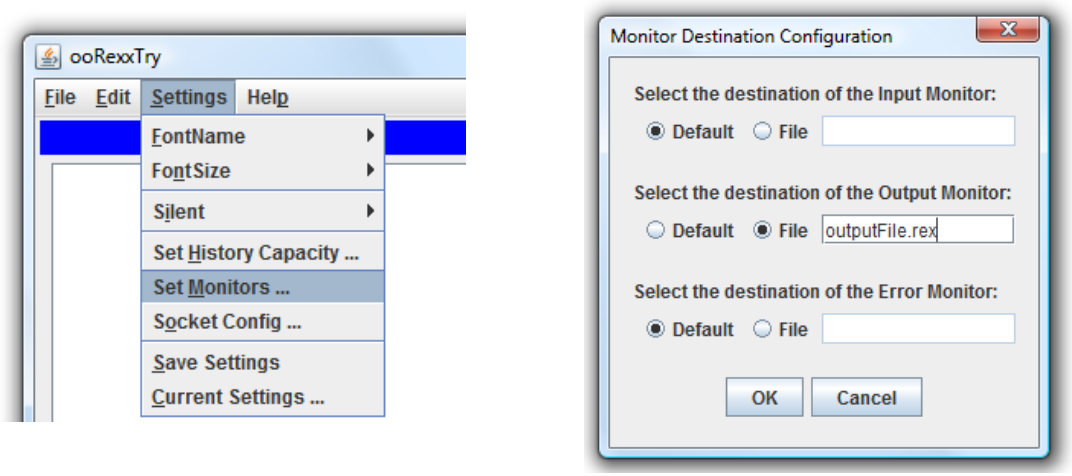

Figure 21: Access to Monitor Configuration and Respective Dialog

Figure 22 shows the `actionPerformed` method of the `DestinationSwitch` class which handles the switching and initiates and prepares new streams.

If the user approves the dialog with the “`OK`” button the program checks every monitor, if the file option was chosen. If so, a new stream is created pointing to the appropriate file. If both the output and the error monitor are about to write to the same file, thus the same stream is utilized.

```
::class DestinationSwitch

/*Whenever settings in the the monitor option dialog gets approved the
method configures the monitors according
  to the choosen configuration.*/
::method actionPerformed
    expose source
    use arg eventObject, slotDir
```

```
    mdialog = slotDir~userData~mdialog

    if eventObject~getSource~equals(slotDir~userData~monitorOkButton)
    then
      do
        incmd = slotDir~userData~ingroup~getSelection~getActionCommand
        if incmd~equals("indefault") then
          do
            .input~destination(.mm.dir~guiInputStream)
            .mm.dir~inputMonitorDestination = 'GUIInputStream'
          end
        else
          do
 -- Create a new stream to use a file as the primary source for input
arguments
             filename = slotDir~userData~infiletf~getText
             .mm.dir~infilestream = .stream~new(filename)
             .input~destination(.mm.dir~infilestream)
             .mm.dir~inputMonitorDestination = filename
          end
      outcmd = slotDir~userData~outgroup~getSelection~getActionCommand
        if outcmd~equals("outdefault") then
          do
            .output~destination(.mm.dir~guiOutputStream)
            .mm.dir~outputMonitorDestination = 'GUIOutputStream'
          end
        else
          do
 -- Create a new stream to write output arguments to a file
           filename = slotDir~userData~outfiletf~getText
           if \filename~equals(slotDir~userData~errfiletf~getText)  -
           then .mm.dir~outfilestream = .stream~new(filename)
            else
              do
              if .mm.dir~errfilestream = -
              .nil then .mm.dir~outfilestream = .stream~new(filename)
                else .mm.dir~outfilestream = .mm.dir~errfilestream
              end
            .output~destination(.mm.dir~outfilestream)
            .mm.dir~outputMonitorDestination = filename
          end
      errcmd = slotDir~userData~errgroup~getSelection~getActionCommand
        if errcmd~equals("errdefault") then
          do
            .error~destination(.mm.dir~guiErrorStream)
            .mm.dir~errorMonitorDestination = 'GUIErrorStream'
          end
        else
          do
             -- Write error data to a file
             filename = slotDir~userData~errfiletf~getText
             if \filename~equals(slotDir~userData~errfiletf~getText) -
             then .mm.dir~errfilestream = .stream~new(filename)
             else
               do
                 if .mm.dir~outfilestream = .nil -
                 then .mm.dir~errfilestream = .stream~new(filename)
                 else .mm.dir~errfilestream = .mm.dir~outfilestream
               end
             .error~destination(.mm.dir~errfilestream)
             .mm.dir~errorMonitorDestination = filename
          end
```

```
        source = "ok"
        mdialog~hide
      end
    else
      do
        source = "cancel"
        mdialog~hide
      end
```

Figure 22: DestinationSwitch Class Sets the Destination for Each Monitor

4.2.5 Socket Connection

`ooRexxTry.rxj` now listens by default on port `8888` for requests from client programs to establish a connection via socket. Using sockets two programs communicate in a two-way scenario.

