Introduction to Writing Plugins for RKWard

Thomas Friedrichsmeier
Meik Michalke

Contents

List of Tables

Abstract

This is a guide to writing plugins for RKWard.

Chapter 1

Introduction

> NOTE
> Documentation as of RKWard release 0.6.4.

This document describes how to write your own plugins. Note, that at the time of this writing, some of the concepts are not yet set in stone. Therefore, this document should be regarded as an introduction to the current approach, and as a basis for discussion. All sorts of comments are welcome. The documentation has grown quite large over time. Don't let that scare you. We recommend reading through the four basic steps (as outlined, below), to get a basic idea of how things work. After that you may want to skim the table of contents to see which advanced topics could be of relevance to you.

For questions and comments, please write to the RKWard development mailing list.

You do not need to read this in order to use RKWard. This document is about extending RKWard. It is targeted at advanced users, or people willing to help improve RKWard.

Writing a standard plugin is basically a four-step process:

- Placing a new Action in the menu hierarchy
- Describing the looks and behavior of the plugin GUI
- Defining, how R-code is to be generated from the settings, the user makes in the GUI
- Adding a help page to your plugin

Those will be dealt with in turn.

Some advanced concepts may be used in those four steps, but are dealt with in separate chapters, to keep things simple:

- GUI logic
- Embedding Plugins into Plugins
- Useful concepts for creating many series of similar plugins

Also, none of the chapters shows all options, but rather only the basic concepts. A complete reference of options is provided separately.

Chapter 2

Preliminaries: What are plugins in RKWard? How do they work?

Of course the first question you might have is: What portions of RKWard functionality is realized using plugins? Or: What can plugins do?

One way to answer this is: Deselect all `.pluginmap` files under **Settings** → **Configure RKWard** → **Plugins**, and see what's missing. A slightly more helpful answer: Most actual statistics functions accessible via the GUI are realized using plugins. Also, you can create fairly flexible GUIs for all kinds of operations using plugins.

The basic paradigm behind RKWard plugins is the one we'll walk you through in this document: An XML file describes what the GUI looks like. An additional JavaScript file is used to generate R syntax from the GUI settings. That is, plugins do not really have to perform any statistical calculations. Rather plugins generate the R syntax needed to run those calculations. The R syntax is then sent to the R backend for evaluation, and typically a result is shown in the output window.

Read on in the next chapters to see how this is done.

Chapter 3

Creating menu entries

When you create a new plugin, you need to tell RKWard about it. So the first thing to do, is to write a `.pluginmap` file (or modify an existing one). The format of `.pluginmap` is XML. I'll walk you through an example (also of course, be sure you have RKWard configured to load your `.pluginmap` -- **Settings** → **Configure RKWard** → **Plugins**):

> TIP
> After reading this chapter, have a look at the rkwarddev package as well. It provides some R functions to create most of RKWard's XML tags for you.

```
<!DOCTYPE rkpluginmap>
```

The doctype is not really interpreted, but set it to *´´rkpluginmap´´* anyway.

```
<document base_prefix="" namespace="myplugins" id="mypluginmap">
```

The *base_prefix* attribute can be used, if all your plugins reside in a common directory. Basically, then you can omit that directory from the filenames specified below. It safe to leave this at *´´´´*.

As you will see below, all plugins get a unique identifier, *id*. The *namespace* is a way to organize those IDs, and make it less likely to create a duplicate identifier accidentally. Internally, basically the namespace and then a '::' gets prepended to all the identifiers you specify in this `.pluginmap`. In general, if you intend to distribute your plugins in an R package, it is a good idea to use the package name as *namespace* parameter. Plugins shipped with the official RKWard distribution have *namespace=´´rkward´´*.

The *id* attribute is optional, but specifying an id for your `.pluginmap` makes it possible for other people to make their `.pluginmap`s load your `.pluginmap`, automatically (see the section on dependencies).

```
  <components>
```

Components? Aren't we talking about plugins? Yes, but in the future, plugins will be no more than a special class of components. What we do here, then, is to register all components/plugins with RKWard. Let's look at an example entry:

```
    <component type="standard" id="t_test_two_vars" file="t_test_two_vars. ↩
        xml" label="Two Variable t-Test" />
```

First the *type* attribute: Leave this to *´´standard´´* for now. Further types are not yet implemented. The *id* we've already hinted at. Each component has to be given a unique (in its namespace) identifier. Pick one that is easily recognizable. Avoid spaces and any special characters. Those are not banned, so far, but might have special meanings. With the *file* attribute, you specify where the description of the actual plugin itself is located. This is relative to the directory the `.pluginmap` file is in, and the *base_prefix* above. Finally, give the component a label. This label will be shown wherever the plugin is placed in the menu (or in the future perhaps in other places as well).

Typically a `.pluginmap` file will contain several components, so here are a few more:

```
    <component type="standard" id="unimplemented_test" file="means/ ↩
        unimplemented.xml" />
    <component type="standard" id="fictional_t_test" file="means/ttests/ ↩
        fictional.xml" label="This is a fictional t-test" />
    <component type="standard" id="descriptive" file="descriptive.xml"  ↩
        label="Descriptive Statistics" />
    <component type="standard" id="corr_matrix" file="corr_matrix.xml"  ↩
        label="Correlation Matrix" />
    <component type="standard" id="simple_anova" file="simple_anova.xml"  ↩
        label="Simple Anova" />
  </components>
```

Ok, this was the first step. RKWard now knows those plugins exist. But how to invoke them? They need to be placed in a menu hierarchy:

```
  <hierarchy>
    <menu id="analysis" label="Analysis">
```

Right below the **<hierarchy>** tag, you start describing, in which **<menu>** your plugins should go. With the above line, you basically say, that your plugin should be in the **Analysis** menu (not necessarily directly there, but in a submenu). The **Analysis** menu is standard in RKWard, so it does not actually have to be created from scratch. However, if it did not exist yet, using the *label* attribute you'd give it its name. Finally, the *id* once again identifies this **<menu>**. This is needed, so several `.pluginmap` files can place their plugins in the same menus. They do this by looking for a **<menu>** with the given *id*. If the ID does not yet exist, a new menu will be created. Otherwise the entries will be added to the existing menu.

```
      <menu id="means" label="Means">
```

Basically the same thing here: Now we define a submenu to the **Analysis** menu. It is to be called **Means**.

```
        <menu id="ttests" label="t-Tests">
```

And a final level in the menu hierarchy: A submenu of the submenu **Means**.

```
          <entry component="t_test_two_vars" />
```

Now, finally, this is the menu we want to place the plugin in. The **<entry>** tag signals, this actually is the real thing, instead of another submenu. The *component* attribute refers to the *id* you gave the plugin/component above.

```
          <entry component="fictional_t_test" />
        </menu>
        <entry component="fictional_t_test" />
      </menu>
      <menu id="frequency" label="Frequency" index="2"/>
```

In case you have lost track: This is another submenu to the **Analysis** menu. See the screenshot below. We'll skip some of what's not visible, marked with [...].

```
      [...]
    </menu>
    <entry component="corr_matrix"/>
    <entry component="descriptive"/>
    <entry component="simple_anova"/>
  </menu>
```

These are the final entries visible in the screenshots below.

```
  <menu id="plots" label="Plots">
    [...]
  </menu>
```

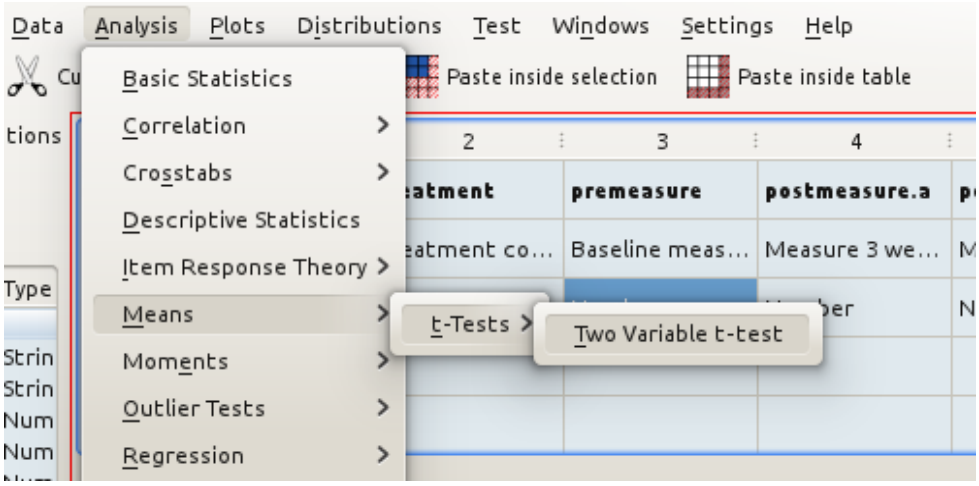
Of course you can also place your plugins in menus other than **Analysis**.

```
  <menu id="file" label="File">
    [...]
  </menu>
```

Even in standard-menus such as **File**. All you need is the correct *id*.

```
  </hierarchy>
</document>
```

That's how to do it. And this screenshot shows the result:



Confused? The easiest way to get started is probably taking some of the existing `.pluginmap` files shipped with the distribution, and modifying them to your needs. Also, if you need help, don't hesitate to write to the development mailing list.

3.1 Controlling the order of menu entries

By default, all items (entries / submenus) inside a menu will be sorted alphabetically, automatically. In *some* cases you may want more control. In this case you can group elements as follows:

- You can define groups inside any menu like this. All elements belonging to the same group will be grouped together:

```
<group id="somegroup"/>
```

- If you want the group to be visually separated from other entries, use:

```
<group id="somegroup" separated="true"/>
```

- Entries, menus, and groups can be appended to a specified group, using:

```
<entry component="..." group="somegroup"/>
```

- In fact, it is also possible to define groups (without separator lines) implicitly:

```
<entry component="first" group="a"/>
<entry component="third"/>
<entry component="second" group="a"/>
```

- Group names are specific to each menu. Group ˝a˝ in menu ˝Data˝ does not conflict with group ˝a˝ in menu ˝Analysis˝, for example.
- The most common use case is defining groups at the top, or at the bottom of a menu. For this, there are pre-defined groups ˝top˝ and ˝bottom˝ in each menu.
- Entries within each group are sorted, alphabetically. Groups appear in the order of declaration (unless appended to another group, of course).
- Menus and entries without group specification logically form a group (˝˝), too.

Chapter 4

Defining the GUI

4.1 Defining a dialog

In the previous chapter you've seen how to register a plugin with RKWard. The most important ingredient was specifying the path to an XML file with a description of what the plugin actually looks like. In this chapter you'll learn how to create this XML file.

> TIP
> After reading this chapter, have a look at the rkwarddev package as well. It provides some R functions to create most of RKWard's XML tags for you.

Once again we'll walk you through an example. The example is a (slightly simplified) version of the two variable t-Test.

```
<!DOCTYPE rkplugin>
```

The doctype is not really interpreted, yet. Set it to *rkplugin*, anyway.

```
<document>
  <code file="t_test_two_vars.js"/>
```

All plugins generate some code. Currently the only way to do so is using JS, as detailed in the next chapter. This defines, where to look for the JS code. The filename is relative to the directory the plugin XML is in.

```
  <help file="t_test_two_vars.rkh"/>
```

It is usually a good idea to also provide a help page for your plugin. The filename of that help page is given, here, relative to the directory, the plugin XML is in. Writing help pages is documented here. If you do not provide a help file, omit this line.

```
  <dialog label="Two Variable t-Test">
```

As you know, plugins may have either a dialog or a wizard interface or both. Here we start defining a dialog interface. The *label* attribute specifies the caption of the dialog.

```
    <tabbook>
      <tab label="Basic settings">
```

GUI elements can be organized using a tabbook. Here we define a tabbook as the first element in the dialog. Use **<tabbook>**[...]**</tabbook>** to define the tabbook and then for each page in the tabbook use **<tab>**[...]**</tab>**. The *label* attribute in the **<tab>** element allows you to specify a caption for that page of the tabbook.

```
        <row id="main_settings_row">
```

The **<row>** and **<column>** tags specify the layout of the GUI elements. Here you say, that you'd like to place some elements side-by-side (left to right). The *id* attribute is not strictly necessary, but we'll use it later on, when adding a wizard interface to our plugin. The first element to place in the row, is:

```
          <varselector id="vars"/>
```

Using this simple tag you create a list from which the user can select variables. You have to specify an *id* for this element, so RKWard knows how to find it.

> WARNING
> You may NOT use a dot (.) in the *id* string.

```
          <column>
```

Next, we nest a **<column>** inside the row. That is the following elements will be placed above each other (top-to-bottom), and all will be to the right of the **<varselector>**.

```
            <varslot types="number" id="x" source="vars" required="true"  ↩
                label="compare"/>
            <varslot types="number" id="y" source="vars" required="true"  ↩
                label="against" i18n_context="compare against"/>
```

These elements are the counterpart to the **<varselector>**. They represent 'slots' into which the user can put variables. You will note that the *source* is set to the same value as the *id* of the **<varselector>**. This means, the **<varslot>**s will each take their variables from the varselector. The **<varslot>**s also have to be given an *id*. They may have a *label*, and they may be set to *required*. This means that the **Submit** button will not be enabled until the **<varslot>** holds a valid value. Finally the *type* attribute is not interpreted yet, but it will be used to take care that only the correct types of variables will be allowed in the **<varslot>**.

