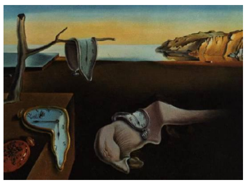


Journal of Statistical Software
June 2012, Volume 49, Issue 9. http://www.jstatsoft.org/

Supplement to: RKWard – A Comprehensive Graphical User Interface and Integrated Development Environment for Statistical Analysis with R

Stefan Rödiger
Charité-Universitätsmedizin Berlin

Thomas Friedrichsmeier
Ruhr-University Bochum

Prasenjit Kapat
The Ohio State University

Meik Michalke
Heinrich Heine University Düsseldorf

Abstract

RKWard is a GUI to R with the objective to provide a portable and extensible R interface for both basic and advanced statistical and graphical analysis. This supplement discusses in detail technical aspects of **RKWard** including the usage of the **KDE**/**Qt** software libraries which are the base of **RKWard**. Statistical procedures and plots are implemented as an extendable plugin architecture in **RKWard**. This plugin architechture is based on ECMAScript (JavaScript), R, and XML. The general design is described and its application is exemplified on a Student's t test.

Keywords: cross-platform, GUI, integrated development environment, plugin, R.

1. Introduction

In this supplement we will give an overview of some key aspects of **RKWard**'s technical design and development process, comparing them briefly to competing GUI solutions, where appropriate. We will give slightly more attention to the details of the plugin framework (Section 6) used in **RKWard**, since this is central to the extensibility of **RKWard**, and we will conclude with an example for extending **RKWard** by a plugin (Section 7).

Note that this document refers to **RKWard** version 0.5.6. Several technical details, described here, have changed in **RKWard** version 0.5.7 and the current development version.

2. Asynchronous command execution

One central design decision in the implementation of **RKWard** is that the interface to the R engine operates asynchronously. The intention is to keep the application usable to a high degree, even during the computation of time-consuming analysis. For instance, while waiting for the estimation of a complex model to complete, the user should be able to continue to use the GUI to prepare the next analysis. Asynchronous command execution is also a prerequisite for an implementation of the plot-preview feature (see the section no graphics windows in the main article). Internally, the GUI frontend and the R engine run in two separate processes[1]. Commands generated from plugins or user actions are placed in queue in the frontend and are evaluated in the backend process in the order they were submitted[2].

The asynchronous design implies that **RKWard** avoids relying on the R engine during interactive use. This is one of several reasons for the use of ECMAScript in plugins, instead of scripting using R itself (see Sections 4 and 6). A further implication is that **RKWard** avoids querying information about the existence and properties of objects in R interactively. Rather, **RKWard** keeps a representation of R objects and their basic properties (e. g., class and dimensions), which is used for the workspace browser, object name completion, function argument hinting, and other places. The object representation includes objects in all environments in the search path, and any objects contained within these environments in a hierarchical tree[3]. The representation of R objects is gathered pro-actively[4]. This has a notable impact on performance when loading packages. Specifically, objects which would usually be "lazy loaded" only when needed (see Ripley 2004) are accessed in order to fetch information on their properties. This means the data has to be loaded from disk; however, the memory is freed immediately after fetching information on the object. Additionally, for packages with extremely large number of objects, **RKWard** provides an option to exclude specific packages from scanning the object structures.

A further side-effect of the asynchronous threaded design is that there is inherently a rather clear separation between the GUI code and the code making direct use of the R application programming interface (API) (see also Figure 1). In future releases it could be made possible to run GUI and R engine on different computers.

[1] Up to **RKWard** version 0.5.4, two separate threads inside a single process were used. This alternate design is still available as a compile time option.

[2] It is possible, and in some cases necessary, to enforce a different order of command execution in internal code. For instance, **RKWard** makes sure that no user command can potentially interfere while **RKWard** is loading the data of a `data.frame` for editing.

[3] Currently, environments of functions or formulas are not taken into account, but slots of S4 objects, and package namespace environments are represented in the object tree.

[4] To limit the amount of processing, and to avoid recursion, **RKWard** currently stops gathering object information at a depth of three levels. Information on deeper levels is gathered on an as-needed basis, when the user accesses information on the respective parent objects.