The computer running the server program is identified by `IP address` or `host name` and `port number`. Input and output operations take place via streams. The server program creates a `ServerSocket` which accepts requests. [Flat10c]

New connections are derived from this `ServerSocket` and run in a separate thread not interrupting the workflow. Figure 23 shows how `ooRexxTry.rxj` manages connections.

```
/*Create a server socket. While flagged for listening, start accepting
client
  connections and manage them in a separate thread.*/
::class CreateServerSocket
::method run
    .mm.dir~srvSock = .bsf~new("java.net.ServerSocket", .mm.dir~port)
    .mm.dir~listening = .true
    do while .mm.dir~listening = .true
        runnableSocket = -
        BsfCreateRexxProxy(.SocketConnection~new, ,.Runnable)
        .Thread~new(runnableSocket)~~bsf.dispatch("start")
    end
    .mm.dir~srvSock~close
```

Figure 23: ServerSocket Creation and Connection Initialization

While flagged for listening the `ServerSocket` accepts a new request and creates a Java `InputStreamReader` from the socket's `InputStream` and a Java `PrintWriter` from the `OutputStream` (cf. Figure 24). The `InputStreamReader` reads bytes from the `InputStream` and decodes them into characters which are subsequently buffered by the `BufferedReader` (`.local.mm.dir~in`). Invoking the `readLine` method on the `BufferedReader` returns a line of characters. The `println` method of the `PrintWriter` (`.local~mm.dir~out`) prints a formatted object representation to the `OutputStream` of the Socket object. Subsequently, the `OutputStream` writes byte data to the client.

```
/*Tap the server socket to listen for a incoming connection*/
::class SocketConnection
::method init
   .local~mm.dir~clientConnected = .false
   .local~mm.dir~socketInputRequired = .false
   self~socket = .mm.dir~srvSock~accept

::method socket attribute

::method run
    if \.mm.dir~listening then return
-- Writer for outband data
    .local~mm.dir~out = -
    .PrintWriter~new(self~socket~getOutputStream, .true)
-- reader for inbound data
    .local~mm.dir~in = .BufferedReader~new(.InputStreamReader~new(self
~socket~getInputStream))
    .mm.dir~out~println("Hello from Server. Type 'Exit' to quit con-
nection.")
    .mm.dir~errorsArea~setText("Client connected.")
    .mm.dir~clientConnected = .true
    do while inputLine \= .nil
-- command that signals session end
       if inputLine~caselessEquals("Exit") then
       do
           .mm.dir~out~println("Exit")
           .mm.dir~errorsArea~setText("Client disconnected.")
           .mm.dir~clientConnected = .false
           leave
       end
       inputLine = .mm.dir~in~readLine
       if .mm.dir~enableInputSocket then
         do
           if .mm.dir~socketInputRequired then
             do
               .mm.dir~inputArea~setText(inputLine)
               .mm.dir~guiInputStream~setControlVar(.true)
             end
           else .mm.dir~codeArea~setText(inputLine)
         end
    end
```

```
-- Close writer and reader and terminate connection
    .mm.dir~out~close
    .mm.dir~in~close
    self~socket~close
```

Figure 24: SocketConnection Class Establishes a Connection to a Client

The remote host and the port a socket is connecting to can be changed dynamically via `Settings -> Socket Config …` as illustrated in figure 25. Such a change only affects subsequently established connections. The lower part of the dialog window is used to determine whether the program should actively use the input and output stream from the socket connection or not.

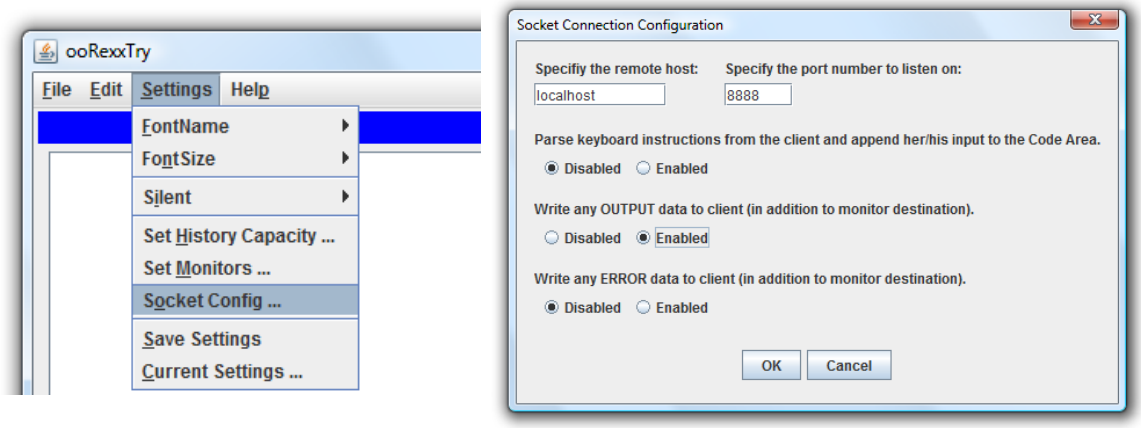

Figure 25: Access to Socket Configuration and Respective Dialog

Provided that the input stream is "`Enabled`" client input is automatically appended to the "Code" area of the GUI. What is more, the client terminal is used as primary source for keyboard input instead of the "`Input`" area.