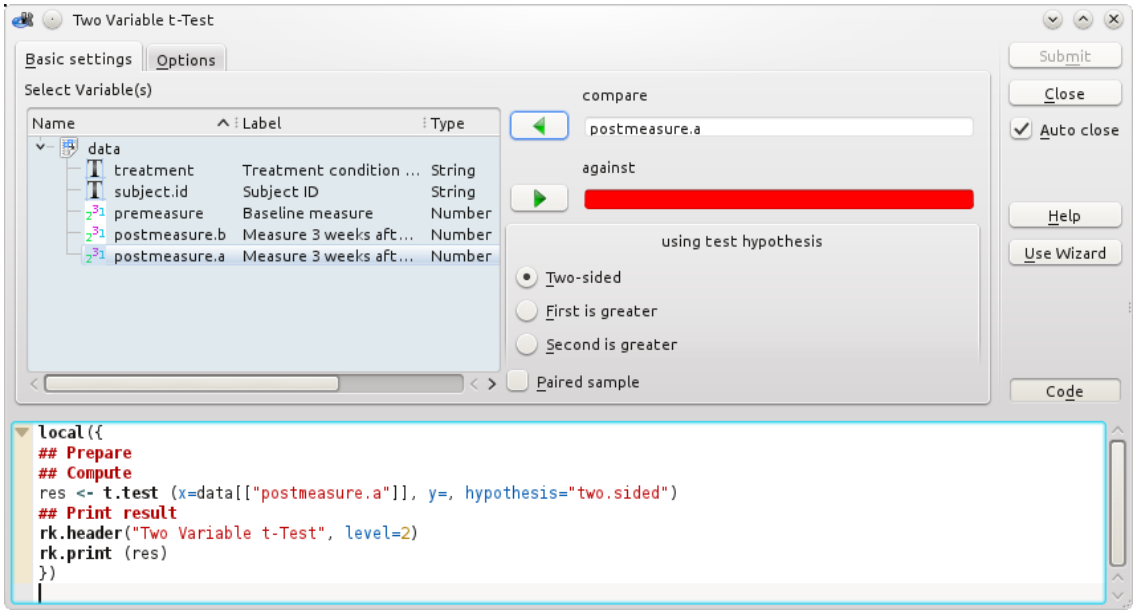
In case you are wondering about the *i18n_context*-attribute: This is to provide context to help the correct translation of the word ˝against˝, used as the **<varslot>**'s label, but does not affect the functionality of the plugin, directly. More on this in a separate chapter.

```
            <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>
```

Here, you define a group of **<radio>** exclusive buttons. The group has a *label* and an *id*. Each **<option>** (button) has a *label* and is assigned a *value*. This is the value the **<radio>** element will return when the option is selected.

```
          </column>
        </row>
      </tab>
```

Each tag has to be closed. We've put all the elements we wanted (the two **<varslots>** and the **<radio>**) in the **<column>**. We put all elements we wanted (the **<varselector>** and the **<column>** with those elements) in the **<row>**. And we've put all the elements we wanted into the first page in the **<tabbook>**. We're not yet done defining the **<tabbook>** (more pages to come), and of course there's more to come in the **<dialog>**, too. But this screenshot is basically what we've done so far:



Note that we have not specified the **Submit**, **Close**, etc. buttons or the code view. Those elements get generated automatically. But of course we still have to define the second page of the **<tabbook>**:

```
    <tab label="Options">
      <checkbox id="varequal" label="assume equal variances" value=", var.equal=TRUE"/>
```

By default elements will be placed top-to-bottom like in a **<column>**. Since that is what we want here, we don't have to explicitly state a **<row>** or **<column>** layout. The first element we define is a checkbox. Just like the **<radio><option>**s, the checkbox has a *label* and a *value*. The *value* is what gets returned, if the check box is checked. Of course the checkbox also needs an *id*.

```
      <frame label="Confidence Interval" id="frame_conf_int">
```

Here's yet another layout element: In order to signal that the two elements below belong together, we draw a **<frame>** (box). That frame may have a *label* (caption). Since the frame is just a passive layout element, it does not need an *id*, we still define one here, as we'll refer to it later, when defining an additional wizard interface.

```
        <checkbox id="confint" label="print confidence interval" value="1" checked="true"/>
        <spinbox type="real" id="conflevel" label="confidence level" min="0" max="1" initial="0.95"/>
      </frame>
```

Inside the **<frame>** we place another **<checkbox>** (using *checked=''true''*, we signal that check box should be checked by default), and a **<spinbox>**. The spinbox allows the user to select a value between *''min''* and *''max''* with the default/initial value *''0.95''*. Setting the *type* to *''real''* signals that real numbers are accepted as opposed to *type=''integer''* which would accept integers only.

> NOTE
> It is also possible, and often preferable, to make the **<frame>** itself checkable, instead of adding a **<checkbox>** inside. See the reference for details. This is not done here, for illustrational purposes.

```
    </tab>
  </tabbook>
</dialog>
```

That's all for the second page of the **<tabbook>**, all pages in the **<tabbook>** and all elements in the **<dialog>**. We're finished defining what the dialog looks like.

```
</document>
```

Finally we close the **<document>** tag, and that's it. The GUI is defined. You can save the file now. But how does R syntax get generated from the GUI-settings? We'll deal with that in the next chapter. First, however, we'll look into adding a wizard interface, and some general considerations.

4.2 Adding a wizard interface

Actually we don't have to define an additional **<wizard>** interface, but here's how that would be done. To add a wizard interface, you'll add a **<wizard>** tag at the same level as the **<dialog>** tag:

```
  <wizard label="Two Variable t-Test">
    <page id="firstpage">
      <text>As a first step, select the two variables you want to compare ↩
          against
        each other. And specify, which one you theorize to be greater.  ↩
            Select two-sided,
        if your theory does not tell you, which variable is greater.</ ↩
            text>
      <copy id="main_settings_row"/>
    </page>
```

Some of this is pretty self explanatory: We add a **<wizard>** tag with a *label* for the wizard. Since a wizard can hold several pages that are shown one after another, we next define the first **<page>**, and put an explanatory **<text>** note in there. Then we use a **<copy>** tag. What this does, is really it saves us having to define yet again, what we already wrote for the **<dialog>**: The copy tag looks for another tag with the same *id* earlier in the XML. This happens to be defined in the **<dialog>** section, and is a **<row>** in which there are the **<varselector>**, **<varslots>** and the 'hypothesis' **<radio>** control. All of this is copied 1:1 and inserted right at the **<copy>** element.

Now to the second page:

```
    <page id="secondpage">
      <text>Below are some advanced options. It's generally safe not to  ↩
          assume the
        variables have equal variances. An appropriate correction will be ↩
            applied then.
        Choosing "assume equal variances" may increase test-strength,  ↩
          however.</text>
      <copy id="varequal"/>
      <text>Sometimes it's helpful to get an estimate of the confidence  ↩
          interval of
        the difference in means. Below you can specify whether one should ↩
            be shown, and
```

```
        which confidence-level should be applied (95% corresponds to a 5% ↩
            level of
        significance).</text>
      <copy id="frame_conf_int"/>
    </page>
  </wizard>
```

Much of the same thing here. We add some texts, and in between that **<copy>** further sections from the dialog interface.

You may of course make the wizard interface look very different to the plain dialog, and not use the **<copy>** tag at all. Be sure, however, to assign corresponding elements the same *id* in both interfaces. This is not only used to transfer settings from the dialog interface to the wizard interface and back, when the user switches interfaces (which does not yet happen in the current version of RKWard), but also simplifies writing your code template (see below).

4.3 Some considerations on GUI design

This section contains some general considerations on which GUI elements to use where. If this is your first attempt of creating a plugin, feel free to skip over this section, as it isn't relevant to getting a basic GUI working. Come back here, later, to see, whether you can refine your plugin's GUI in some way or another.

4.3.1 <radio> vs. <checkbox> vs. <dropdown>

The three elements **<radio>**, **<checkbox>**, **<dropdown>**, all serve a similar function: To select one out of several options. Obviously, a check box only allows to choose between two options: checked or not checked, so you cannot use it, if there are more than two options to choose from. But when to use which of the elements? Some rules of thumb:

If you find yourself creating a **<radio>** or **<dropdown>** with only two options, ask yourself, whether the question is essentially a yes / no type of question. E.g. a choice between 'adjust results' and 'do not adjust results', or between 'remove missing values' and 'keep missing values'. In this case a **<checkbox>** is the best choice: It uses little space, will have the least words of labels, and is easiest to read for the user. There are very few situations where you should choose a **<radio>** over a **<checkbox>**, when there are only two options. An example of that might be: 'Method of calculation: 'pearson'/'spearman''. Here, more methods might be thinkable, and they don't really form a pair of opposites.

Choosing between a **<radio>** and a **<dropdown>** is mostly a question of space. The **<dropdown>** has the advantage of using little space, even if there are a lot of options to choose from. On the other hand, a **<radio>** has the advantage of making all possible choices visible to the user at once, without clicking on the dropdown arrow. Generally, if there are six or more options to choose from, a **<dropdown>** is preferable. If there are five or less options, a **<radio>** is the better choice.

Chapter 5

Generating R code from GUI settings

5.1 Using JavaScript in RKWard plugins

Now we have a GUI defined, but we still need to generate some R code from that. For that, we need another text file, `code.js`, located in the same directory as the `description.xml`. You may or may not be familiar with JavaScript (or, to be technically precise: ECMA-script). Documentation on JS can be found in abundance, both in printed form, and on the Internet (e.g.: https://developer.mozilla.org/en/Core_JavaScript_1.5_Guide). But for most purposes you will not need to know much about JS at all, as we'll only use some very basic features.

> **TIP**
> After reading this chapter, have a look at the rkwarddev package as well. It provides some R functions to create JavaScript code commonly used in RKWard. It can also autodetect variables used in a plugin XML file and create basic JavaScript code from that for you to start with.

> **NOTE**
> Plugin .js files are assumed to be UTF-8 encoded. Be sure to check you editor's encoding, if using any non-ascii characters.

For the two variable t-test, the `code.js` file looks as follows (with comments in between):

5.1.1 preprocess()

```
function preprocess () {
}
```

The JS file is organized into three separate functions: `preprocess()`, `calculate()`, and `printout()`. This is because not all code is needed at all stages. Currently the preprocess-function is not really used in many places (typically you will omit it altogether).

5.1.2 calculate()

```
function calculate () {
  echo ('res <- t.test (x=' + getString ("x") + ', y=' + getString ("y") +  ↩
      ', hypothesis="' + getString ("hypothesis") + '"' + getString (" ↩
      varequal"));
  var conflevel = getString ("conflevel");
  if (conflevel != "0.95") echo (', conf.level=' + conflevel);
  echo (')\n');
}
```

This function generates the actual R syntax to be run from the GUI settings. Let's look at this in detail: The code to be used is generated using `echo()` statement. Looking at the `echo()` statement step by step, the first part of it is

```
res <- t.test (
```

as plain text. Next we need to fill in the value, the user selected as the first variable. We fetch this using `getString (´´x´´)`, and append it to the string to be 'echoed'. This prints out the value of the GUI-element with `id=´´x´´`: our first **<checkbox>**. Next, we append a ', ', and do the same to fetch the value of the element `´´y´´`- the second **<checkbox>**. For the hypothesis (the **<radio>** group), and the equal variances **<checkbox>**, the procedure is very similar.

Note that instead of concatenating the output snippets with '+', you can also use several `echo()` statements. Everything is printed on a single line. To produce a line break in the generated code, insert a `´´\n´´` in the echoed string. In theory, you can even produce many lines with a single echo-statement, but please keep it to one line (or less) of generated code per `echo()`.

> **NOTE**
> Besides `getString()`, there are also functions `getBoolean()`, which will try to return the value as a logical (suitable for using in an `if()`-statement), and `getList()`, which will try to return list-like data in a JS `Array()`. We will show examples of those, later.
> When looking at existing plugins, you will also find plenty of plugins using `getValue()`, instead of `getString()`, and in fact the two are *almost* identical. However using `getString()`, `getBoolean()` and `getList()` is the recommended practice since version 0.6.1.

It gets a little more tricky for the confidence level. For reasons of aesthetics, we don't want to explicitly specify the confidence level to use, if it corresponds to the default value. Hence, instead of printing the value unconditionally, we first fetch into a variable. Then we check, whether that variable differs from `´´0.95´´` and if so print out an additional argument. Finally, we echo a closing bracket and a line break: `´´)\n´´`. That's all for the calculate function.

5.1.3 printout()

```
function printout () {
  echo ('rk.header (' + i18n ("Two Variable t-Test") + ')\n');
  echo ('rk.print (res)\n');
}
```

And this was all there is to the printout function in most cases. `rk.header()` prints a standard headline for the results. Note that in the .js files, you have to mark up all translatable strings by hand, using **i18n()**, or some alternative commands. More on this in the chapter on internationalization. You can also add some more information to this, if you like, e.g.:

```
function printout () {
  new Header (i18n ("Two Variable t-Test"))
            .addFromUI ("varequal")
            .add (i18n ("Confidence level"), getString ("conflevel"))  //  ↩
                Note: written like this for illustration purposes. More  ↩
                automatic:
  //        .addFromUI ("conflevel")
            .print ();
echo ('rk.print (res)\n');
}
```

`rk.print()` utilizes the R2HTML package to provide HTML formatted output. Another helpful function is `rk.results()`, which can also output different kinds of result tables. If in doubt, however, just use `rk.print()`, and be done with. The JS class `Header` is a JS level helper to generate a call to `rk.header()` (just take a look at the generated R code). In some cases you may want to call `echo ('rk.header (...)')` directly to print a header for your output.

Note that internally, the output is just a plain HTML document at this point of time. Therefore you might be tempted to add custom HTML using `rk.cat.output()`. While this will work, please don't do this. The output format may change (e.g. to ODF) in the future, so it's best not to introduce HTML specific code. Rather keep things simple with `rk.header()`, `rk.print()`, `rk.results()`, and -- if needed -- `rk.print.literal()`. If those don't seem to satisfy your formatting needs, contact us on the mailing list for help.

Congratulations! You created your first plugin. Read on in the next chapters for more advanced concepts.

5.2 Conventions, policies, and background

There are many ways to write R code for a certain task, and there are even more ways to generate this R code from JS. How exactly you do it, is left up to you. Still there are a number of considerations that you should follow, and background information you should understand.

5.2.1 Understanding the `local()` environment

More often than not you will have to create one or more temporary R objects in the code generated by your plugin. Normally, you do not want those to be placed in the user's workspace, potentially even overwriting user variables. Hence, all plugin generated code is run in a `local()` environment (see R help page on function `local()`). This means, all variables you create are temporary and will not be saved permanently.