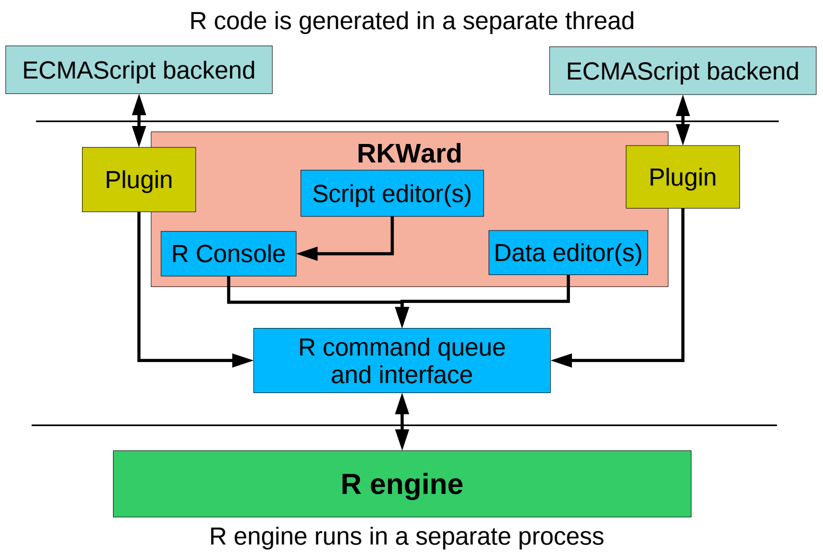

R code is generated in a separate thread
ECMAScript backend
ECMAScript backend
Plugin
RKWard
Script editor(s)
Plugin
R Console
Data editor(s)
R command queue and interface
R engine
R engine runs in a separate process

Figure 1: Technical design of **RKWard**. Only a few central components are visualized. All communication with the R engine is passed through a single interface living in the frontend process. The R engine itself runs in a separate process. Separate threads within the frontend process are used to generate R code from plugins.

3. Object modification detection

RKWard allows the user to run arbitrary commands in R at any time, even while editing a `data.frame` or while selecting objects for analysis in a GUI dialog. Any user command can potentially add, modify, or remove objects in R. **RKWard** tries to detect such changes in order to always display accurate information in the workspace browser, object selection lists, and object views. Beyond that, detecting any changes is particularly important with respect to objects which are currently being edited in the data editor (which provides an illusion of in-place editing, see the section on the spreadsheet-like data editor in the main article). Here, it is necessary to synchronize the data between R and the GUI in both directions.

For simplicity and performance, object modification detection is only implemented for objects inside the "global environment" (including environments inside the global environment), since this is where changes are typically done. Currently, object modification detection is based on active bindings. Essentially, any object which is created in the global environment is first moved to a hidden storage environment, and then replaced with an active binding. The active binding acts as a transparent proxy to the object in the storage environment, which registers any write-access to the object[5].

The use of active bindings has significant performance implications when objects are accessed very frequently. This is particularly notable where an object inside the global environment is used as the index variable in a loop, as illustrated by the following example. When control returns to the top level prompt, after the first assignment, `i` will become subject to object modification detection (i. e., it will be wrapped into an active binding). The subsequent `for`

[5] This is similar to the approach taken in the **trackObjs** package (Plate 2009).

loop will then run slow.

```
R> i <- 1
R> for (i in 1:100000) i + i
```

In contrast, in the following example, `i` is a local object, and will not be replaced by an active binding. Therefore the loop will run approximately as fast as in a plain R session:

```
R> f <- function () {
R+    i <- 1
R+    for (i in 1:100000) i + i
R+ }
R> f ()
```

Future versions of **RKWard** will try to avoid this performance problem. One approach that is currently under consideration is to simply perform a pointer comparison of the `SEXP` records of objects in global environment with their copies in a hidden storage environment. Due to the implicit sharing of `SEXP` records (R Development Core Team 2012b,a), this should provide for a reliable way to detect changes for most types of R objects, with comparatively low memory and performance overhead. Special handling will be needed for environments and active bindings.