Figure 26 shows this principle in action. With an "enabled" input stream the "`parse pull`" instruction waits for input from the client and processes it afterwards (cf. Figure 21). See C3 in the appendix for the source code of the input related client script.

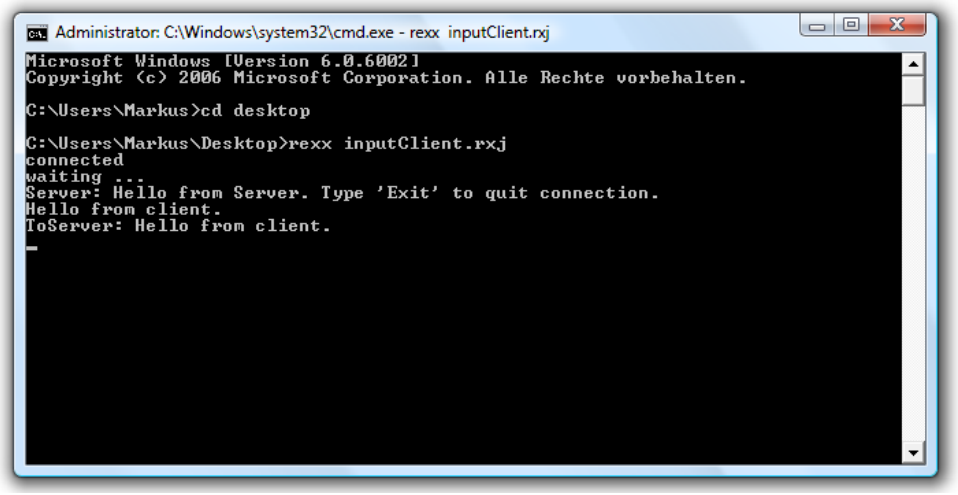

Figure 26: Connected Client Sending Message to Server

As visualized in Figure 27, the client input is processed as expression of the `say` instruction and the result is displayed in the "`Output/Says`" area.

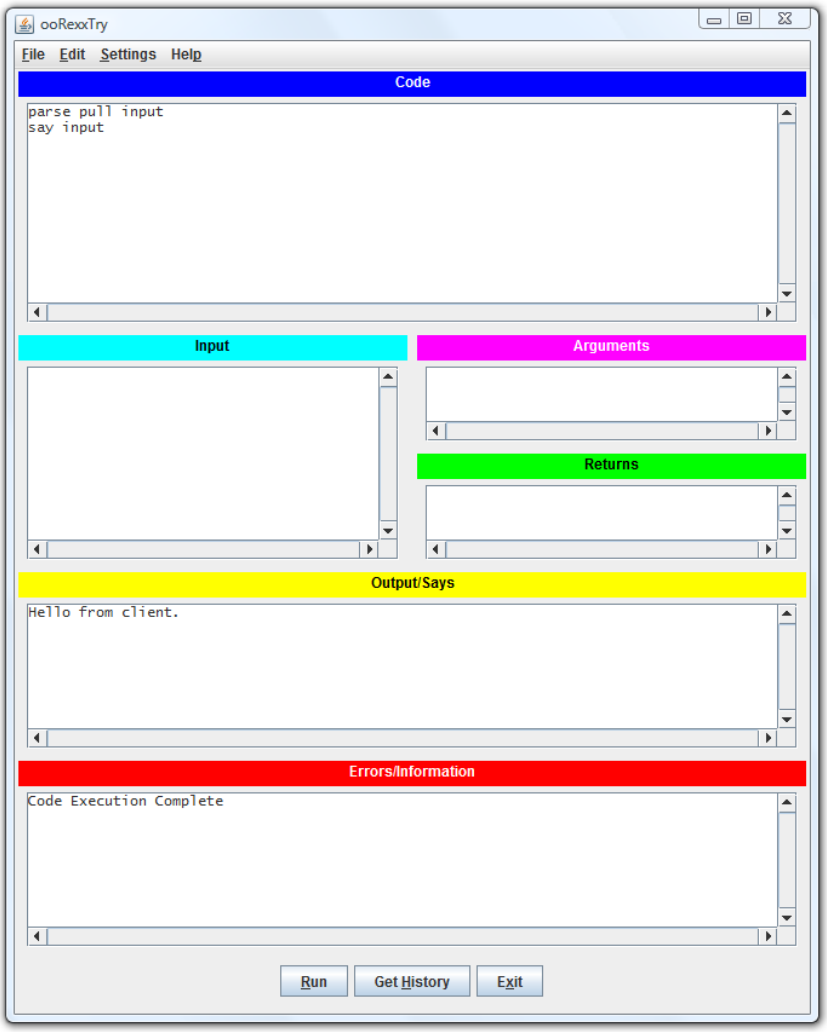

Figure 27: GUI Showing Processed Client Input

The two remaining options at the "`Socket Config`" dialog enable or disable the output or the error stream respectively to simultaneously write to the areas of the GUI and to the socket client. Figures 28 and 29 demonstrate a trace process while both the output and the error stream option is set to "`Enabled`".