If the user explicitly asks for a variable to be saved, you will need to assign to that object using `.GlobalEnv$objectname <- value`. In general, do not use the `<<-` operator. It will not necessarily assign in .GlobalEnv.

One important pitfall is using `eval()`. Here, you need to note that eval will by default use the current environment for evaluation, i.e. the local one. This will work well most of the times, but not always. Thus, if you need to use `eval()`, you will probably want to specify the *`envir`* parameter: `eval(..., envir=globalenv())`.

5.2.2 Code formatting

The most important thing is for your generated R code to work, but it should be also easy to read. Therefore, please also keep an eye on formatting. Some considerations:

Normal top-level R statements should be left aligned.

Statements in a lower block should be indented with one tab (see example below).

If you do very complex calculations, add a comment here and there, esp. to mark up logical sections. Note that there is a dedicated function **comment()** for inserting translatable comments in the generated code.

For example, generated code might look like this. The same code without indentation or comments would be pretty hard to read, despite its modest complexity:

```
# first determine the wobble and rotation
my.wobble <- wobble (x, y)
my.rotation <- wobble.rotation (my.wobble, z)

# boggling method needs to be chosen according to rotation
if (my.rotation > wobble.rotation.limit (x)) {
  method <- "foo"
  result <- boggle.foo (my.wobble, my.rotation)
} else {
  method <- "bar"
  result <- boggle.bar (my.wobble, my.rotation)
}
```

5.2.3 Dealing with complex options

Many plugins can do more than one thing. For instance, the 'Descriptive Statistics' plugin can compute mean, range, sum, product, median, length, etc. However, typically the user will only choose to have some of those calculations performed. In this case, please try to keep the generated code as simple as possible. It should only contain portions relevant to the options that are actually selected. To achieve this, here is an example of a common design patterns as you would use it (in JS; here, ˝domean˝, ˝domedian˝, and ˝dosd˝ would be <checkbox> elements):

```
function calculate () {
  echo ('x <- <' + getString ("x") + ')\n');
  echo ('results <- list ()\n');

  if (getBoolean ("domean.state")) echo ("results$" + i18n ("Mean value") + ↩
      " <- mean (x)\n");
  if (getBoolean ("domedian.state")) echo ("results$" + i18n ("Median") + " ↩
      <- median (x)\n");
  if (getBoolean ("dosd.state")) echo ("results$" + i18n ("Standard  ↩
     deviation") + " <- sd (x)\n");
  //...
}
```

5.3 Tips and tricks

Here are a few assorted tricks which may make writing plugins less tedious:

If you need the value of a GUI setting at several places in you plugin's code, consider assigning it to a variable in JS, and using that instead of fetching it again and again with `getString()/getBoolean()/getList()`. This is faster, more readable, and less typing all at the same time:

```
function calculate () {
  var narm = "";  // na.rm=FALSE is the default in all functions below
```

```
  if (getBoolean ("remove_nas")) {
    $narm = ", na.rm=TRUE";
  }
  // ...
  echo ("results$foo <- foo (x" + narm + ")\n");
  echo ("results$bar <- bar (x" + narm + ")\n");
  echo ("results$foobar <- foobar (x" + narm "\n");
  // ...
}
```

The simple helper function `makeOption()` can make it easier to omit parameters that are at their default value, in many cases:

```
function calculate () {
  var options
  //...
  // This will do nothing, if VALUE is 0.95 (the default). Otherwise it  ↩
      will append ', conf.int=VALUE' to options.
  options += makeOption ("conf.int", getString ("confint"), "0.95");
  //...
}
```

Chapter 6

Writing a help page

When your plugin basically works, the time has come to provide a help page. While typically you will not want to explain all the underlying concepts in depth, you may want to add some more explanation for some of the options, and link to related plugins and R functions.

> **TIP**
> After reading this chapter, have a look at the rkwarddev package as well. It provides some R functions to create most of RKWard's XML tags for you. It's also capable of creating basic help file skeletons from existing plugin XML files for you to start with.

You may recall putting this inside your plugin XML (if you haven't put this in, do so now):

```
<document>
  [...]
  <help file="filename.rkh" />
  [...]
</document>
```

Where, obviously, you'd replace `filename` with a more appropriate name. Now it's time to create this `.rkh` file. Here is a self-descriptive example:

```
<!DOCTYPE rkhelp>
<document>
  <summary>
In this section, you'll put some short, very basic information about what ↩
    the plugin does.
This section will always show up on the very top of the help page.
  </summary>

  <usage>
The usage section may contain a little more practical information. It does ↩
    not explain all
the settings in detail (that's done in the "settings" section), however.

To start a new paragraph, insert an empty line, as shown above.
This line, in contrast, will be in the same paragraph.

In all sections you can insert some simple HTML code, such as <b>bold</b> ↩
    or
<i>italic</i> text. Please keep formatting to the minimum needed, however.
```

```
The usage section is always the second section shown in a help page.
  </usage>

  <section id="sectionid" title="Generic section" short_title="Generic">
If you need to, you can add additional sections between the usage and settings sections.
However, usually, you will not need this while documenting plugins. The "id"-attribute
provides an anchor point to jump to this section from the navigation menu. The "short_title"
attribute provides a short title to use in the navigation bar. This is optional, by default
the main "title" will be used both as a heading to the section, and as the link name in the
navigation bar.

In any section you may want to insert links to further information. You do this by adding

<link href="URL">link name</link>

Where URL could be an external link such as http://rkward.kde.org .
Several special URLs are supported in the help pages:

<link href="rkward://page/path/page_id"/>

This links to a top level rkward help page (not for a plugin).

<link href="rkward://component/[namespace/]component_id"/>

This links to the help page of another plugin. The [namespace/] part may be omitted
(in this case, rkward is assumed as the standard namespace, e.g.:
<link href="rkward://component/import_spss"/> or
<link href="rkward://component/rkward/import_spss"/> are equivalent).
The component_id is the same that you specified in the .pluginmap.

<link href="rkward://rhelp/rfunction"/>

Links to the R help page on "rfunction".

Note that the link names will be generated automatically for these types of links.
  </section>

  <settings>
    <caption id="id_of_tab_or_frame"/>
    <setting id="id_of_element">
Description of the GUI element identified by the given id
    </setting>
    <setting id="id_of_elementb" title="description">
Usually the title of the GUI element will be extracted from the
XML definition of the plugin, automatically. However,
for some GUI elements, this description may not be enough to identify them, reliably.
In this case, you can add an explicit title using the "title" attribute.
    </setting>
```

```
    <setting id="id_of_elementc">
Description of the GUI element identified by "id_of_elementc"
    </setting>
    [...]
  </settings>

  <related>
The related section typically just contains some links, such as:

<ul>
  <li><link href="rkward://rhelp/mean"/></li>
  <li><link href="rkward://rhelp/median"/></li>
  <li><link href="rkward://component/related_component"/></li>
</ul>
  </related>

  <technical>
The technical section (optional, always last) may contain some technical details of the plugin
implementation, which are of interest only to RKWard developers. This is particularly relevant
for plugins that are designed to be embedded in many other plugins, and could detail, which
options are available to customize the embedded plugin, and which code sections contain which
R code.
  </technical>
</document>
```

Chapter 7

Logic interactions between GUI elements

7.1 GUI logic

All the basic concepts of creating a plugin for RKWard have been described in the previous chapters. Those basic concepts should be sufficient for many -- if not most -- cases. However, sometimes you want more control over how your plugin's GUI behaves.

For instance, suppose you want to extend the t-test example used in this documentation to allow both: comparing a variable against another variable (as shown), and comparing a variable against a constant value. Now, one way of doing this would be to add a radio-control that switches between the two modes, and adding a spinbox to enter the constant value to compare against. Consider this simplified example:

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

  <dialog label="T-Test">
    <row>
      <varselector id="vars"/>
      <column>
        <varslot id="x" types="number" source="vars" required="true" label ↩
            ="compare"/>
        <radio id="mode" label="Compare against">
          <option value="variable" checked="true" label="another variable ( ↩
              select below)"/>
          <option value="constant" label="a constant value (set below)"/>
        </radio>
        <varslot id="y" types="number" source="vars" required="true" label ↩
            ="variable" i18n_context="Noun; a variable"/>
        <spinbox id="constant" initial="0" label="constant" i18n_context=" ↩
            Noun; a constant"/>
      </column>
    </row>
  </dialog>
</document>
```

So far so good, but there are a number of problems with this GUI. First, both the varslot and the spinbox are always shown, whereas only one of the two is really used. Worse, the varslot

always requires a valid selection, even if you compare against a constant. Obviously, if we create a multi-purpose GUI like this, we want more flexibility. Enter: the **<logic>** section (inserted at the same level as **<code>**, **<dialog>**, or **<wizard>**).

```
[...]
<code file="code.js"/>

<logic>
  <convert id="varmode" mode="equals" sources="mode.string" standard=" ↩
      variable" />

  <connect client="y.visible" governor="varmode" />
  <connect client="constant.visible" governor="varmode.not" />
</logic>

<dialog label="T-Test">
[...]
```

The first line inside the logic section is a **<convert>** tag. Basically, this provides a new boolean (on or off, true or false) property, which can be used later on. This property (´´*varmode*´´) is true, whenever the upper radio button is selected and false whenever the lower radio button is selected. How is this done?

First, under *sources*, the source properties to work on are listed (in this case only one each; you could list several as *sources=´´mode.string;somethingelse´´*, then ´´*varmode*´´ would only be true, if both ´´*mode.string*´´ and ´´*somethingelse*´´ are equal to the string ´´*variable*´´). Note that in this case we don't just write ´´*mode*´´ (as we would in `getString(´´mode´´)`), but ´´*mode.string*´´. This is actually the internal way a radio control works: It has a property 'string', which holds its string value. `getString(´´mode´´)` is just a shorthand, and equivalent to `getString(´´mode.string´´)`. See the reference for all properties of the different GUI elements.

Second, we set the mode of conversion to *mode=´´equals´´*. This means, we want to check, whether the source(s) is (are) equal to a certain value. Finally standard is the value to compare against, so with *standard=´´variable´´*, we check whether the property ´´*mode.string*´´ is equal to the string ´´*variable*´´ (the value of the upper radio option). If it is equal, then the property varmode is true, else it is false.

Now to the real stuff: We **<connect>** the ´´*varmode*´´ property to y.visible, which controls whether the varslot ´´*y*´´ is shown or not. Note that any element which is made invisible is implicitly non-required. Thus, if the upper radio-option is selected, the varslot ´´*y*´´ is required, and visible. Else it is not required and hidden.

For the spinbox, we want the exact reverse. Fortunately, we do not need another **<convert>** for this: Boolean properties can be negated very easily by appending the modifier ´´*not*´´, so we **<connect>** ´´*varmode.not*´´ to the spinbox's visibility property. In effect, either the varslot is shown and required, *or* the spinbox is shown and required - depending on which option is selected in the radio control. The GUI is changing itself according to the radio option. Try the example, if you like.

For a complete list of properties, refer to the reference. One more property, however, is special in that all GUI elements have it: 'enabled'. This is slightly less drastic than 'visible'. It does not show/hide the GUI element, but only enables/disables it. Disabled elements are typically shown grayed out, and do not react to user input.

> NOTE
> Besides **<convert>** and **<connect>**, there are several further elements for use in the **<logic>** section. E.g. conditional constructs can also be implemented using the **<switch>**-element. Refer to the reference on logic elements for details.

7.2 Scripted GUI logic

While connecting properties as described above is often enough, sometimes it is more flexible or more convenient to use JS to script the GUI logic. In this way, the above example could be re-written as:

```
  [...]
  <code file="code.js"/>
'
  <logic>
    <script><![CDATA[
      // ECMAScript code in this block
      // the top-level statement is only called once
      gui.addChangeCommand ("mode.string", "modeChanged ()");

      // this function is called whenever the "mode" was changed
      modeChanged = function () {
        var varmode = (gui.getString ("mode.string") == "variable");
        gui.setValue ("y.enabled", varmode);
        gui.setValue ("constant.enabled", !varmode);
      }
    ]]></script>
  </logic>

  <dialog label="T-Test">
  [...]
```

The first line of code tells RKWard to call the function `modeChanged()` whenever the value of the *`id=´´mode´´`* radio box changes. Inside this function, we define a helper-variable *`´´varmode´´`* which is true when the mode is *`´´variable´´`*, false as it is *`´´constant´´`*. Then we use `gui.setValue()` to set the 'enabled' properties of *`´´y´´`* and *`´´constant´´`*, in just the same way as we did using **<connect>** statements, before.

The scripted approach to GUI logic becomes particularly useful when you want to change the available option according to the type of object that the user has selected. See the reference for available functions.

Note that the scripted approach to GUI logic can be mixed with **<connect>** and **<convert>**-statements if you like. Also note that the **<script>** tag allows to specify a script file name in addition to or as an alternative to inlining the script code. Typically, inlining the script code as shown above is most convenient, however.

Chapter 8

Embedding Plugins into Plugins

8.1 Use cases for embedding

When writing plugins, you will often find that you're creating a number of plugins that only differ in some respects, but have a lot more in common. For instance, for plotting, there are a number of generic R options that can be used with mostly all types of plots. Should you create a GUI and JS-template for those over and over again?

Obviously that would be quite a hassle. Fortunately, you don't have to do that. Rather you create the common functionality once, and later you can embed it into several plugins. In fact it is possible to embed any plugin into any other plugin, even if the original author of the embedded plugin never thought, somebody would want to embed their plugin into another one.

8.2 Embedding inside a dialog

Ok, enough said. How does it work? Simple: Just use the **<embed>** tag. Here's a stripped down example:

```
<dialog>
  <tabbook>
    <tab [...]>
      [...]
    </tab>
    <tab label="Plot Options" i18n_context="Options concerning the plot">
      <embed id="plotoptions" component="rkward::plot_options"/>
    </tab>
    <tab [...]>
      [...]
    </tab>
  </tabbook>
</dialog>
```

What happens here, is that the entire GUI or the plot options plugin (except of course for the standard elements like **Submit** button, etc.) is embedded right into your plugin (try it!).

As you can see the syntax of the **<embed>**-tag is fairly simple. It takes an *id* as most elements. The parameter component specifies which plugin to embed, as defined in the `.pluginmap` file (*´´rkward::plot_options´´* is the result of concatenating the namespace 'rkward', a separator '::', and the name of the component 'plot_options').

8.3 Code generation when embedding

So far so good, but what about the generated code? How are the code for the embedding and embedded plugin merged? In the embedding plugin's JS code, simply write something like this:

```
function printout () {
  // ...
  echo ("myplotfunction ([...]" + getString ("plotoptions.code.printout");  ↩
      + ")\n");
  // ...
}
```

So essentially, we're fetching the code generated by the embedded plugin just like we're fetching any other GUI setting. Here the string *´´plotoptions.code.printout´´* can be deparsed to: 'The printout section of the generated code of the element with the *id* plotoptions' (plotoptions is the ID we gave for the **<embed>** tag above). And yes, if you want advanced control, you can even fetch the values of individual GUI elements inside the embedded plugin (but not the other way around, as the embedded plugin does not know anything about its surroundings).

8.4 Embedding inside a wizard

If your plugin provides a wizard GUI, embedding works basically in the same way. You'll generally use:

```
  <wizard [...]>
    [...]
    <page id="page12">
      [...]
    </page>
    <embed id="plotoptions" component="rkward::plot_options"/>
    <page id="page13">
      [...]
    </page>
    [...]
  </wizard>
```

If the embedded plugin provides a wizard interface, its pages will be inserted right between *´´page12´´* and *´´page13´´* of your plugin. If the embedded plugin provides a dialog interface only, a single new page will be added between your pages *´´page12´´* and *´´page13´´*. The user will never notice.

8.5 Less embedded embedding: Further Options button

While embedding is cool, you should be careful not to overdo it. Too many functions inside a GUI just make it hard to find the relevant options. Of course, sometimes you may want to embed a great deal of options (like all the options to `plot()`), but as those are really optional, you don't want them prominently in your GUI.

An alternative is to embed those options 'as a button':

```
<dialog>
  <tabbook>
    [...]
    <tab label="Options">
```

```
      [...]
      <embed id="plotoptions" component="rkward::plot_options" as_button=" ↩
          true" label="Specify plotting options"/>
    </tab>
    [...]
  </tabbook>
</dialog>
```

In this case, a single push button will be added to your plugin, labeled **Specify plotting options**. When you press that button, a separate dialog will come up, with all the options of the embedded plugin. Even while this embedded GUI is not visible most of the time, you can fetch its settings just as described above.

> CAUTION
> Probably the 'button' approach should only ever be used for plugins that can never be invalid (for missing/bad settings). Otherwise the user would not be able to submit the code, but might have a hard time finding out, the reason for that is hidden behind some button.

8.6 Embedding/defining incomplete plugins

Some plugins -- and as a matter of fact, the plot_options used as an example above, is one of them -- are not complete by themselves. They simply do not have the GUI elements to select some important values. They are meant to be used only embedded into other plugins.

In how far is the plot_options plugin incomplete? Well, for some option settings, it needs to know the name of the objects/expressions for the x and y axes (actually it will do fine if it only has either, but it needs at least one to function properly). However, it does not have a mechanism of selecting those objects, or entering them any other way. So how does it know about them?

In the logic section of the plot_options plugin there are two additional lines, not covered, yet:

```
  <logic>
    <external id="xvar" />
    <external id="yvar" />

    [...]
  </logic>
```

This defines two additional properties in the plot_options plugin, whose sole purpose is to be connected to some (yet unknown) properties of the embedding plugin. In the plot_options plugin those two properties are simply used like any other, and for instance there are calls to `getString(''xvar'')` in the plot_options JS template.

Now, for the incomplete plugin there is no way of knowing, where it will be embedded, and what the relevant settings in the embedding plugin will be called. So we need to add two additional lines in the embedding plugin's logic section as well:

```
  <logic>
    [...]

    <connect client="plotoptions.xvar" governor="xvarslot.available" />
    <connect client="plotoptions.yvar" governor="yvarslot.available" />
  </logic>
```

This is nothing new in principle, we've covered **<connect>** statements in the chapter of GUI logic. You simply connect the values in two varlots (called *''xvarslot''* and *''yvarslot''* in this example) to the receiving 'external' properties of the embedded plugin. That's it. Everything else is taken care of automatically.

Chapter 9

Dealing with many similar plugins

9.1 Overview on different approaches

Sometimes, you may wish to develop plugins for a series of similar functions. As an example, consider the distribution plots. These generate fairly similar code, and of course it's desirable to make the graphical interfaces look similar to each other. Finally large sections of the help files can be identical. Only a few parameters are different for each plugin.

The naive approach to this is to develop one plugin, then basically copy and paste the entire contents of the `.js`, `.xml`, and `.rkh` files, then changing the few portions that are different. However, what if sometime later you find a spelling mistake that has been copied and pasted to all plugins? What if you want to add support for a new feature? You'd have to visit all plugins again, and change each single one. A tiresome and tedious process.

A second approach would be to use embedding. However, in some cases this does not lend itself well to the problem at hand, mostly because the 'chunks' you can embed are sometimes too large to be useful, and it places some constraints on the layout. For these cases, the concepts including `.js` files, including `.xml` files and snippets can be very useful (but see the thoughts on when it is preferable to use embedding).

One word of caution, before you begin reading, though: These concepts can help making it simpler to deal with many similar plugins, and can improve maintainability and readability of those plugins. However, overdoing it can easily lead to the reverse effect. Use with some caution.

9.2 Using the JS include statement

You can easily include one script file into another in RKWard plugins. The value of this becomes immediately obvious, if some sections of your JS code are similar across plugins. You can simply define those sections in a separate `.js` file, and include this in all the plugin `.js` files. For example, as in:

```
// this is a file called "common_functions.js"

function doCommonStuff () {
  // perhaps fetch some options, etc.
  // ...
  comment ("This is R code you want in several different plugins\n");
  // ...
}
```

```
// this is one of your regular plugin .js files

// include the common functions
include ("common_functions.js");

function calculate () {
  // do something
  // ...

  // insert the common code
  doCommonStuff ();
}
```

Note that sometimes it's even more useful to reverse this, and define the 'skeleton' of `preprocess()`, `calculate()`, and `printout()` functions is a common file, and make these call back for those part which are different across plugins. E.g.:

```
// this is a file called "common_functions.js"

function calculate () {
  // do some things which are the same in all plugins
  // ...

  // add in something that is different across plugins
  getSpecifics ();

  // ...
}
```

```
// this is one of your regular plugin .js files

// include the common functions
include ("common_functions.js");

// note: no calculate() function is defined in here.
// it in the common_functions.js, instead.

function getSpecifics () {
  // print some R code
}
```

One issue you should be aware of when using this technique is variable scoping. See the JS manual on variable scopes.

This technique is heavily used in the distribution plot and distribution CLT plot plugins, so you may want to look there for examples.

9.3 Including `.xml` files

Basically the same feature of including files is also available for use in the `.xml`, `.pluginmap` and `.rkh` files. At any place in these files you can place an **<include>** tag as shown below. The effect is that the entire contents of that XML file (to be precise: everything within the **<document>** tag of that file) is included verbatim at this point in the file. Note that you can only include another XML file.

```
<document>
  [...]
  <include file="another_xml_file.xml"/>
  [...]
</document>
```

The attribute *file* is the filename relative to the directory the current file is located in.

9.4 Using <snippets>

While including files as shown in the previous section is fairly powerful, it become most useful when used in combination with **<snippets>**. Snippets are really smaller sections which you can insert at another point in the file. An example illustrates this best:

```
<document>
  <snippets>
    <snippet id="note">
      <frame>
        <text>
    This will be inserted at two places in the GUI
        </text>
      </frame>
    </snippet>
  </snippets>
  <dialog label="test">
    <column>
      <insert snippet="note"/>
      [...]
      <insert snippet="note"/>
    </column>
  </dialog>
</document>
```

Hence, you define the snippet at one place at the top of the XML file, and then you **<insert>** it at any place(s) you wish.

While this example is not too useful in itself, think about combining this with an **<include>**d `.xml` file. Note that you can also place snippets for the `.rkh` file in the same file. You'd simply **<include>** the file there as well, and **<insert>** the relevant snippet:

```
<!-- This is a file called "common_snippets.xml" -->
<document>
  <snippet id="common_options">
    <spinbox id="something" [...]/>
    [...]
  </snippet>
  <snippet id="common_note">
    <text>An important note for this type of plugin</text>
  </snippet>

  <snippet id="common_help">
    <setting id="something">This does something</setting>
    [...]
  </snippet>
</document>
```

```
<!-- This is the .xml file of the plugin -->
<document>
  <snippets>
    <!-- Import the common snippets -->
    <include file="common_snippets.xml"/>
  </snippets>

  <dialog label="test2">
    <insert snippet="common_note"/>
    <spinbox id="something_plugin_specific" [...] />
    <insert snippet="common_options"/>
  </dialog>
</document>
```

Similar to inclusion in JS, the reverse approach is often even more useful:

```
<!-- This is a file called "common_layout.xml" -->
<document>
  <column>
    <insert snippet="note">
    [...]
    <insert snippet="plugin_parameters">
  </column>
  [...]
</document>
```

```
<!-- This is the .xml file of the plugin -->
<document>
  <snippets>
    <snippet id="note">
      <text>The note used for this specific plugin</text>
    </snippet>

    <snippet id="plugin_parameters">
      <frame label="Parameters specific to this plugin">
        [...]
      </frame>
    </snippet>
  </snippets>

  <dialog label="test3">
    <include file="common_layout.xml"/>
  </dialog>
</document>
```

Finally, it is also possible to **<insert>** snippets into other snippets, provided that: a) there is only one level of nesting, and b) the **<snippets>** section is placed at the top of the file (before a nested snippet is inserted); this is because **<insert>** statements are resolved from top to bottom.

9.5 <include> and <snippets> vs. <embed>

At first glance, **<include>** and **<snippets>** provides functionality rather similar to embedding: It allows to reuse some portions of code across plugins. So what's the difference between these approaches, and when should you use which?

The key difference between these concepts is that embeddable plugins are a more tight bundle. They combine a complete GUI, code to generate R code from this, and a help page. In contrast, include and insert allow much more fine grained control, but at the price of less modularity.

That is, a plugin embedding another plugin will typically not need to know much about the internal details of the embedded plugin. A prime example is the plot_options plugin. Plugins wishing to embed this do not necessarily need to know about all the options provided, or how they are provided. This is a good thing, as otherwise a change in the plot_options plugin might make it necessary to adjust all plugins embedding this (a lot). In contrast, include and insert really exposes all the internal details, and plugins using this will -- for example -- need to know the exact ids and perhaps even the type of the elements used.

Hence the rule of thumb is this: include and insert are great if the relevant options are only needed for a clearly limited group of plugins. Embedded plugins are better, if the group of plugins it may be useful to is not clearly defined, and if the functionality can easily be modularized. Another rule of thumb: If you can put the common portions into a single 'chunk', then do so, and use embedding. If you need lots of small snippets to define the common portions -- well, use **<snippets>**. A final way to look at it: If all plugins provide *highly* similar functionality, includes and inserts are probably a good idea. If they merely share one or two common 'modules', embedding is likely better.

Chapter 10

Concepts for use in specialized plugins

This chapter contains information on some topics that are useful only to certain classes of plugins.

10.1 Plugins that produce a plot

Creating a plot from a plugin is easy to do. However, there are a few subtle gotchas to avoid, and also some great generic functionality that you should be aware of. This section shows you the basic concepts, and concludes with a canonical example that you should follow whenever creating plot plugins.

10.1.1 Drawing a plot to the output window

In order to draw a plot to the output window, use `rk.graph.on()` directly before creating the plot, and `rk.graph.off()`, directly afterwards. This is similar to e.g. calling `postscript()` and `dev.off()` in a regular R session.

Importantly, however, you must *always* call `rk.graph.off()` after calling `rk.graph.on()`. Otherwise the output file will be left in a broken state. To ensure `rk.graph.off()` really gets called, you should wrap *all* R commands between the two calls in `try()` statement. Never heard of that? Don't worry, it's easy. All you need to do is follow the pattern shown in example, below.

10.1.2 Adding preview functionality

> NOTE
> This section discusses adding preview functionality to plugins producing plots. There are separate sections on previews of (HTML) output, previews of (imported) data, and custom previews. However, it is recommended that you read this section first, as the approach is similar in each case.

A very useful feature for all plugins generating a plot/graph is to provide an automatically updating preview. To do so, you will need two things: Adding a **<preview>** check box to your GUI definition, and adjusting the generated code for the preview.

Adding a **<preview>** check box is simple. Just place the following somewhere in your GUI. It will take care of all the behind-the-scenes magic of creating a preview device, updating the preview whenever the setting have changed, etc. Example:

> NOTE
> Starting with version 0.6.5 of RKWard **<preview>** preview elements are special-cased in plugin dialogs (not wizards): They will be placed in the button-column, irrespective of where exactly they are defined in the UI. It is still a good idea to define them at a sensible place in the layout, for backwards compatibility.