4. Choice of toolkit and implementation languages

In addition to R, **RKWard** is based on the **KDE** libraries (**KDE** e.V. 2012), which are in turn based on **Qt** (Nokia Corporation 2012), and implemented mostly in C++. Compared to many competing libraries, this constitutes a rather heavy dependency. Moreover, the **KDE** libraries are still known to have portability issues especially on Mac OS X, and to some degree also on the Microsoft Windows platform (Jarvis 2010).

The major reason for choosing the **KDE** and **Qt** libraries has been the many high level features, they provide. This has allowed **RKWard** development to make quick progress despite limited resources. Most importantly, the **KDE** libraries provide a full featured text editor (Cullmann n.d.) as a component which can be seamlessly integrated into a host application using the KParts technology (Faure 2000). Additionally, another KPart provides HTML browsing capabilities in a similarly integrated way. The availability of **KWord** (KOffice.Org 2010) as an embeddable KPart might prove useful in future versions of **RKWard**, when better integration with office suites will be sought. Additionally **Qt** libraries offer the encapsulation of the look-and-feel on specific platforms for a high degree of interoperability and a wide selection of powerful widgets (Raaphorst 2003).

Another technology from the **KDE** libraries that is important to the development of **RKWard** is the "XMLGUI" technology (Faure 2000). This is especially helpful in providing an integrated GUI across the many different kinds of document windows and tool views supported in **RKWard**.

Plugins in **RKWard** rely on XML (`http://www.w3.org/XML/`) and ECMAScript (`http://www.ecmascript.org/`; see Section 6). XML is not only well suited to describe the layout of the GUI of plugins, but simple functional logic can also be represented (see also Visne *et al.* 2009). ECMAScript was chosen for the generation of R commands within plugins, in particular due to its availability as an embedded scripting engine inside the **Qt** libraries. While at first glance R itself would appear as a natural choice of scripting language as well, this would make it impossible to use plugins in an asynchronous way. Further, the main functional requirement in this place is the manipulation and concatenation of text strings. While R provides support for this, concatenating strings with the +-operator, as available in ECMAScript, allows for a very readable way to perform such basic text manipulation.

5. On-screen graphics windows

Contrary to the approach used in **JGR** (Helbig *et al.* 2011), **RKWard** does not technically provide a custom on-screen graphics device. **RKWard** detects when new graphics windows are created via calls to `X11()` or `windows()`. These windows are then "captured" in a platform dependent way (based on the XEmbed (Ettrich and Taylor 2002) protocol for **X11**, or on reparenting for the Microsoft Windows platform). An **RKWard** menu bar and a toolbar is then added to these windows to provide added functionality. While this approach requires some platform dependent code, any corrections or improvements made to the underlying R native devices will automatically be available in **RKWard**.

A recent addition to the on-screen device is the "plot history" feature which adds a browsable list of plots to the device window. Since **RKWard** does not use a custom on-screen graphics device, this feature is implemented in a package dependent way. For example, as of this writing, plotting calls that use either the "standard graphics system" or the "**lattice** system" can be added to the plot history; other plots are drawn but not added. The basic procedure is to identify changes to the on-screen canvas and record the existing plot before a new plot wipes it out. A single global history for the recorded plots is maintained which is used by all the on-screen device windows. This is similar to the implementation in Rgui.exe (Microsoft Windows), but unlike the one in Rgui.app (Mac OS X). Each such device window points to a position in the history and behaves independently when recording a new plot or deleting an existing one.

Plot history support for the **lattice** system (Sarkar 2008) is implemented by inserting a hook in the `print.lattice()` function. This hook retrieves and stores the `lattice.status` object from the `lattice:::.LatticeEnv` environment, thereby making `update()` calls on trellis objects transparent to the user. Any recorded trellis object is then replayed using `plot.lattice()`, bypassing the recording mechanism. The standard graphics system, on the other hand, is implemented differently because the hook in `plot.new()` is ineffective for this purpose. A customized function is overloaded on `plot.new()` which stores and retrieves the existing plot, essentially, using `recordPlot()` and replays them using `replayPlot()`.

The actual plotting calls are tracked using appropriate `sys.call()` commands in the hooks. These call strings are displayed as a drop-down menu on the toolbar for non-sequential browsing (see the section on graphics windows in the main article) providing a very intuitive browsing interface unlike the native implementations in `windows()` and `quartz()` devices.