Output and error data is marked with `[.OUTPUT]` and `[.ERROR]` and is given to the `PrintWriter` object which sends the messages to the client. See C4 in the appendix for the source code of the output related client script.

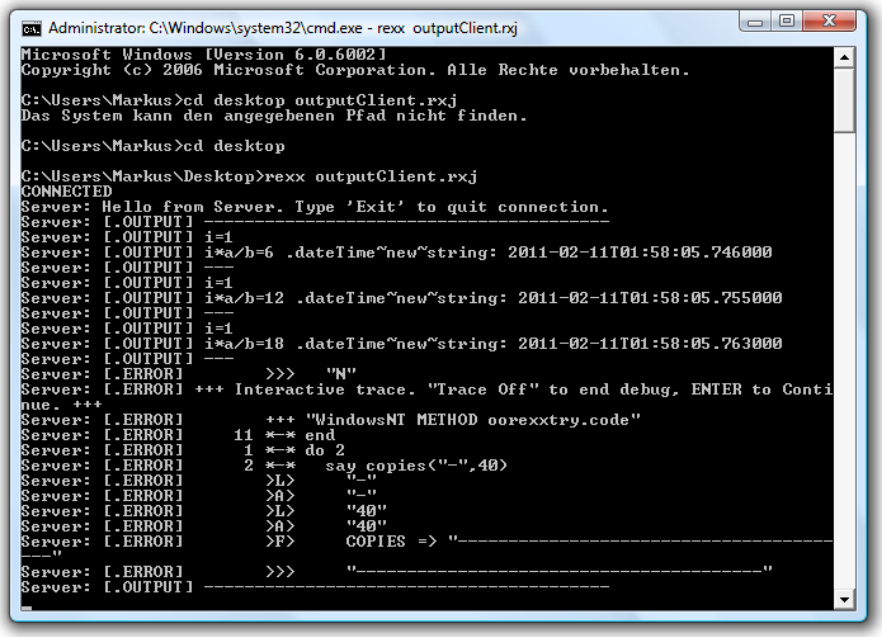

Figure 28: Connected Client Receiving Output and Error Related Data

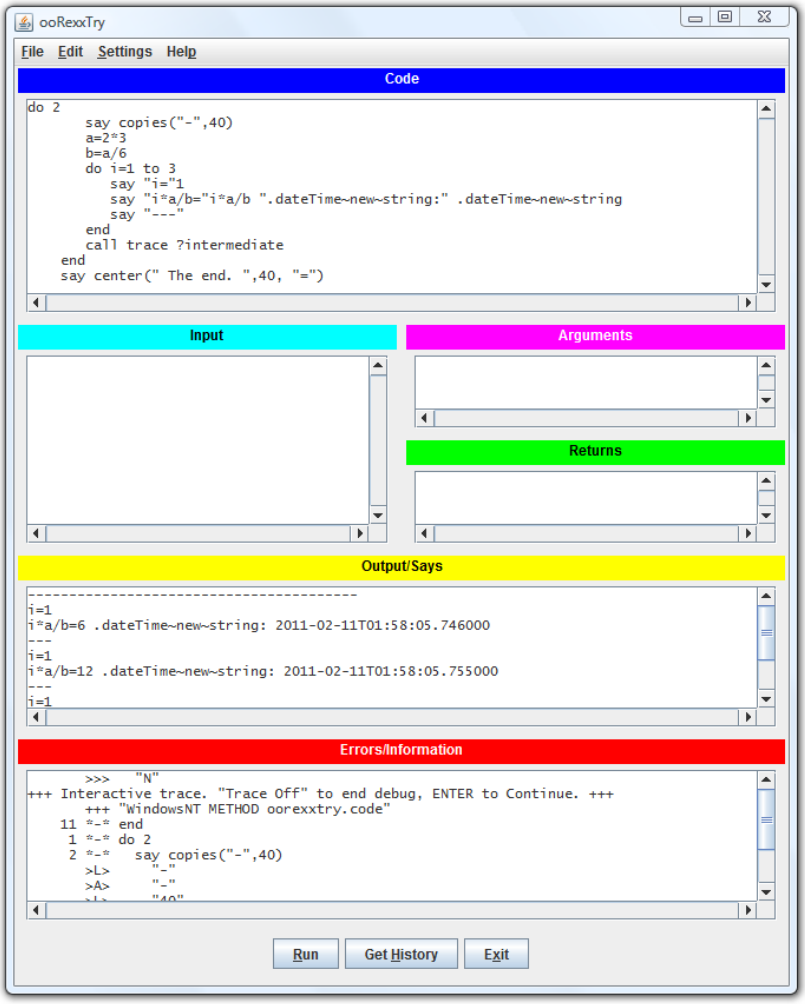

Figure 29: GUI Showing Tracing in Process

If the connection is being cut or the client or the server wishes to terminate it (by sending “`exit`”) the input and output stream and subsequently the socket connection itself get closed (cf. figure 24).