6. Plugin infrastructure

One of the earliest features of **RKWard** was the extensibility by plugins. Basically, plugins in **RKWard** provide complete GUI dialogs, or re-usable GUI components, which accept user settings and translate those user settings into R code[6]. Thus, the plugin framework is basically a tool set used to define GUIs for the automatic generation of R code.

Much of the functionality in **RKWard** is currently implemented as plugins. For example, importing different file formats relying on the **foreign** package is achieved by this approach. Similarly, **RKWard** provides a modest GUI driven tool set for statistical analysis, especially for item response theory, distributions, and descriptive statistical analysis.

6.1. Defining a plugin

Plugins consist of four parts as visualized in Figure 2 (see Section 7 for an example; for a complete manual, see Friedrichsmeier and Michalke 2011):

- An XML file (Section 7.1), called a "plugin map", is used to declare one or more plugins, each with a unique identifier. For most plugins, the plugin map also defines the placement in the menu hierarchy. Plugin maps are meant to represent groups of plugins. Users can disable/enable such groups of plugins in order to reduce the complexity of the menu hierarchy.

- A second XML file describes the plugin GUI layout itself (Section 7.2). Most importantly this includes the definition of the GUI layout and GUI behavior. High level GUI elements can be defined with simple XML-tags. Layout is based on "rows" and "columns", instead of pixel counts. In most cases this allows for a very sensible resizing behavior. **RKWard** supports single-page dialogs and multi-page wizards, however, most plugins define only a single-page GUI. GUI behavior can be programmed by connecting "properties" of the GUI elements to each other. For example, the state of a checkbox could be connected to the "enabled" property of a dependent control. More complex logic is also supported, as is procedural scripting of GUI behavior using ECMAScript.

- A separate ECMAScript file (Section 7.3) is used to translate GUI settings into R code[7]. This ECMAScript file is evaluated asynchronously in a separate thread. **RKWard** currently enforces structuring the code into three separate sections for preprocessing, calculating, and printing results. The generated code is always run in a local environment, in order to allow the use of temporary variables without the danger of overwriting user data.

- A third XML file defines a help page. This help page usually links to the R help pages of the main functions/concepts used by the plugin, as well as to other related **RKWard** help pages. Compared to R help pages, the plugin help pages try to give more hands-on

[6] Plugins are also used in some other contexts within **RKWard**, for instance, the integrated text editor (**kate** part) supports extensions via plugins and user scripts. At this point we will focus only on plugins generating R code.

[7] In earlier versions of **RKWard**, PHP was used as a scripting engine, and PHP interpreters were run as separate processes. Usage of PHP was abandoned in **RKWard** version 0.5.3 for reasons of performance and simplicity.

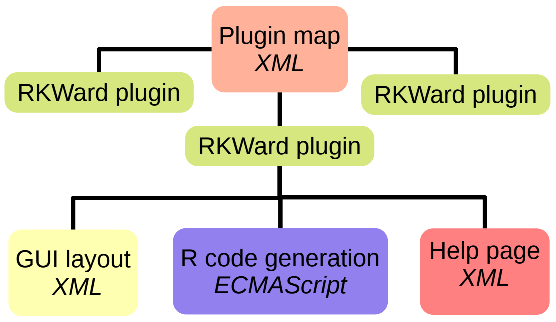


Figure 2: Plugin structure of **RKWard**. One or more plugins are declared in a "plugin map". Each plugin is defined by two XML files, and one ECMAScript file.

advice on using the plugin. Plugins can be invoked from their help page by clicking on a link near the top, which can be useful after following a link from a related help page.

Changes to the source code of these elements take effect without the requirement to recompile **RKWard**.

6.2. Embedding and reuse of plugins

RKWard supports several mechanisms for modularization and re-use of functionality in plugins. File inclusion is one very simple but effective mechanism, which can be used in the ECMAScript files, but is also supported in the XML files. In script files, this is most useful by defining common functions in an included file. For the XML files, the equivalent is to define "snippets" in the included file, which can then be inserted.

A third mechanism allows to completely embed one plugin into another. For instance the `plot_options` plugin is used by many plugins in **RKWard**, to provide common plot options such as labels, axis options, and grids. Other plugins can embed it using the `embed`-tag in their XML file (the plugin supports hiding irrelevant options). The generated code portions can be fetched from the ECMAScript file just like any other GUI settings, and inserted into the complete code. Other examples of embedded plugins are options for histograms, barplots, and empirical cumulative distribution function (ECDF) plots (which in turn embed the generic plot options plugin).

6.3. Enforcing a consistent interface

RKWard tries to make it easy to create a consistent interface in all plugins. GUI-wise this is supported by providing high-level GUI elements, and embeddable clients. Also, the standard elements of each dialog (`Submit`, and `Cancel` buttons, on-the-fly code view, etc.) are hard coded. Up to version 0.5.3 of **RKWard** it was not possible to use any GUI elements in plugins which were not explicitly defined for this purpose. In the current development version, theoretically, all GUI elements available from **Qt** can be inserted, where necessary.

For generating output, the function `rk.header()` can be used to print a standardized caption for each piece of output. Printing results in vector or tabular form is facilitated by `rk.results()`. A wide range of objects can be printed using `rk.print()`, which is just a thin wrapper around the `HTML()` function of the **R2HTML** package (Lecoutre 2003) in the current implementation. The use of custom formatting with HTML is possible, but discouraged. Standard elements such as a horizontal separator, and the `Run again` link (see the section on the results output in the main article) are inserted automatically, without the need to define them for each plugin.

Regarding the style of the generated R code, enforcing consistency is harder, but plugins which are to become part of the official **RKWard** application are reviewed for adherence to some guidelines. Perhaps the most important guidelines are

- Write readable code, which is properly indented, and commented where necessary.
- Do not hide any relevant computations from the user by performing them in the ECMAScript. Rather, generate R code which will perform those computations, transparently.
- Plugins can be restricted to accept only certain types of data (such as only one-dimensional numeric data). Use such restrictions where appropriate to avoid errors, but be very careful not to add too many of them.

6.4. Handling of R package dependencies

A wide range of plugins for diverse functionality is present in **RKWard**, including plots (e. g., boxplot) or standard tests (e. g., Student's *t* test)[8]. Some of the plugins depend on R packages other than the recommended R base packages. Examples herein are the calculation of kurtosis, skewness or the exact Wilcoxon test.

RKWard avoids loading all these packages pro-actively, as **Rcmdr** does. Rather, plugins which depend on a certain package simply include an appropriate call to `require()` in the pre-processing section of the generated R code. The `require()` function is overloaded in **RKWard**, in order to bring up the package-installation dialog whenever needed. Packages invoked by `require()` remain loaded in the active **RKWard** session unless unloaded manually (from the workspace browser, or using the R function `detach()`).

Dependencies between (embedded) plugins are handled using the `<require>`-tag in the plugin map.

6.5. Development process

RKWard *core and external plugins*

Newly developed plugins are placed in a dedicated plugin map file[9]. Plugins in this map are not visible to the user by default, but need to be enabled manually. Once the author(s) of a plugin announces that they consider it stable, the plugin is subjected to a review for correctness, style, and usability. The review status is tracked in the project wiki. Currently at least one positive review is needed before the plugin is allowed to be made visible by default, by moving it to an appropriate plugin map.

[8] At the time of this writing, there are 164 user-accessible plugins in **RKWard**. Listing all is beyond the scope of this article.

[9] `under_development.pluginmap`

With the release of version 0.5.5, **RKWard** gained support for downloading additional sets of plugins directly from the internet. By simply clicking an `Install` button in a graphical dialog (`Settings`→`Configure RKWard`→`Plugins`), an external plugin set is downloaded, unpacked and its plugin map added to **RKWard**'s configuration, so it becomes instantly available after the configuration dialog is closed. External plugin sets are neither officially included nor supported by the **RKWard** developers. However, they allow plugin developers to easily extend **RKWard** with state-of-the-art or highly specialized features. To achieve this, **RKWard** (version 0.5.6) draws on **KNewStuff2**, a **KDE** library providing support for **GHNS**[10].