4.2.6 Java Class Specialization in ooRexx

As mentioned initially `ooRexxTry.rxj` also leverages the functionality to specialize Java classes in ooRexx using something called `ProxyClass`[5]. As referred to, `RexxProxy` objects allow implementing Java interfaces and abstract classes in ooRexx. Invoking a method on an implemented interface results in Java sending a message to the appropriate `RexxProxy` object. Now with a `ProxyClass` the same process flow can be realized but the `RexxProxy` object is also able to subsequently forward the original method call to the addressed Java class after some processing.

`ooRexxTry.rxj` uses this feature to implement the AWT Dimension class as a `ProxyClass` along with its `setSize` method. In the context of the input and output areas on the GUI, `Dimension` objects are used to determine the minimum and preferred sizes of the components relative to the main frame. Thus ensure smooth scaling.

The `setSize` method of a `Dimension` object would normally take the width and height of the component. In order to demonstrate the functionality the section is about, the signature of the initial `setSize` method call is modified to encapsulate the width and height setting and delegate it to the `RexxProxy`. As a result, "`setSize`" only takes two simple values standing for the size ratio between the GUI's main frame and an inner component. Further instructions are appended by the `RexxProxy` which also carries out the final method call to the addressed Java (super) class.

[5] cf. Public Class BSF http://wi.wu-wien.ac.at:8002/rgf/rexx/bsf4oorexx/current/additionalResources/refcardBSF4ooRexx.pdf

As demonstrated in figure 30 the first step is to create a `ProxyClass` for a Java class:

proxyClassObj = bsf.createProxyClass(javaClz, newClzName|.nil, javaMeth[, javaMeth2, …])

The next step is to create a `RexxProxy` which receives calls to the determined method(s). This `RexxProxy` object is also passed as first argument when instantiating the `ProxyClass` object. Arguments a specific Java constructor may requires are passed subsequently:

javaObj = proxyClassObj~new(rexxProxyObj[, construcArg1, …])

Calling an implemented method ultimately leads to the invocation of the respective method of the `RexxProxy` object. Calls to Java methods not implemented in this way are dispatched as usual.

```
/*Rexx proxy class for Java class specialization in ooRexx;
      first argument: Java super class
      second argument: optional new class name, .nil for default
      third argument: list of methods to specialize*/
    dimpc = bsf.createProxyClass("java.awt.Dimension", -
            .nil, "setSize")
-- Proxy object, receives method call
    dimrp = BsfCreateRexxProxy(.Dimensioning~new)

    /*Create an instance of the proxy class by supplying the proxy
      object as first argument followed by
      all other necessary arguments the Java constructor would take.*/
    dim = dimpc~new(dimrp)~~setSize(1, 35)
    argspanel~~setPreferredSize(dim)~~setMinimumSize(dim)
    codepanel~~setPreferredSize(dim)~~setMinimumSize(dim)
    dim2 = dimpc~new(dimrp)~~setSize(1, 12)
    argsScrollPane~~setPreferredSize(dim2)~~setMinimumSize(dim2)
```

Figure 30: Implement and Instantiate Java Class and Call Modified Method

The `Dimensioning` class (`RexxProxy`) in figure 31 implements the method "`setSize`" from `java.awt.Dimension`. Note that methods implemented in this way get a directory object attached as the last argument when invoked. This directory a.k.a. `slotDir` holds the entry `javaObject` representing the Java class object.

The following command calls the appropriate method of the Java super class:

slotDir~javaObject~methName_forwardToSuper([arg1, arg2, …])

In other words, the RexxProxy object dispatches the method call of a Java class after some crucial processing in ooRexx.

```
/* Rexx proxy used to specialize the Java class java.awt.Dimension in
ooRexx.*/
::class Dimensioning
::method setSize
-- slotDir: directory, which is attached to every call of
-- a Java method implemented in ooRexx.
-- Holds information about the method and the java object itself.
-- javaObject: messages addressed to the Java super class are sent
-- to this object.
    use arg wdivider, hdivider, slotDir

    width = .mm.dir~frame~getWidth
    height = .mm.dir~frame~getHeight

/*Invoke the method in the Java super class.*/
slot-
Dir~javaObject~setSize_forwardToSuper((width/wdivider)~format(,0), -
(height/hdivider)~format(,0))
```

Figure 31: Dimensioning Class Serving as Proxy Class

5 Round-up

ooRexxTry is a convenient GUI-based tool for becoming familiar with ooRexx and for quick code testing. Just enter your Rexx code in the "Code" area of the graphical user interface and promptly "run" feedback.