Automated testing

A second requirement for new plugins is that each plugin must be accompanied by at least one automated test. The automated testing framework in **RKWard** consists of an R package, **rkwardtests**, providing a set of R functions which allow to run a plugin with specific GUI settings, automatically. The resulting R code, R messages, and output are then compared to a defined standard. Automated tests are run routinely after changes in the plugin infrastructure, and before any new release.

The automated testing framework is also useful in testing some aspects of the application which are not implemented as plugins, but this is currently limited to very few basic tests.

7. Extending RKWard – an example of creating a plugin

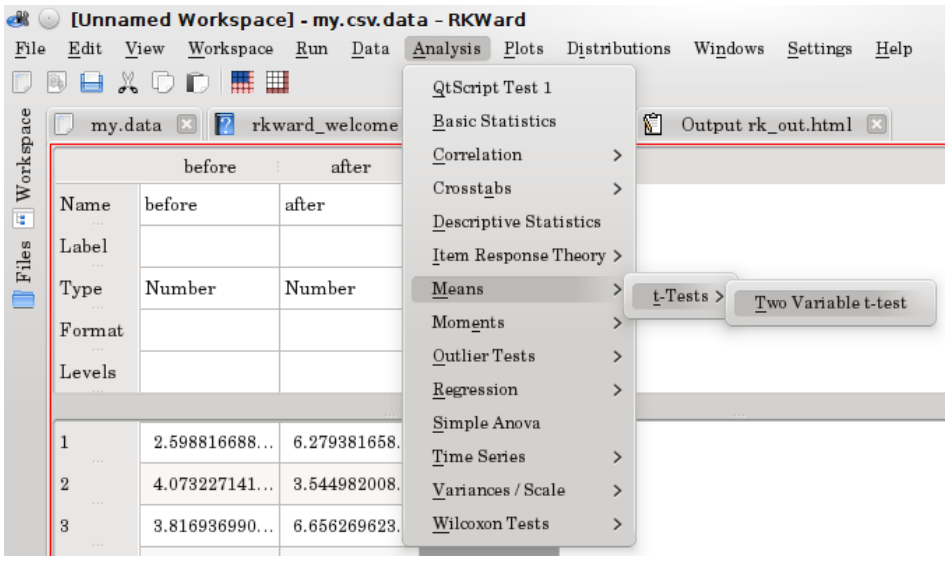
As discussed in Section 6, plugins in **RKWard** are defined by four separate files (Figure 2). To give an impression of the technique, this section shows (portions of) the relevant files for a plugin that provides a simple dialog for a Student's t test. For brevity, the help-file is omitted.

7.1. Defining the menu hierarchy

A so called `.pluginmap` file declares each plugin, and, if appropriate, defines where it should be placed in the menu hierarchy. Usually each `.pluginmap` file declares many plugins. In this example we only show one, namely, a two variable Student's t test (see Figure 3). The pluginmap (`<!DOCTYPE rkpluginmap>`) gives a unique identifier ("id"), the location of the GUI description ("file"), and the window title ("label"). The menu layout is defined in a hierarchical structure by nesting `<menu>` elements to form toplevel menus and submenus. Menus with the same "id" are merged across `.pluginmap` files. Moreover, the position within the menu can be explicitly defined (attribute "index"). This might be required if the menu entries are to be ordered non-alphabetically.

```
<!DOCTYPE rkpluginmap>
<document base_prefix="" namespace="rkward">
  <components>
    <component type="standard" id="t_test_two_vars"
          file="demo_t_test_two_vars.xml" label="Two Variable t-test" />
  </components>
  <hierarchy>
```

[10] **GHNS** (Get Hot New Stuff) is a technology platform (software and specifications) for desktop users to share their work. It is hosted under the umbrella of the freedesktop.org project at `http://ghns.freedesktop.org`. In future versions of **RKWard**, this framework will be deprecated in favor of standard R packages.



	before	after
Name	before	after
Label		
Type	Number	Number
Format		
Levels		
1	2.598816688…	6.279381658.
2	4.073227141…	3.544982008.
3	3.816936990…	6.656269623.

Figure 3: Generated menu structure as defined by the plugin map.

```
    <menu id="analysis" label="Analysis" index="4">
      <menu id="means" label="Means" index="4">
        <menu id="ttests" label="t-Tests">
          <entry component="t_test_two_vars" />
        </menu>
      </menu>
    </menu>
  </hierarchy>
</document>
```

7.2. Defining the dialog GUI

The main XML file of each plugin defines the layout and behavior of the GUI, and references the ECMAScript file that is used for generating R code from GUI settings and the help file (not included in this paper).

GUI logic can be defined directly in the XML file (the `<logic>` element). In this example, the `Assume equal variances` checkbox is only enabled for paired sample tests. Optionally, GUI behavior can also be scripted in ECMAScript.

The XML file defines the Student's t test plugin (`<!DOCTYPE rkplugin>`) to be organized in two tabs[11]. On the first tab, two variables can be selected (`<varslot .../>`). These are set to be `required`, i. e., the `Submit` button will remain disabled until the user has made a valid selection for both. The second tab includes some additional settings like the confidence level (default 0.95).

```
<!DOCTYPE rkplugin>
<document>
  <code file="demo_t_test_two_vars.js"/>
  <help file="demo_t_test_two_vars.rkh"/>
```

[11] A screenshot of the resulting dialog can be found in the main article.

```
  <logic>
    <connect client="varequal.enabled" governor="paired.not"/>
  </logic>

  <dialog label="Two Variable t-Test">
    <tabbook>
      <tab label="Basic settings" id="tab_variables">
        <row id="basic_settings_row">
          <varselector id="vars"/>
          <column>
            <varslot type="numeric" id="x" source="vars" required="true"
              label="compare"/>
            <varslot type="numeric" id="y" source="vars" required="true"
              label="against"/>
            <radio id="hypothesis" label="using test hypothesis">
              <option value="two.sided" label="Two-sided"/>
              <option value="greater" label="First is greater"/>
              <option value="less" label="Second is greater"/>
            </radio>
            <checkbox id="paired" label="Paired sample" value="1" value_unchecked="0" />
          </column>
        </row>
      </tab>
      <tab label="Options" id="tab_options">
        <checkbox id="varequal" label="assume equal variances" value="1"
          value_unchecked="0"/>
        <frame label="Confidence Interval" id="confint_frame">
          <spinbox type="real" id="conflevel" label="confidence level" min="0" max="1"
            initial="0.95"/>
          <checkbox id="confint" label="print confidence interval" value="1"
            checked="true"/>
        </frame>
        <stretch/>
      </tab>
    </tabbook>
  </dialog>
</document>
```

7.3. Generating R code from GUI settings

A simple ECMAScript script is used to generate R code from GUI settings (using `echo()` commands). Generated code for each plugin is divided into three sections: “Preprocess”, “Calculate”, and “Printout”, although each may be empty.

```
var x;
var y;
var varequal;
var paired;

function preprocess () {
  x = getValue ("x");
  y = getValue ("y");

  echo ('names <- rk.get.description (' + x + ", " + y + ')\n');
}
```

```
function calculate () {
  varequal = getValue ("varequal");
  paired = getValue ("paired");

  var conflevel = getValue ("conflevel");
  var hypothesis = getValue ("hypothesis");

  var options = ", alternative=\"" + hypothesis + "\"";
  if (paired) options += ", paired=TRUE";
  if ((!paired) && varequal) options += ", var.equal=TRUE";
  if (conflevel != "0.95") options += ", conf.level=" + conflevel;

  echo ('result <- t.test (' + x + ", " + y + options + ')\n');
}

function printout () {
  echo ('rk.header (result\$method, \n');
  echo ('  parameters=list ("Comparing", paste (names[1], "against", names[2]),\n');
  echo ('  "H1", rk.describe.alternative (result)');
  if (!paired) {
    echo (',\n');
    echo ('  "Equal variances", "');
    if (!varequal) echo ("not");
    echo (' assumed"');
  }
  echo ('))\n');
  echo ('\n');
  echo ('rk.results (list (\n');
  echo ('  \'Variable Name\'=names,\n');
  echo ('  \'estimated mean\'=result\$estimate,\n');
  echo ('  \'degrees of freedom\'=result\$parameter,\n');
  echo ('  t=result\$statistic,\n');
  echo ('  p=result\$p.value');
  if (getValue ("confint")) {
    echo (',\n');
    echo ('  \'confidence interval percent\'=(100 * attr(result\$conf.int, "conf.level")),\n');
    echo ('  \'confidence interval of difference\'=result\$conf.int ');
  }
  echo ('))\n');
}
```