Now with the support from Java widget libraries, which substitute the Windows-genuine library packages, ooRexxTry runs on every system with a Java Run-time Environment installed independently of the underlying operating system. A great variety of classes from the java.awt, javax.swing, java.lang and java.io packages was leveraged to meet the requirements of the aforementioned use cases.

The GUI of the Java-based ooRexxTry.rxj features a new "Input" area and uses custom streams for the .input, .output and .error monitor instead of the standard streams. As a result, the user is able to provide all input directly at the GUI and receives all output there. So the command prompt becomes obsolete and does not have to be observed.

ooRexxTry.rxj also enhances the basic functionality with a code execution history. Previously run code is stored along with a time stamp in a history which is displayed at a respective dialog. The user always has an overview of her/his recent code executions and can select an item to retrieve from the history.

Now that ooRexxTry.rxj listens for socket connections a client may connect via socket to remotely provide input to the program and to get output and error data from it.

Last but not least, ooRexxTry.rxj offers the possibility to dynamically switch the destination objects (streams) the monitors of the program read from and write to. Hence, the user can select a file source for primary input or output instead of the assigned areas of the GUI.

6 References

[DRef10] Ashley D. W., Flatscher R. G. et al.: Windows OODialog Reference, Version 4.1.0 Edition, December 2010

[ERef10] Ashley D. W., Flatscher R. G. et al.: Windows Extensions Reference, Version 4.1.0 Edition, December 2010

[FeigSA] Feigenbaum B.: SWT, Swing or AWT: Which is right for you?

http://www.ibm.com/developerworks/grid/library/os-swingswt/

Requested in December 2010

[Flat01] Flatscher R. G.: Java Bean Scripting with Rexx

http://wi.wu.ac.at/rgf/rexx/orx12/JavaBeanScriptingWithRexx_orx12.pdf

[Flat04] Flatscher R. G.: Camouflaging Java as Object Rexx

http://wi.wu-wien.ac.at/rgf/rexx/orx15/2004_orx15_bsf-orx-layer.pdf

[Flat10b] Flatscher R. G.: Creating Portable GUIs for ooRexx Using BSF4ooRexx

http://wi.wu-wien.ac.at:8002/rgf/wu/lehre/autojava/material/foils/AutoJava-BSF4ooRexx-02-GUI.pdf

[Flat10c] Flatscher R. G.: Creating Portable Socket- and SSL-Applications in ooRexx Using BSF4ooRexx

http://wi.wu-wien.ac.at:8002/rgf/wu/lehre/autojava/material/foils/AutoJava-BSF4ooRexx-03-Sockets.pdf

[FlatIntro] Flatscher R. G.: An Introduction to Procedural and Object-oriented Programming (ooRexx)

http://wi.wu-wien.ac.at:8002/rgf/wu/lehre/autowin/material/foils/ooRexx_1.pdf

[JSEAPI] Java™ 2 Platform, Standard Edition, v 1.4.2 API Specification

http://download.oracle.com/javase/1.4.2/docs/api/

[ooRexx] http://www.oorexx.org/about.html

Requested in December 2010

[Peed07] Peedin L.: ooRexxTry Reference, November 2007

[RRef10] Ashley D. W., Flatscher R. G. et al.: Open Object Rexx Reference, Version 4.1.0 Edition, December 2010

7 Appendix

C1 RunIt Class

Parses and formats arguments and code, traps argument errors and instantiates the code execution thread.

```
-- Initialize code execution
::class RunIt
::method actionPerformed
    expose slotDir
    use arg , slotDir
    self~runCode

::method runCode
    expose slotDir

    frame = .mm.dir~frame
    codeArea = .mm.dir~codeArea
    returnsArea = .mm.dir~returnsArea
    errorsArea = .mm.dir~errorsArea
    saysArea = .mm.dir~saysArea
    cursorpos = codeArea~getCaretPosition
    frame~setTitle(.mm.dir~title)
    frame~setCursor(.Cursor~getPredefinedCursor(.Cursor~wait_cursor))
    argsArea = .mm.dir~argsArea
    if .mm.dir~hdialog \= .nil then -
    if .mm.dir~hdialog~isShowing then .mm.dir~hdialog~hide

    arg_array = .array~new
    arg_string = argsArea~getText
    arg_array = arg_string~makeArray('0a'x)

    .local~mm.dir~code_string = codeArea~getSelectedText

    -- Clear any previous say data
    saysArea~setText('')

    -- Clear any previous returns data
    returnsArea~setText('')

    .local~mm.dir~emsg = ''
    .local~mm.dir~imsg = ''

    errorsArea~setText('Code Is Executing')

    .local~mm.dir~Error? = .false
    .local~mm.dir~badarg = ''
    .local~mm.dir~say_stg = ''
    .local~mm.dir~error_stg = ''
    -- Interpret each argument so that expressions can be used
    signal on syntax name ArgSyntax
    do i = 1 to arg_array~items
        .local~mm.dir~badarg = i arg_array[i]
        interpret 'arg_array['i'] =' arg_array[i]
    end
    signal off syntax
```

```
    -- Run the code in a dynamically created method

    found_cc = .false
    if .mm.dir~code_string \= .nil then
        do
            code_array = .array~new
            code_array = .mm.dir~code_string~makeArray('0a'x)
        end
    else
        do
            .mm.dir~code_string = codeArea~getText
            code_array = .mm.dir~code_string~makeArray('0a'x)
        end
    do ca = 1 to code_array~items
        a_ca = code_array[ca]~strip()
        if a_ca~pos('::') = 1 then
            do
                found_cc = .true
                leave ca
            end
    end

    if \found_cc then
        do
  runnable = BsfCreateRexxProxy(.ExecuteThread~new('oorexxtry.code', -
  code_array, arg_array, slotDir~userData), ,.Runnable)
            .Thread~new(runnable)~~bsf.dispatch("start")
  -- Run code in separate thread
        end
    else
        do
            /*Temporary code storage*/
            .local~mm.dir~tempFile = 'ooRexxTry_test9999.rex'
            c_stream = .stream~new(.mm.dir~tempFile)
            c_stream~open('Write Replace')
            do ca = 1 to code_array~items
                c_stream~lineout(code_array[ca])
            end
            c_stream~close
            arg_string = ''
            do ca = 1 to arg_array~items
                arg_ca = '"'arg_array[ca]'"'
                arg_string = arg_string','arg_ca
            end
            arg_string = arg_string~strip('b',',')
   runnable = BsfCreateRexxProxy(.ExecuteThread~new('oorexxtry.code',-
   'call ooRexxTry_test9999.rex' arg_string, arg_array, -
   slotDir~userData), ,.Runnable)
            .Thread~new(runnable)~~bsf.dispatch("start")
  -- Run code in separate thread
        end

return

ArgSyntax:
    errorarg = .mm.dir~badarg~subword(2)

    call ppCondition condition("o")

    if \.mm.dir~silent then
        call beep 600,100
    returnsArea~setText('')
```

```
    argsArea~requestFocus
    argsArea~select(arg_string~wordPos(errorarg), -
    arg_string~wordPos(errorarg)+errorarg~length)
    frame~setCursor(.Cursor~getDefaultCursor)

    -- Close the file streams after the run or when an error occurred.
Allow to immediate (re)view file code or alternate it.
    if .mm.dir~infilestream \= .nil then .mm.dir~infilestream~close
    if .mm.dir~outfilestream \= .nil then .mm.dir~outfilestream~close
    if \.mm.dir~errorMonitorDestination~equals('GUIErrorStream') then
      do
        .mm.dir~errorsArea~setText("An Arguments error occured. " -
        "Please see "||slotDir~userData~errfiletf~getText|| -
        " for further details.")
        .mm.dir~errfilestream~close
      end
return
```

C2 Executor Class

Creates an executable representation of the code, runs the code, traps errors raised during execution, stores successfully run code in the history and appends returned values on the GUI.

```
-- Class that dynamically creates a method to take the arguments and
execute the code.
::class executor public
::method init
    expose rt_method userData
    use arg method_name,code, userData
    .local~mm.dir~Error? = .false
    .local~mm.dir~error_stg = ''
    signal on syntax name ExecSyntax
    rt_method = .method~new(method_name, code)
return

-- Syntax trap similiar to rexxc.exe
ExecSynTax:
    call ppCondition condition("o")

    .local~mm.dir~Error? = .true

    if .mm.dir~infilestream \= .nil then .mm.dir~infilestream~close
    if .mm.dir~outfilestream \= .nil then .mm.dir~outfilestream~close
    if \.mm.dir~errorMonitorDestination~equals('GUIErrorStream') then
      do
        .mm.dir~errorsArea~setText("An Execution error occurred. " -
        "Please see "||userData~errfiletf~getText|| -
        " for further details.")
        .mm.dir~errfilestream~close
     end

return

-- Method that actually runs our code
```

```
::method run
    expose rt_method say_string userData
    signal on syntax name RunSyntax
    .local~mm.dir~run_results = .directory~new
    .local~mm.dir~say_stg = ''

    my_result = ''
    if \.mm.dir~Error? then
        do
            -- Run the Code
            args = arg(1)
            self~run:super(rt_method, 'a', args)

            -- Test if there was anything returned by the code
            if symbol('result') = 'VAR' then my_result = result
            -- Load the says and returns into environment variables
            -- for updating the dialog areas
            .local~mm.dir~run_results['returns'] = my_result
            if .mm.dir~error_stg == '' then -
            .mm.dir~errorsArea~setText("Code Execution Complete")
        end

    .mm.dir~storage~storeData(.mm.dir~code_string)

    if .nil \= .mm.dir~run_results['returns'] then
        .mm.dir~returnsArea~setText(.mm.dir~run_results['returns'])
-- Let the user know when code execution is complete
    if \.mm.dir~silent then
        call beep 150,150
    .mm.dir~frame~setCursor(.Cursor~getDefaultCursor)
-- Return the focus to the code input area
    .mm.dir~codeArea~requestFocus

    -- Close the file streams after the run or when an error occurred.
    -- Allow to immediate (re)view file code or alternate it.
    if .mm.dir~infilestream \= .nil then .mm.dir~infilestream~close
    if .mm.dir~outfilestream \= .nil then .mm.dir~outfilestream~close
    if .mm.dir~errfilestream \= .nil then .mm.dir~errfilestream~close

return

-- Syntax trap for errors in the code
RunSyntax:
    call ppCondition condition("o")

    if \.mm.dir~silent then call beep 600,100
-- Let the user know when code execution is complete
     if \.mm.dir~silent then
        call beep 150,150
    .mm.dir~frame~setCursor(.Cursor~getDefaultCursor)

    -- Close the file streams after the run or when an error occurred.
    -- Allow to immediate (re)view file code or alternate it.
    if .mm.dir~infilestream \= .nil then .mm.dir~infilestream~close
    if .mm.dir~outfilestream \= .nil then .mm.dir~outfilestream~close
    if \.mm.dir~errorMonitorDestination~equals('GUIErrorStream') then
      do
      .mm.dir~errorsArea~setText("A Run error occurred. Please see " -
       ||userData~errfiletf~getText||" for further details.")
        .mm.dir~errfilestream~close
      end
    return
```