Acknowledgments

The software **RKWard**, presented in this paper, is currently developed by Thomas Friedrichsmeier (lead developer), Prasenjit Kapat, Meik Michalke, and Stefan Rödiger. Many more people have contributed, or are still contributing to the project in various forms. We would like to thank (in alphabetical order) Adrien d'Hardemare, Daniele Medri, David Sibai, Detlef Steuer, Germán Márquez Mejía, Ilias Soumpasis, Jannis Vajen, Marco Martin, Philippe Grosjean, Pierre Ecochard, Ralf Tautenhahn, Roland Vollgraf, Roy Qu, Yves Jacolin, and many more people on `rkward-devel@lists.sourceforge.net` for their contributions.

The first two authors of this article have contributed equally, and both are available for correspondence pertaining to this article. Questions and comments regarding the software **RKWard** should be addressed to the project's main mailing list, `rkward-devel@lists.sourceforge.`

net.

References

Cullmann C (n.d.). ***KatePart***. URL http://kate-editor.org/about-katepart/.

Ettrich M, Taylor O (2002). *XEmbed Protocol Specification Version 0.5.* URL http://standards.freedesktop.org/xembed-spec/xembed-spec-latest.html.

Faure D (2000). "Creating and Using Components (KParts)." In D Sweet (ed.), ***KDE** 2.0 Development*. Sams, Indianapolis, IN, USA.

Friedrichsmeier T, Michalke M (2011). *Introduction to Writing Plugins for **RKWard***. URL http://rkward.sourceforge.net/documents/devel/plugins/index.html.

Helbig M, Urbanek S, Fellows I (2011). ***JGR**: Java Gui for R.* R Package Version 1.7-9, URL http://CRAN.R-project.org/package=JGR.

Jarvis S (2010). "KDE 4 on Windows." *Linux J.*, **2010**. ISSN 1075-3583.

KDE eV (2012). *About **KDE***. Berlin. URL http://www.kde.org/community/whatiskde/.

KOfficeOrg (2010). ***KWord***. URL http://www.koffice.org/kword/.

Lecoutre E (2003). "The **R2HTML** Package – Formatting HTML Output on the Fly or by Using a Template Scheme." *R News*, **3**(3), 33–36.

Nokia Corporation (2012). ***Qt** – Cross-Platform Application and UI Framework*. Oslo. URL http://qt.nokia.com.

Plate T (2009). ***trackObjs**: Track Objects.* R Package Version 0.8-6, URL http://CRAN.R-project.org/package=trackObjs.

Raaphorst S (2003). "A Usability Inspection of Several Graphical User Interface Toolkits." URL http://www.cs.utoronto.ca/~sr/academic/csi51222paper.pdf.

R Development Core Team (2012a). *R Internals.* R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-14-3.

R Development Core Team (2012b). *Writing R Extensions.* R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-11-9.

Ripley BD (2004). "Lazy Loading and Packages in R 2.0.0." *R News*, **4**(2), 2–4. URL http://CRAN.R-project.org/doc/Rnews/Rnews_2004-2.pdf.

Sarkar D (2008). ***Lattice**: Multivariate Data Visualization with R*. Springer, New York. ISBN 978-0-387-75968-5, URL http://lmdvr.r-forge.r-project.org.

Visne I, Dilaveroglu E, Vierlinger K, Lauss M, Yildiz A, Weinhaeusel A, Noehammer C, Leisch F, Kriegner A (2009). "**RGG**: A General GUI Framework for R Scripts." *BMC Bioinformatics*, **10**(1), 74.

Affiliation:

Stefan Rödiger
Lausitz University of Applied Sciences
Department of Bio-, Chemistry and Process Engineering
AND
Kardiologie-CCM, Charité-Universitätsmedizin Berlin
Germany
E-mail: stefan_roediger@gmx.de
E-mail: rkward-devel@lists.sourceforge.net

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume 49, Issue 9 *Submitted:* 2010-12-28
June 2012 *Accepted:* 2011-05-06