C3 Socket Client: Input

Simple script that establishes a connection to the server application via socket. Parses terminal input and sends messages to the server application.

```
host = 'localhost'      -- server to connect to
port = 8888             -- port the application listens on
-- create socket and connect to server
socket2server=.bsf~new('java.net.Socket', host, port)
say connected
say "waiting ..."

-- output stream for messages destined for the server
out = .bsf~new("java.io.PrintWriter", -
      socket2server~getOutputStream, .true)
-- input stream for messages from the server
in = .bsf~new("java.io.BufferedReader", -
  .bsf~new("java.io.InputStreamReader", socket2server~getInputStream))

-- stream for command line parsing
stdIn = .bsf~new("java.io.BufferedReader", -
        .bsf~new("java.io.InputStreamReader", -
.       bsf~bsf.import('java.lang.System')~in))

say ("Server: "||in~readLine)   -- initial message from the server
toServer = stdIn~readLine       -- parse client input

do while toServer \= .nil
  say ("ToServer: "||toServer)
-- stop listening for client input when he/she wishes to
-- terminate connection by typing "exit"
  if toServer~caselessEquals("Exit") then leave
  out~println(toServer)
  toServer = stdIn~readLine
end
out~println("exit")
say "Disconnected from Server."
-- close streams and socket and terminate connection
out~close
in~close
stdIn~close
socket2server~close

::requires bsf.cls
```

C4 Socket Client: Output

Simple script that establishes a connection to the server application via socket. Receives error and output related data from the server and displays it to the user.

```
host = 'localhost'       -- server to connect to
port = 8888              -- port the application listens on
-- create socket and connect to server
socket2server=.bsf~new('java.net.Socket', host, port)
say connected
say "waiting ..."

-- output stream for messages destined for the server
out = .bsf~new("java.io.PrintWriter", -
      socket2server~getOutputStream, .true)

-- input stream for messages from the server
in = .bsf~new("java.io.BufferedReader", -
  .bsf~new("java.io.InputStreamReader", socket2server~getInputStream))

fromServer = in~readLine         -- initial message from the server

do while fromServer \= .nil
  serverMsg = fromServer~subStr(11)
-- listen for messages from the server until "exit" is sent
  if serverMsg~caselessEquals("Exit") then leave
  say ("Server: "||fromServer)
  fromServer = in~readLine
end
out~println("Exit")
say "Disconnected from Server."
-- close streams and socket and terminate connection
out~close
in~close
socket2server~close

::requires bsf.cls -- get Java support
```

Eigenständigkeitserklärung

Lehrveranstaltungsnummer: 0502 Semester: WS 2010/11

Lehrveranstaltung: Projektseminar aus Wirtschaftsinformatik

Lehrveranstaltungsleiter: ao.Univ.Prof. Dr. Rony G. Flatscher

Verfasser: Markus Moldaschl

Matrikelnummer: 0751916

Ich versichere / stimmte zu:

1. dass ich die Arbeit selbstständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfe bedient habe.

2. dass ich dieses Thema bisher weder im In- noch im Ausland (einer Beurteilerin/einem Beurteiler zur Begutachtung) in irgendeiner Form als Prüfungsarbeit vorgelegt habe.

3. dass ich damit einverstanden bin, dass die vorliegende Arbeit oder Auszüge dieser Arbeit im Rahmen von Forschungsprojekten und für Publikationen weiterverwendet werden können.

Datum Unterschrift