```
<document>
  [...]
  <dialog [...]>
    [...]
    <preview id="preview"/>
    [...]
  </dialog>
  [...]
</document>
```

And that's it for the GUI definition.

Adjusting the JS template is a little more work. You will have to create a new function called `preview()` in addition to the `preprocess()`, `calculate()`, etc. functions. This function should generate the code needed to produce the plot, and only that. Esp. no printing of headers, `rk.graphics.on()`, or similar calls. See the example, below for the typical pattern that you will use.

10.1.3 Generic plot options

You will have noticed that most plotting plugins in RKWard provide a wide range of generic options e.g. for customizing axis titles or figure margins. Adding these options to your plugin is easy. They are provided by an embeddable plugin called **rkward::plot_options**. Embed this in your plugin UI like this:

```
<document>
  [...]
  <logic [...]>
    <connect client="plotoptions.xvar" governor="x.available"/>
    <set id="plotoptions.allow_type" to="true"/>
    <set id="plotoptions.allow_ylim" to="true"/>
    <set id="plotoptions.allow_xlim" to="false"/>
    <set id="plotoptions.allow_log" to="false"/>
    <set id="plotoptions.allow_grid" to="true"/>
  </logic>
  <dialog [...]>
    [...]
    <embed id="plotoptions" component="rkward::plot_options" as_button="↩
        true" label="Plot Options"/>
    [...]
  </dialog>
  [...]
</document>
```

This will add a button to your UI to bring up a window with plot options. The logic section is just an example. It allows you some control over the plot options plugin. Read more in the

plot_options plugin's help page (linked from the help page of any plugin providing the generic options).

Next you need to make sure that the code corresponding to your plot options is added to the generated code for your plot. To do so, fetch the properties **code.preprocess**, **code.printout**, and **code.calculate** from the embedded plot options plugin, and insert them into your code as shown in the example, below.

10.1.4 A canonical example

Here's an example .JS file that you should use as a template, whenever you create a plotting plugin:

```
function preprocess () {
  // the "somepackage" is needed to create the plot
  echo ("require (somepackage)\n");
}

function preview () {
  // we call all stages of the general code. Only the printout ()  ↩
      function needs to be called slightly different for the plot preview
  preprocess ();
  // calculate (); // in this example, the plugin has no calculate ()  ↩
      function.
  printout (true); // in this case, 'true' means: Create the plot, but  ↩
      not any headers or other output.
}

function printout (is_preview) {
  // If "is_preview" is set to false, it generates the full code,  ↩
      including headers.
  // If "is_preview" is set to true, only the essentials will be  ↩
      generated.

  if (!is_preview) {
    echo ('rk.header (' + i18n ("An example plot") + ')\n\n');
    echo ('rk.graph.on ()\n');
  }
  // only the following section will be generated for is_preview==true

  // remember: everything between rk.graph.on() and rk.graph.off() should ↩
       be wrapped inside a try() statement:
  echo ('try ({\n');
  // insert any option-setting code that should be run before the actual  ↩
      plotting commands.
  // The code itself is provided by the embedded plot options plugin.  ↩
      printIndentedUnlessEmpty() takes care of pretty formatting.
  printIndentedUnlessEmpty ('\t', getString ("plotoptions.code.preprocess ↩
      "), '', '\n');

  // create the actual plot. plotoptions.code.printout provides the part  ↩
      of the generic plot options
  // that have to be added to the plotting call, itself.
  echo ('plot (5, 5' + getString ("plotoptions.code.printout") + ')\n');

  // insert any option-setting code that should be run after the actual  ↩
      plot.
```

```
  printIndentedUnlessEmpty ('\t', getString ("plotoptions.code.calculate ↩
      "), '\n');
  echo ('})'\n);  // the closure of the try() statement

  if (!is_preview) {
    echo ('rk.graph.off ()\n');
  }
}
```

10.2 Previews for data, output and other results

10.2.1 Previews of (HTML) output

> NOTE
> This section discusses adding preview functionality to plugins creating output / HTML printouts. It is recommended that you read the separate section on plot previews, before this section.

Creating a preview of HTML output is almost the same procedure as creating a plot preview. In this case, simply make sure that the **preview()** generates the relevant **rk.print()/rk.results()** commands. It is generally a good idea to omit the header statements in the preview, however. Here is a stripped-down example:

```
<!-- In the plugin's XML file -->>
<dialog label="Import CSV data" >
  <browser id="file" type="file" label="File name"/>
  <!-- [...] -->>
  <preview id="preview" mode="output"/>
</dialog>>
```

Note the specification of *mode=´´output´´* in the **<preview>** element.

```
// In the plugin's JS file
function preview () {
  // generates the code used for preview
  printout (true);
}

function printout (is_preview) {
  // only generates a header if is_preview==false
  if (!is_preview) {
    new Header ("This is a caption").print ();
  }
  echo ('rk.print (result)');
}
```

10.2.2 Previews of (imported) data

> NOTE
> This section discusses adding preview functionality to plugins creating (importing) data. It is recommended that you read the separate section on plot previews, before this section.

Creating a preview of imported data (any type of data that **rk.edit()** can handle), is very similar to creating a plot preview. The following stripped down example should help illustrate how to create a data preview:

```
<!-- In the plugin's XML file -->>
<dialog label="Import CSV data" >
  <browser id="file" type="file" label="File name"/>
  <!-- [...] -->>
  <preview id="preview" active="true" mode="data"/>
</dialog>>
```

Note that the **<preview>** element specifies *mode=´´data´´* this time. *active=´´true´´* simply makes the preview active by default.

```
// In the plugin's JS file
function preview () {
  // generates the code used for preview
  calculate (true);
}

function calculate (is_preview) {
  echo ('imported <- read.csv (file="' + getString ("file") /* [+ options ↩
      ] */);
  if (is_preview) {
    echo ('preview_data <- imported\n');
  } else {
    echo ('.GlobalEnv$' + getString ("name") + ' >- imported\n');
  }
}

function printout () {
  // [...]
}
```

Again, the **preview()** function generates almost the same R code as the **calculate()** function, so we create a helper function **doCalcuate()** to factor out the common parts. The most important thing to note is that you will have to assign the imported data to a object called *preview_data* (inside the current - local - environment). *Everything else will happen automatically* (roughly speaking, RKWard will call **rk.edit(preview_data)**, wrapped inside a call to **.rk.with.window.hints()**).

> NOTE
> While previews are a great feature, they do consume resources. In the case of data previews, there may be cases, where previews can cause significant performance issues. This could be for importing huge datasets (which are just too large to be opened for editing in RKWard's editor window), but also ˝normal˝ datasets could be mis-imported, creating a huge number of rows or columns. *It is very much recommended that you limit the preview_data* to a dimension that provides a useful preview, without the danger of creating noticable performance issues (e.g. 50 rows by 50 columns should be more than enough in most cases).

10.2.3 Custom previews

The **<preview>** element can be used to create previews for any type of ˝document˝ window that can be attached to RKWard's workplace. In addition to plots and data windows, this includes HTML files, R scripts, and object summary windows. For the latter ones, you will have to use **<preview mode=˝custom˝>**.

If you have read the sections describing plot preview and data previews, you should have a general idea on the procedure, but ˝custom˝ previews require slightly more manual work behind the scenes. The most important R function to look at is **rk.assign.preview.data()**, here. The following short listing shows what your generated (preview) R code could look like for a plugin creating a text file output:

```
## To be generated in the preview() code section of a plugin
pdata <- rk.get.preview.data("SOMEID")
if (is.null (pdata)) {
  outfile <- rk.get.tempfile.name(prefix="preview", extension=".txt")
  pdata <- list(filename=outfile, on.delete=function (id) {
    unlink(rk.get.preview.data(id)$filename)
  })
  rk.assign.preview.data("SOMEID", pdata)
}
try ({
  cat ("This is a test", pdata$filename)
  rk.edit.files(file=pdata$filename)
})
```

Here you should get the value *SOMEID* from the *id* property of the **<preview>**-element. I.e. using **getString (˝preview.id˝)** in the plugin's .js file.

10.3 Context-dependent plugins

So far we have assumed, all plugins are always meaningful, and all placed in the main menu. However, some plugins are meaningful only (or additionally) in a certain context. For instance a plugin to export the contents of an R X11 graphics device is obviously most useful, when placed in the menu of an X11 device, not in the main menubar. Also, such a plugin should know about the device number that it should operate on, without having to ask the user about this.

We call such plugins context-dependent. Correspondingly, in the `.pluginmap` file, they are not (or not only) placed in the main **<hierarchy>** but rather into a **<context>** element. So far only two different contexts are supported (more will come later): x11 and file import. We'll deal with those in turn. Even if you are only interested in the import context, please also read the section on the x11 context, as this is slightly more elaborate.

10.3.1 X11 device context

To use a plugin in the context of an x11 device - that is place it in the menubar of the window you get when you call `x11()` in the console, first declare it as usual in the `.pluginmap` file:

```
<document [...]>
  <components>
    [...]
    <component id="my_x11_plugin" file="my_x11_plugin.xml" label="An X11 ↩
        context plugin"/>
    [...]
  </components>
```

However, you do not need to define it in the hierarchy (you can, if it is also meaningful as a top-level plugin):

```
  <hierarchy>
    [...]
  </hierarchy>
```

Instead, add a definition of the ˝x11˝ context, and add it to the menus there:

```
  <context id="x11">
    [...]
    <menu id="edit">
      [...]
      <entry id="my_x11_plugin"/>
    </menu>
  </context>
</document>
```

In the logic section of the plugin xml, you can now declare two **<external>** properties: *devnum* and *context*. *context* (if declared) will be set to *´´x11´´* when the plugin is invoked in this context. *devnum* will be set to the number of the graphics device to operate on. And that's all.

10.3.2 Import data context

Before reading this section, please make sure to read the section on the X11 device context, as that explains the basic concepts.

The *´´import´´* context is used to declare import file filter plugins. You simply place those in a context with *id=´´import´´* in the `.pluginmap` file. However, there is one additional twist when declaring these plugins: In order to offer a unified file selection dialog for all supported file types, you need to declare one additional bit of information on your component:

```
<document [...]>
  <components>
    [...]
    <component id="my_xyz_import_plugin" file="my_xyz_import_plugin.xml"  ↩
       label="Import XYZ files">
      <attribute id="format" value="*.xyz *.zyx" label="XYZ data files"/>
    </component>
    [...]
  </components>
  <hierarchy>
    [...]
  </hierarchy>
  <context id="import">
    [...]
    <menu id="import">
      [...]
      <entry id="my_xyz_import_plugin"/>
    </menu>
  </context>
  [...]
</document>
```

The attribute line simply says, that the associate filename extensions for XYZ files are `*.xyz` or `*.zyx`, and that the filter should be labeled 'XYZ data files' in the file selection dialog.

You can declare two **<external>** properties in your plugin. *filename* will be set to the selected file name, and *context* will be set to *´´import´´*.

10.4 Querying R for information

In some cases, you may want to fetch further information from R, to be presented in your plugin's UI. For instance, you may want to offer a selection of the levels of a factor that the user has

selected for analysis. Since version 0.6.2 of RKWard it is possible to do so. Before we start, it is important that you are aware of some caveats:

R Code run from inside the plugin's UI logic is evaluated in R's event loop, meaning they can be run *while* other computations are running. This is to make sure your plugin's UI will be usable, even while R is busy doing other things. However, this makes it really important, that your code does not have side effects. In particular:

- Do *not* make any assignments in .GlobalEnv or any other non-local environment.
- Do *not* print anything to the output file.
- Do *not* plot anything on-screen.
- In general, do *not* do anything that has side-effects. Your code may *read in information*, not ˝do˝ anything.

With this in mind, here's the general pattern. You will use this inside a scripted UI logic section:

```
<script><![CDATA[
    last_command_id = -1;
    gui.addChangeCommand ("variable", "update ()");
    update = function () {
      gui.setValue ("selector.enabled", 0);
      variable = gui.getValue ("variable");
      if (variable == "") return;

      last_command_id = doRCommand ('levels (' + variable + ')', " ↩
          commandFinished");
    }

    commandFinished = function (result, id) {
      if (id != last_command_id) return;  // another result is about to ↩
           arrive
      if (typeof (result) == "undefined") {
        gui.setListValue ("selector.available", Array ("ERROR"));
        return;
      }
      gui.setValue ("selector.enabled", 1);
      gui.setListValue ("selector.available", result);
    }
]]></script>
```

Here, *variable* is a property holding an object name (e.g. inside a **<varslot>**). Whenever that changes, you will want to update the display of levels inside the **<valueselector>**, named *selector*. The key function here is **doRCommand()**, taking as first parameter the command string to run, and as second parameter the name of a function to call, when the command has finished. Note that the command is running asynchronously, and this makes things a bit more complex. For one thing you want to make sure, that your **<valueselector>** remains disabled, while it does not contain up-to-date information. Another thing is that you could potentially have queued more than one command, before you get the first results. This is why every command is given an ˝id˝, and we store that in *last_command_id* for later reference.

When the command is done, the specified callback is called (*commandFinished*, in this case) with two parameters: The result itself, and the id of the corresponding command. The result will be of a type resembling the representation in R, i.e. a numeric Array, if the result is numeric, etc. It can even be an R **list()**, but in this case it will be represented as a JS **Array()** without names.

Note that even this example is somewhat simplified. In reality you should take additional precautions, e.g. to avoid putting an extreme amount of levels into the selector. The good news is that probably you do not have to do all this yourself. The above example is taken from the **rkward::level_select** plugin, for instance, which you can simply embed in your own plugin. This even allows you to specify a different expression to run in place of **levels()**.

10.5 Referencing the current object

For many plugins it is desirable to work on the 'current' object. For instance a 'sort' plugin could pre-select the data.frame that is currently being edited for sorting. The name of the current object is available to plugins as a pre-defined property called *current_object*. You can connect to this property in the usual way. If no object is current, the property equates to an empty string.

Currently the *current_object* can only be of class `data.frame`, but please do not rely on this, since this will be extended to other types of data in the future. If you are interested in `data.frame` objects, only, connect to the *current_dataframe* property, instead. Alternatively, you can enforce type requirements by using appropriate constraints on your **<varslot>**s, or by using GUI logic scripting.

10.6 Repeating (a set of) options

Sometimes you want to repeat a set of options for an arbitrary number of items. E.g. suppose you want to implement a plugin for sorting a data.frame. You may want to allow for sorting by an arbitrary number of columns (in case of ties among the first column(s)). This could simply be realized by allowing the user to select multiple variables in a **<varslot>** with *multi=''true''*. But if you want to extend this, e.g. allowing the user to specify for each variable whether it should be converted to character / numeric, or whether sorting should be ascending or descending, you need more flexibility. Other examples would be plotting multiple lines in one plot (allowing to select object, line style, line color, etc. for each line), or specifying a mapping for recoding from a set of old values to new values.

Enter the **<optionset>**. Let's look at a simple example, first:

```
<dialog [...]>
  [...]
  <optionset id="set" min_rows="1">
    <content>
      <row>
        <input id="firstname" label="Given name(s)" size="small">
        <input id="lastname" label="Family name" size="small">
        <radio id="gender" label="Gender">
          <optioncolumn label="Male" value="m"/>
          <optioncolumn label="Female" value="f"/>
        </radio>
      </row>
    </content>

    <optioncolumn id="firstnames" label="Given name(s)" connect="firstname. ↩
        text">
    <optioncolumn id="lastnames" label="Family name" connect="lastname.text ↩
        ">
    <optioncolumn id="gender" connect="gender.string">
  </optionset>
  [...]
</dialog>
```

Here, we created a UI for specifying a number of persons (e.g. authors). The UI requires at least one entry (*min_rows=''1''*). Inside the **<optionset>**-element, we begin by specifying the **<content>**, i.e. those elements that belong to the option set. You will be familiar with most elements inside the **<content>**.

Next we specify the variables of interest that we will want to read from the option set in our JS file. As we will be dealing with an arbitrary number of items, we cannot just read `getString`

`(´´firstname´´)` in JS. Rather, for each value of interest, we specify an **<optioncolumn>**. For the first optioncolumn in the example, **<connect=˝firstname.text˝>** means that the content of the **<input>** element ˝firstname˝ is read for each item. **<optioncolumn>**s for which a *label* is given, will be shown in the display, in a column by that label. In JS, we can now fetch the first names for all authors using `getList(´´set.firstname´´)`, `getList(´´set.lastnames´´)` for the family names, and `getList(´´set.gender´´)` for an array of ˝m˝/˝f˝ strings.

Note that there are no restrictions on what you can place inside an **<optionset>**. You can even use embedded components. Just as with any other element, all you have to do is to collect the output variables of interest in an **<optioncolumn>**-specification. In the case of embedded plugins, this is often a section of the ˝code˝ property. E.g.:

```
<dialog [...]>
  [...]
  <optionset id="set" min_rows="1">
    <content>
      [...]
      <embed id="color" component="rkward::color_chooser" label="Color"/>
    </content>

    [...]
    <optioncolumn id="color_params" connect="color.code.printout">
  </optionset>
  [...]
</dialog>
```

Of course you can also use UI logic inside an optionset. There are two options for doing this: You can do so by making connection (or scripting) in the main **<logic>** section of your plugin, as usual. However, you will access the UI elements in the contents region as (e.g.) ˝set.contents.firstname.XYZ˝. Note the prefix ˝set˝ (the *id* you have assigned to the set and ˝contents˝). Alternatively, you can add a separate **<logic>** section as a child element of your **<optionset>**. In this case, *id*s will be addressed relative to the contents region, e.g. ˝firstname.XYZ˝. Only the **<script>**-element is not allowed in the logic section of an optionset. If you want to use scripting, you will have to utilize the plugin's main **<logic>** section.

> NOTE
> When scripting logic in an optionset, all you can do is access the *current* content region. Thus, typically, it is only meaningful to connect elements inside the contents region to each other. Connecting a property outside the **<optionset>** to a property inside the content region, may be useful for initialization. However, modifying the contents region after initialization will *not* apply to items that the user has already defined. Only to the currently selected item in the set.

10.6.1 ˝Driven˝ optionsets

So far we have considered an **<optionset>** that provides buttons for adding / removing items. However, in some cases, it is much more natural to select items outside the **<optionset>**, and only provide options for customizing some aspects of each item in an **<optionset>**. E.g. suppose you want to allow the user to plot several objects inside one plot. For each object, the user should be able to specify line color. You *could* solve this by placing a **<varselector>** and **<varslot>** inside the **<content>** area, allowing the user to select one item at a time. However, it will mean much less clicks for the user, if you use a **<varslot multi=˝true˝>** *outside* the **<optionset>**, instead. Then you will connect this selection of objects to a so-called ˝driven˝ optionset. Here's how:

```
<dialog [...]>
  <logic>
    <connect client="set.vars" governor="vars.available"/>
```

```
    <connect client="set.varnames" governor="vars.available.shortname"/>
  </logic>
  [...]
  <varselector id="varsel"/>
  <varslot id="vars" label="Objects to plot"/>
  <optionset id="set" keycolumn="var">
    <content>
      [...]
      <embed id="color" component="rkward::color_chooser" label="Line color ↩
          "/>
    </content>

    [...]
    <optioncolumn id="vars" external="true">
    <optioncolumn id="varnames" external="true" label="Variable">
    <optioncolumn id="color_params" connect="color.code.printout">
  </optionset>
  [...]
</dialog>
```

We'll start looking at the example at the bottom. You'll note that two **<optioncolumn>** specifications have *external=´´true´´*. This tells RKWard that these are controlled from outside the **<optionset>**. Here, the sole purpose of the ˝varnames˝-optioncolumn is to provide easy-to-read labels in the optionset display (it is connected to the ˝shortname˝ modifier of the property holding the selected objects). The purpose of the ˝vars˝-optioncolumn is to serve as the ˝key˝ column, as specified by **<optionset keycolumn=˝vars˝...>**. This means that for each entry in this list, the set will offer one set of options, and options are logically tied to these entries. This column is connected to the property holding the selected objects in the **<varslot>**. That is for each object that is selected there, the **<optionset>** will allow to specify line color.

> NOTE
> External column can also be *connect*ed to properties inside the **<content>** region. However, it is important to note that optioncolumns declared *external=´´true´´* should never be modified from inside the **<optionset>**, and optioncolumns declared *external=´´false´´* (the default) should never be modified from outside the **<optionset>**.

10.6.2 Alternatives: When not to use optionsets

Optionsets are a powerful tool, but they can sometimes do more harm than good, as they add considerable complexity, both from the perspective of a plugin developer, and from the perspective of a user. Thus, think twice, when using them. Here's some advice:

- For some simple cases, the **<matrix>** element may provide a useful lightweight alternative.
- Don't make your plugin do too much. We gave the example of using an optionset for a plugin to draw several lines within one plot. But in general it is not a good idea to create a plugin that will produce individual plots for each item in an optionset. Rather make the plugin produce one plot, and the user can call it multiple times.
- If you don't expect more than two or three items in a set, consider repeating the options, manually, instead.

Chapter 11

Handling dependencies and compatibility issues

11.1 RKWard version compatibility

We do our best to make sure that plugins developed for an old version of RKWard will remain functional in later versions of RKWard. However, the reverse is not always true as new features are been added. Since not all users are running the latest version of RKWard, this means your plugin may not work for everybody.

When you are aware of such compatibility issues, you should make sure to document this fact in your `.pluginmap` file, using the **<dependencies>** element. The **<dependencies>** can either be specified as a direct child of the `.pluginmap`'s <document> element, or as a child element of individual **<component>** definitions. In the first case, the dependencies apply to *all* plugins in the map. In the latter case only to the individual **<component>**(s). You can also mix top ˝global˝ and ˝specific˝ dependencies. In this case the ˝global˝ dependencies are added to those of the individual component.

Let's look at a small example:

```
<document ...>
  <dependencies rkward_min_version="0.5.0c" />
  <components ...>
    <component id="myplugin" file="reduced_version_of_myplugin.xml" ...>
      <dependencies rkward_max_version="0.6.0z" />
    </component>
    <component id="myplugin" file="fancy_version_of_myplugin.xml" ...>
      <dependencies rkward_min_version="0.6.1" />
    </component>
    ...
x </components ...>
</document>
```

In this example, all plugins are known to require at least version 0.5.0c of RKWard. One plugin, with *id=´´myplugin´´* is provided in two alternative variants. The first, stripped down, version will be used for RKWard versions before 0.6.1. The latter utilizes features that are new in RKWard 0.6.1, and will only be used from RKWard 0.6.1 onwards.

Providing alternative variants like this is a very user friendly way to make use of new features, while still keeping support for earlier versions of RKWard. Alternative versions should share the same *id* (warnings will be produced, otherwise), and may only be defined *within the same* `.pluginmap` file.

Plugin which are not compatible with the running version of RKWard, and which do not come with an alternative version will be ignored with a warning.

> NOTE
> Actually RKWard 0.6.1 is the first version to interpret dependencies - and to report dependency errors - at all. Thus, contrary to what the example may suggest, specifying any earlier versions in the dependencies will not have any direct effect (but may still be a good idea for documentation purposes).

Sometimes it will even be possible to handle version incompatibility issues *inside* a single `.plugin map` file, using the **<dependency_check>** element, described in the following section.

11.2 R version compatibility

Similar to *rkward_min_version* and *rkward_max_version*, the **<dependencies>** element allows specification of the attributes *R_min_version* and *R_max_version*. However, there are the following differences:

- Plugins which fail to meet the R version requirement are *not* currently skipped when reading a `.pluginmap` file. The user can still call the plugin, and will not see any immediate warning (in future versions, a warning message will probably be shown)
- In consequence it is also *not* possible to define alternative versions of a plugin depending on the running version of R.
- However, it is often easy to achieve backwards compatibility as shown below. If you are aware of R compatibility issues, please consider using this method, instead of defining a dependency on a particular version of R.

In many cases, it is easily possible to provide reduced functionality, if a certain feature is not available in the running version of R. Consider the following short example of a plugin .xml file:

```
<dialog [...]>
  <logic>
    <dependency_check id="ris210" R_min_version="2.10.0"/>
    <connect client="compression.xz.enabled" governor="ris210"/>
  </logic>
  [...]
  <radio id="compression" label="Compression method">
    <option label="None" value="">
    <option label="gzip" value="gzip">
    <option id="xz" label="xz" value="xz">
  </radio>
  [...]
</dialog>
```

In this example the compression option ˝xz˝ will simply be disabled when the R runtime version is older than 2.10.0 (which did not support xz compression). The **<dependency_check>** element supports the same attributes as the **<dependencies>** element in `.pluginmap` files. It creates a boolean property, which is true, if the specified dependencies are met, false otherwise.

11.3 Dependencies on R packages

Dependencies on specific R packages can be defined, but as of RKWard 0.6.1, these dependencies are neither checked, nor installed / loaded, automatically. They are shown in the plugin help files, however. Here is an example definition:

```
<dependencies>
  <package
    name="heisenberg"
    min_version="0.11-2"
    repository="http://rforge.r-project.org"
  />
</dependencies>
```

> NOTE
> Always make sure to add appropriate `require()` calls, if you plugin needs certain packages to be loaded.

> NOTE
> If you distribute your `.pluginmap` as an R package, and all plugins depend on a particular package, then you should define that dependency on the R package level. Defining dependencies to R packages on the level of the RKWard `.pluginmap` is most useful, if only some of your plugins need the dependency, the dependency is not available from CRAN, or your `.pluginmap` is not distributed as an R package.

11.4 Dependencies on other RKWard `.pluginmaps`

If your plugins depend on plugins defined in another `.pluginmap` (that is *not* part of your package) you can define this dependency like this:

```
<dependencies>
  <pluginmap
    name="heisenberg_plugins"
    url="http://eternalwondermaths.example.org/hsb"
  />
</dependencies>
```

Currently will neither load, nor install, nor even warn about missing `.pluginmaps`, but at least information on dependencies (and where to obtain them) will be shown on the plugin's help page. You do not have to (and you should not) declare dependencies on `.pluginmaps` that are shipped with the official RKWard distribution, or on `.pluginmaps` that are inside your own package. Further, if a required `.pluginmap` is distributed as an R package, declare a dependency of the package (as shown in the previous section), rather than on the map.

To make sure that required `.pluginmaps` are actually loaded, use the **<require>**-tag (refer to the reference for details).

11.5 An example

To clarify how dependency definitions can be mixed, here's a combined example:

```
<document ...>
  <dependencies rkward_min_version="0.5.0c">
    <package
      name="heisenberg"
```

```
      min_version="0.11-2"
      repository="http://rforge.r-project.org"
    />
    <package
      name="DreamsOfPi"
      min_version="0.2"
    />
    <pluginmap
      name="heisenberg_plugins"
      url="http://eternalwondermaths.example.org/hsb"
    />
  <dependencies>

  <require map="heisenberg::heisenberg_plugins"/>

  <components ...>
    <component id="myplugin" file="reduced_version_of_myplugin.xml" ...>
      <dependencies rkward_max_version="0.6.0z" />
    </component>
    <component id="myplugin" file="fancy_version_of_myplugin.xml" ...>
      <dependencies rkward_min_version="0.6.1" />
    </component>
    ...
x </components ...>
</document>
```

Chapter 12

Plugin translations

So far we have used a few concepts regarding translations or ˝i18n˝ (short for ˝internationalization˝, which has 18 characters between i and n) in passing. In this chapter we give a more in-depth account of what i18n functionally for RKWard plugins. For the most part you will *not* need all of this in your plugins. However, it may be a good idea to read over this chapter in full, as understanding these concepts should help you creating plugins that are fully translatable, and that allow for a high quality of translations.

12.1 General considerations

One important point to understand about software translations, in contrast to translations of other text materials, is that translators will often have a rather hard time getting a complete picture of *what* they are translating. Software translations are necessarily based on rather short fragments of text: Every label you give to an **<option>** in a **<radio>**, every string that you mark for translation in an **i18n()**-function call, will form a separate ˝translation unit˝. In essence, each such fragment will be presented to the translator in isolation. Well, not complete isolation, as we do try to provide translator with as much meaningful context as can be extracted, automatically. But at some points translators will need additional context to make sense of a string, especially where strings are short.

12.2 i18n in RKWard's xml files

For RKWard's xml files, i18n will mostly just work. If you are writing your own **.pluginmap** (e.g. for an external plugin), you will have to specify a `po_id` next to the pluginmap's `id`. This defines the ˝message catalog˝ to use. In general this should be set identical to the `id` of your **.pluginmap**, but if you provide several **.pluginmap**s and want to control, how message catalogs are divided up, this allows you to do so. The `po_id` is inherited by any **.pluginmap** you include, unless that declares a different `po_id`, and by all plugins declared in it.

For plugins and help pages, you do not need to tell RKWard which strings are to be translated, because that is generally evident from their usage. However, as explained above, you should keep an eye out for strings that may be ambiguous or need some explaining in order to be translated, correctly. For strings that could have different meanings, provide an `i18n_context` like this:

```
<checkbox id="scale" label="Scale" i18n_context="Show the scale"/>
<checkbox id="scale" label="Scale" i18n_context="Scale the plot"/>
```

Providing *i18n_context* will cause the two strings to be translated separately. Otherwise they would share a single translation. In addition, the context is shown to the translator. The *i18n_context*-attribute is supported on all elements that can have translatable strings, somewhere, including elements that contain text inside them (e.g. **<text>**-elements).

In other cases the string to translate has a single non-ambiguous meaning, but may still need some explaining. In this case you can add a comment that will be shown to translators. Examples might include:

```
<!-- i18n: No, this is not a typo for screen plot! -->
<component id="scree_plot" label="Scree plot"/>

<!-- i18n: If you can, please make this string short. Having more than some ↩
     15 chars
looks really ugly at this point, and the meaning should be mostly self- ↩
    evident to the
user (selection from a list of values shown next to this element) -->
<valueslot id="selected" label="Pick one"/>
```

Note that such comments must precede the element they apply to, and must start with either ˝i18n:˝ or ˝TRANSLATORS:˝.

Finally, in rare cases, you may want to exclude certain strings from translation. This may make sense, for example, if you offer a choice between several R function names in a **<radio>**-control. You do not want these to be translated, then (but depending on the context, you should consider giving a descriptive label, instead):

```
<radio id="transformation" label="R function to apply">
  <option id="as.list" noi18n_label="as.list()"/>
  <option id="as.vector" noi18n_label="as.vector()"/>
  [...]
</radio>
```

Note that you will omit the *label*-attribute, then, and specify *noi18n_label*, instead. Also, note that in contrast to *i18n_context* and comments, using *noi18n_label* will make your plugin incompatible with versions of RKWard prior to 0.6.3.

12.3 i18n in RKWard's js files and sections

In contrast to the .xml files, making the js files of a plugin translatable requires more custom work. The key difference, here, is that there is no decent automatic way to tell, whether a string is meant to be displayed as a human readable string, or a piece of code. So you need to mark this up, yourself. We have already shown examples of this, all along. Here is a more complete description of the i18n-functions available in js code, and some hints for more complex cases:

i18n (msgid, [...])

The most important function. Marks the string for translation. The string (whether translated or not) is returned quoted using double quotes (˝˝). An arbitrary number of placeholders can be used in the string like shown below. Using such placeholders instead of concatenating small substrings is much easier for translators:

```
i18n ("Compare objects %1 and %2", getString ('x'), getString ('y ↩
    '));
```

i18nc (msgctxt, msgid, [...])

Same as **i18n()**, but additionally providing a message context:

```
i18nc ("proper name, not state of mind", "Mood test");
```

i18np (msgid_singular, msgid_plural, n, [...])

Same as **i18n()**, but for messages that may be different in singular or plural form (and some languages have differentiate yet more numerical forms). Note that just like with **i18n()**, you can use an arbitrary number of replacements, but the first ('%1') is required, and has to be an integer.

```
i18np ("Comparing a single pair", "Comparing %1 distinct pairs",  ↩
    n_pairs);
```

i18ncp (msgctxt, msgid_singular, msgid_plural, n, [...])

i18np() with added message context.

comment (comment, [indentation])

Echos a code comment, marked for translation. In contrast to the other i18n() functions, this is not quoted, but a '#' is added to each line of the comment.

```
comment ("Transpose the matrix");
echo ('x <- t (x)\n');
```

To add comments to the translators (see above for a discussion of the differences between comment and context), add a comment starting with ˝i18n:˝ or ˝translators:˝ directly above the **i18n()**-call to comment. E.g.:

```
// i18n: Spelling is correct: Scree plot.
echo ('rk.header (' + i18n ("Scree plot") + ')\n');
```

12.3.1 i18n and quotes

For the most part, you will not have to worry about i18n() behavior with respect to quotes. As, typically, translatable strings are string literals, quoting them is just the right thing to do, and saves you some typing. Also, in functions like **makeHeaderCode()/Header()** that usually quote their arguments, i18n()'ed strings are protected from duplicate quoting. Essentially, this works, by sending the translated string first through **quote()** (to make it quoted), then through **noquote()** (to protect it from additional quoting). Should you require a translatable string that is not quoted, use **i18n(noquote (˝My message˝))**. Should you require a translatable string to be quoted, a second time, send it through **quote()**, *twice*.

That said, it is generally not a good idea to make bits like function names or variable names translatable. For one thing, R, the programming language, is inherently in English, and there is no internationalization of the language itself. Code comments are a different beast, but you should use the **comment()**-function for those. Secondly, making syntactically relevant parts of the generated code translatable means that translations could actually break your plugin. E.g. if an unsuspecting translator translates a string meant as a variable name in two distinct words with a space in between.

12.3.2 i18n and backwards compatibility

One unfortunate aspect of the lack of **i18n()**-support in RKWard versions up to 0.6.2 is that adding **i18n()** calls will make the plugin require RKWard version 0.6.3. If your plugin is developed outside RKWard's official release, this may be a problem. Here are some possible options on how to handle this:

- Provide the plugin in two versions for RKWard >= 0.6.3 and RKWard < 0.6.3, as described in the chapter on handling dependencies
- Simply don't translate the strings in the .js-file, yet. Obviously this is an easy, but rather inelegant solution.
- Include some support code in your .js-file(s) like shown below:

```
// js-function "comment" was not defined before 0.6.3
if (typeof (comment) == 'undefined) {
  // define function i18n(), and any others you may need. Note that ↩
     your implementation could actually be simpler than
  // shown, here, e.g. if you do not make use of placeholders.
  i18n = function (msgid) {
    var ret = msgid;
    for (var i = 1; i < arguments.length; i++) {
      ret = ret.replace(new RegExp("%" + i, 'g'), arguments[i]);
    }
    if (msgid.noquote) {
      ret.noquote = msgid.noquote;
      return (ret);
    }
    return (noquote (quote (ret)));
  }
}
```

12.4 Translation maintainance

Now that you have made your plugin translatable, how do you actually get it translated? In general you only need to worry about this, when developing an external plugin. For plugins in RKWard's main repository, all the magic is done for you. Here's the basic workflow. Note that you need the ˝gettext˝ tools, installed:

- Mark up all strings, providing context and comments as needed
- Run **python scripts/update_plugin_messages.py --extract-only /path/to/my.pluginmap**. scripts/update_plugin_messages.py is not currently part of the source releases, but can be found in a source repository checkout.
- Distribute the resulting **rkward__*POID*.pot** file to your translators. For external plugins, it is recommended to place it in a subfolder ˝po˝ in inst/rkward.
- Translator opens the file in a translation tool such as **lokalize**. Actually, even if you are not going to prepare any translation, yourself, you should try this step for yourself. Browse the extracted strings looking out for problems / ambiguities.
- Translator saves the translation as **rkward__*POID*.*xx*.po** (where *xx* is the language code), and sends it back to you.
- Copy **rkward__*POID*.*xx*.po** to your sources, next to **rkward__*POID*.pot**. Run **python scripts/update_plugin_messages.py /path/to/my.pluginmap** (Note: without *--extract-only*, this time). This will merge the translation with any interim string changes, compile the translation, and install it into ***DIR_OF_PLUGINMAP*/po/*xx*/LC_MESSAGES/rkward__*POID*.mo** (where *xx* is the language code, again).
- You should also include the non-compiled translation (i.e. **rkward__*POID*.*xx*.po**) in your distribution, in the ˝po˝ subdirectory.

- For any update of your plugin, run **python scripts/update_plugin_messages.py /path/-to/my.pluginmap** to update the .pot file, but also the existing .po-files, and the compiled message catalogs.

12.5 Writing plugin translations

We assume you know your trade as a translator, or are willing to read up on it, elsewhere. A few words specifically about translations of RKWard plugins, though:

- RKWard plugins were not translatable until version 0.6.3, and were mostly not written with i18n in mind, before then. Thus you are going to encounter rather more ambiguous strings, and other i18n problems than in other mature projects. Please don't just silently work around these, but let us (or the plugin maintainers) know, so we can fix these issues.
- Many RKWard plugins refer to highly specialized terms, from data handling and statistics, but also from other fields of science. In many cases, a good translation will require at least basic knowledge of these fields. In some cases, there *is* no good translation for a technical term, and the best option may be to leave the term untranslated, or to include the English term in parentheses. Don't focus too much on the 100% mark of translated strings, focus on providing a good translation, even if that means skipping some strings (or even skipping some message catalogs as a whole). Other users may be able to fill in any gaps in technical terms.
- At the time of this writing, RKWard's project hosting is about to change, and this also affect the translation workflow. Do read comments accompanying the .pot-files, on how translations should be handled. If in doubt, it is never wrong to send your work the rkward-devel mailing list, or to ask for up-to-date instructions, there.

Chapter 13

Author, license and version information

So you have written a set of plugins, and you are getting ready to share your work. To make sure users will know what your work is all about, under what terms they can use and distribute it, and whom they should contact about issues or ideas, you should add some information *about* your plugins. This can be done using the **<about>** element. It can be used in either the `.pluginmap` or in individual plugin .xml files (in both cases as a direct child of the document tag). When specified in the `.pluginmap` it will apply to all plugins. If **<about>** is specified in both places, the **<about>** information in the plugin .xml file will override that in the `.pluginmap` file. You can also add an **<about>** element to .rkh-pages, which are not connected to a plugin, if there is a need for that.

Here's an example `.pluginmap` file with only a few explanations, below. In cases of doubt, more information may be available from the reference.

```
<document
  namespace="rkward"
  id="SquaretheCircle_rkward"
>
  <about
    name="Square the Circle"
    shortinfo="Squares the circle using Heisenberg compensation."
    version="0.1-3"
    releasedate="2011-09-19"
    url="http://eternalwondermaths.example.org/23/stc.html"
    license="GPL"
    category="Geometry"
  >
    <author
      given="E.A."
      family="Dölle"
      email="doelle@eternalwondermaths.example.org"
      role="aut"
     />
    <author
      given="A."
      family="Assistant"
      email="alterego@eternalwondermaths.example.org"
      role="cre, ctb"
     />
  </about>
  <dependencies>
```

```
    ...
  </dependencies>
  <components>
    ...
  </components>
  <hierarchy>
    ...
  </hierarchy>
</document>
```

Most of this should explain itself, so we'll not discuss each and every tag element. But let's look at some details that probably need some commentary for easier understanding.

The *category* element in **<about>** can be defined rather freely, but should be meaningful, as it's thought to be used to order plugins into groups. All other attributes in this opening tag are mandatory and must be filled with reasonable content.

At least one **<author>** with a valid email address and the role 'aut' ('author') must also be given. In case your plugin causes problems or someone would like to share its gratitude with you, it should be easy to contact someone who's involved. For further information on other valid roles, like 'ctb' for code contributors or 'cre' for package maintenance, please refer to the R documentation on person().

> NOTE
> Remember that you can use **<include>** and / or **<insert>** to repeat information across several .xml files (e.g. information on an author who was involved with several plugins). More information.

> TIP
> You don't have to write this XML code by hand. If you use the function `rk.plugin.skeleton()` from the rkwarddev package and provide all necessary information via the *about* option, it will automatically create a `.pluginmap` file with a working <about> section for you.

Chapter 14

Share your work with others

14.1 External plugins

As of version 0.5.5, RKWard provides a comfortable way to install additional third party plugins which do not belong to the core package itself. We call these 'external plugins'. They come in form of an R package and can be managed directly via the usual package management features of R and/or RKWard.

This section of the documentation describes how external plugins are to be packaged, so that RKWard can use them. The plugin creation itself is of course identical to the previous sections. That is, you should probably first write a working plugin, and then check back here to learn how to distribute it.

Since external plugins are a relatively young feature, details of this might probably change in future releases. You're welcome to contribute your ideas to improve the process.

> TIP
> These docs explain the details of external plugins so you can learn how they work. In addition to that, also have a look at the rkwarddev package, which was designed to automate a lot of the writing process.

14.2 Why external plugins?

The number of packages to extend the functionality of R is immense already, and climbing. On one hand, we want to encourage you to write plugins for even the most specialised tasks you need solved. On the other hand, the average user should not get lost in huge menu trees full of unknown statistical terms. Therefore it seemed reasonable to keep the plugin handling in RKWard quite modular as well. The RKWard team maintains its own public package repository at http://files.kde.org/rkward/R, designated to host your external plugins.

As a rule of thumb, plugins that seem to serve a widely used purpose (e.g. t-Tests) should become part of the core package, while those who serve a rather limited group of people with special interests should be provided as an optional package. For you as a plugin author it's best practice to just start with an external plugin.

14.3 Structure of a plugin package

For external plugins to install and work properly, they must follow some structural guidelines regarding their file hierarchy.

14.3.1 File hierarchy

Let's have a look at the prototypic file hierarchy of an elaborate plugin archive. You don't have to include all of these directories and/or files for a plugin to work (read on to learn what's absolutely necessary), consider this a 'best practice' example:

```
plugin_name/
  inst/
    rkward/
      plugins/
        plugin_name.xml
        plugin_name.js
        plugin_name.rkh
        ...
      po/
        ll/
          LC_MESSAGES/
            rkward__plugin_name_rkward.mo
        rkward__plugin_name_rkward.ll.po
        rkward__plugin_name_rkward.pot
      tests/
        testsuite_name/
          RKTestStandards.sometest_name.rkcommands.R
          RKTestStandards.sometest_name.rkout
          ...
        testsuite.R
      plugin_name.pluginmap
      ...
ChangeLog
README
AUTHORS
LICENSE
DESCRIPTION
```

> **NOTE**
> In this example, all occasions of plugin_name, testsuite_name and sometest_name are to be replaced with their correct names, accordingly. Also, ll is a placeholder for a language abbreviation (e.g., 'de', 'en' or 'es').

> **TIP**
> You don't have to create this file hierarchy by hand. If you use the function `rk.plugin.skeleton()` from the rkwarddev package, it will automatically create all necessary files and directories for you, except the `po` directory which is created and managed by the translation script.

14.3.1.1 Basic plugin components

It is mandatory to include at least three files: a `.pluginmap`, a plugin .xml description and a plugin .js file. That is, even the ˝plugins˝ directory is optional. It might just help to give your files some order, especially if you include more that one plugin/dialog in the archive, which is of course no problem. You can have as many directories for the actual plugin files as you see fit, they just have to resemble the `.pluginmap`, respectively. It is also possible to even include several `.pluginmap` files, if it fits your needs, but you should include them all in 'plugin_name.pluginmap' then.

Each R package must have a valid `DESCRIPTION` file, which is also crucial for RKWard recognizing it as a plugin provider. Most of the information it carries is also needed in the plugin Meta-information and possibly dependencies, but in a different format (the R documentation explains the DESCRIPTION file in detail).

In addition to the general contents of a `DESCRIPTION` file, make sure to also include the line 'Enhances: rkward'. This will cause RKWard to automatically scan the package for plugins if it is installed. An example `DESCRIPTION` file looks like this:

```
Package: SquaretheCircle
Type: Package
Title: Square the circle
Version: 0.1-3
Date: 2011-09-19
Author: E.A. Dölle <doelle@eternalwondermaths.example.org>
Maintainer: A. Assistant <alterego@eternalwondermaths.example.org>
Enhances: rkward
Description: Squares the circle using Heisenberg compensation.
License: GPL
LazyLoad: yes
URL: http://eternalwondermaths.example.org/23/stc.html
Authors@R: c(person(given="E.A.", family="Dölle", role="aut",
    email="doelle@eternalwondermaths.example.org"),
    person(given="A.", family="Assistant", role=c("cre",
    "ctb"), email="alterego@eternalwondermaths.example.org"))
```

> **TIP**
> You don't have to write this file by hand. If you use the function `rk.plugin.skeleton()` from the rkwarddev package and provide all necessary information via the 'about' option, it will automatically create a working `DESCRIPTION` file for you.

14.3.1.2 Additional information (optional)

`ChangeLog`, `README`, `AUTHORS`, `LICENSE` should be self-explaining and are entirely optional. Actually, they won't be interpreted by RKWard, so they are rather meant to carry additional information that might be relevant e.g. for distributors. Most of their relevant content (author credits, licence terms etc.) will be included in the actual plugin files anyway, though (see the section on meta-information). Note that all of these files could also be placed somewhere in the ˝inst˝ directory, if you want them not only to be present in the source archive but the installed package as well.

14.3.1.3 Automated plugin testing (optional)

Another optional directory is ˝tests˝, which is meant to provide files needed for automated plugin testing. These tests are helpful to quickly check if your plugins still work with new versions of R or RKWard. If you want to include tests, you should really restrain yourself to the naming scheme and hierarchy shown here. That is, tests should reside in a directory called `tests`, which includes a file `testsuite.R` and a folder with tests standards named after the appropriate test suite. You can, however, provide more than one test suite; in that case, if you don't want to append them all in the one `testsuite.R` file, you can split them in e.g. one file for each test suite and create one `testsuite.R` with `source()` calls for each suite file. In either case, create separate subdirectories with test standards for each defined suite.

The benefits of upholding to this structure is that plugin tests can be run simply by calling `rktests.makplugintests()` from the rkwardtests package without additional arguments. Have a look at the online documentation on Automated Plugin Testing for further details.

14.4 Building the plugin package

As explained earlier, external RKWard plugins are in effect R packages, and therefore the packaging process is identical. In contrast to ˝real˝ R packages, a pure plugin package doesn't carry any further R code (although you can of course add RKWard plugins to usual R packages as well, using the same methods explained here). This should make it even easier to create a functioning package, as long as you have a valid `DESCRIPTION` file and adhere to the file hierarchy explained in previous sections.

The easiest way to actually build and test your plugin is to use the R command on the command line, for example:

```
R CMD build SquaretheCircle
R CMD INSTALL SquaretheCircle_0.1-3.tar.gz
```

> TIP
> You don't have to build the package like this on the command line. If you use the function `rk.build.package()` from the rkwarddev package, it will build and/or check your plugin package for you.

Chapter 15

Plugin development with the rkwarddev package

15.1 Overview

Writing external plugins involves writing files in three languages (XML, JavaScript and R) and the creation of a standardized hierarchy of directories. To make this a lot easier for willing plugin developers, we're providing the rkwarddev package. It provides a number of simple R functions to create the XML code for all dialog elements like tabbooks, checkboxes, dropdownlists or file-browsers, as well as functions to create JavaScript code and RKWard help files to start with. The function `rk.plugin.skeleton()` creates the expected directory tree and all necessary files where they are supposed to be.

This package is not installed by default, but has to be installed manually from RKWard's own repository. You can either do that by using the GUI interface (**Settings** → **Configure packages**), or from any running R session:

```
install.packages("rkwarddev", repos="http://files.kde.org/rkward/R")
library(rkwarddev)
```

rkwarddev depends on another small package called 'XiMpLe', which is a very simple XML parser and generator and also present in the same repository.

The full documentation in PDF format can also be found there. A more detailed introduction to working with the package can be found in the rkwarddev vignette.

15.2 Practical example

To get you an idea how 'scripting a plugin' looks like, compared to the direct approach you have seen in the previous chapters, we'll create the full t-test plugin once again -- this time only with the R functions of the rkwarddev package.

> TIP
> The package will add a new GUI dialog to RKWard under **File** → **Export** → **Create RKWard plugin script**. Like the name suggests, you can create plugin skeletons for further editing with it. This dialog itself was in turn generated by an rkwarddev script which you can find in the 'demo' directory of the installed package and package sources, as an additional example. You can also run it by calling `demo(´´skeleton_dialog´´)`

15.2.1 GUI description

You will immediately notice that the workflow is considerably different: Contrary to writing the XML code directly, you do not begin with the **<document>** definition, but directly with the plugin elements you'd like to have in the dialog. You can assign each interface element -- be it check boxes, dropdown menus, variable slots or anything else -- to individual R objects, and then combine these objects to the actual GUI. The package has functions for each XML tag that can be used to define the plugin GUI, and most of them even have the same name, only with the prefix `rk.XML.*`. For example, defining a **<varselector>** and two **<varslot>** elements for the ´´*x*´´ and ´´*y*´´ variable of the t-test example can be done by:

```
variables <- rk.XML.varselector(id.name="vars")
var.x <- rk.XML.varslot("compare", source=variables, types="number",  ↩
    required=TRUE, id.name="x")
var.y <- rk.XML.varslot("against", source=variables, types="number",  ↩
    required=TRUE, id.name="y")
```

The most interesting detail is probably *source=variables*: A prominent feature of the package is that all functions can generate automatic IDs, so you don't have to bother with either thinking of *id* values or remembering them to refer to a specific plugin element. You can simply give the R objects as reference, as all functions who need an ID from some other element can also read it from these objects. `rk.XML.varselector()` is a little special, as it usually has no specific content to make an ID from (it can, but only if you specify a label), so we have to set an ID name. But `rk.XML.varslot()` wouldn't need the *id.name* arguments here, so this would suffice:

```
variables <- rk.XML.varselector(id.name="vars")
var.x <- rk.XML.varslot("compare", source=variables, types="number",  ↩
    required=TRUE)
var.y <- rk.XML.varslot("against", source=variables, types="number",  ↩
    required=TRUE)
```

In order to recreate the example code to the point, you'd have to set all ID values manually. But since the package shall make our lives easier, from now on we will no longer care about that.

> TIP
> rkwarddev is capable of a lot of automation to help you build your plugins. However, it might be preferable to not use it to its full extend. If your goal is to produce code that is not only working but can also be easily read and compared to your generator script by a human being, you should consider to always set useful IDs with *id.name*. Naming your R objects identical to these IDs will also help in getting script code that is easy to understand.

If you want to see how the XML code of the defined element looks like if you exported it to a file, you can just call the object by its name. So, if you now called 'var.x' in your R session, you should see something like this:

```
<varslot id="vrsl_compare" label="compare" source="vars" types="number"  ↩
    required="true" />
```

Some tags are only useful in the context of others. Therefore, for instance, you won't find a function for the **<option>** tag. Instead, both radio buttons and dropdown lists are defined including their options as a named list, where the names represent the labels to be shown in the dialog, and their value is a named vector which can have two entries, *val* for the value of an option and the boolean *chk* to specify if this option is checked by default.

```
test.hypothesis <- rk.XML.radio("using test hypothesis",
  options=list(
    "Two-sided"=c(val="two.sided"),
```

```
    "First is greater"=c(val="greater"),
    "Second is greater"=c(val="less")
  )
)
```

The result looks like this:

```
<radio id="rad_usngtsth" label="using test hypothesis">
  <option label="Two-sided" value="two.sided" />
  <option label="First is greater" value="greater" />
  <option label="Second is greater" value="less" />
</radio>
```

All that's missing from the elements of the 'Basic settings' tab is the check box for paired samples, and the structuring of all of these elements in rows and columns:

```
check.paired <- rk.XML.cbox("Paired sample", value="1", un.value="0")
basic.settings <- rk.XML.row(variables, rk.XML.col(var.x, var.y, test. ↩
    hypothesis, check.paired))
```

`rk.XML.cbox()` is a rare exception where the function name does not contain the full tag name, to save some typing for this often used element. This is what `basic.settings` now contains:

```
<row id="row_vTFSPP10TF">
  <varselector id="vars" />
  <column id="clm_vrsTFSPP10">
    <varslot id="vrsl_compare" label="compare" source="vars" types="number" ↩
        required="true" />
    <varslot id="vrsl_against" label="against" i18n_context="compare  ↩
        against" source="vars" types="number" required="true" />
    <radio id="rad_usngtsth" label="using test hypothesis">
      <option label="Two-sided" value="two.sided" />
      <option label="First is greater" value="greater" />
      <option label="Second is greater" value="less" />
    </radio>
    <checkbox id="chc_Pardsmpl" label="Paired sample" value="1"  ↩
        value_unchecked="0" />
  </column>
</row>
```

In a similar manner, the next lines will create R objects for the elements of the 'Options' tab, introducing functions for spinboxes, frames and stretch:

```
check.eqvar <- rk.XML.cbox("assume equal variances", value="1", un.value ↩
    ="0")
conf.level <- rk.XML.spinbox("confidence level", min=0, max=1, initial ↩
    =0.95)
check.conf <- rk.XML.cbox("print confidence interval", val="1", chk=TRUE)
conf.frame <- rk.XML.frame(conf.level, check.conf, rk.XML.stretch(), label ↩
    ="Confidence Interval")
```

Now all we need to do is to put the objects together in a tabbook, and place that in a dialog section:

```
full.dialog <- rk.XML.dialog(
  label="Two Variable t-Test",
  rk.XML.tabbook(tabs=list("Basic settings"=basic.settings, "Options"=list( ↩
      check.eqvar, conf.frame)))
)
```

We can also create the wizard section with its two pages using the same objects, so their IDs will be extracted for the **<copy>** tags:

```
full.wizard <- rk.XML.wizard(
    label="Two Variable t-Test",
    rk.XML.page(
      rk.XML.text("As a first step, select the two variables you want to compare against
        each other. And specify, which one you theorize to be greater. Select two-sided,
        if your theory does not tell you, which variable is greater."),
      rk.XML.copy(basic.settings)),
    rk.XML.page(
      rk.XML.text("Below are some advanced options. It's generally safe not to assume the
        variables have equal variances. An appropriate correction will be applied then.
        Choosing \"assume equal variances\" may increase test-strength, however."),
      rk.XML.copy(check.eqvar),
      rk.XML.text("Sometimes it's helpful to get an estimate of the confidence interval of
        the difference in means. Below you can specify whether one should be shown, and
        which confidence-level should be applied (95% corresponds to a 5% level of
        significance)."),
      rk.XML.copy(conf.frame)))
```

That's it for the GUI. The global document will be combined in the end by `rk.plugin.skeleton()`.

15.2.2 JavaScript code

Until now, using the rkwarddev package might not seem to have helped so much. This is going to change right now.

First of all, just like we didn't have to care about IDs for elements when defining the GUI layout, we don't have to care about JavaScript variable names in the next step. If you want more control, you can write plain JavaScript code and have it pasted to the generated file. But it's probably much more efficient to do it the rkwarddev way.

Most notably you don't have to define any variable yourself, as `rk.plugin.skeleton()` can scan your XML code and automatically define all variables you will probably need -- for instance, you wouldn't bother to include a check box if you don't use its value or state afterwards. So we can start writing the actual R code generating JS immediately.

> TIP
> The function `rk.JS.scan()` can also scan existing XML files for variables.

The package has some functions for JS code constructs that are commonly used in RKWard plugins, like the `echo()` function or `if() {...} else {...}` conditions. There are some differences between JS and R, e.g., for `paste()` in R you use the comma to concatenate character strings, whereas for `echo()` in JS you use '+', and lines must end with a semicolon. By using the R functions, you can almost forget about these differences and keep writing R code.

These functions can take different classes of input objects: Either plain text, R objects with XML code like above, or in turn results of some other JS functions of the package. In the end, you will

always call `rk.paste.JS()`, which behaves similar to `paste()`, but depending on the input objects it will replace them with their XML ID, JavaScript variable name or even complete JavaScript code blocks.

For the t-test example, we need two JS objects: One to calculate the results, and one to print them in the `printout()` function:

```
JS.calc <- rk.paste.JS(
  echo("res <- t.test (x=", var.x, ", y=", var.y, ", hypothesis=\"", test. ↩
      hypothesis, "\""),
  js(
    if(check.paired){
      echo(", paired=TRUE")
    },
    if(!check.paired && check.eqvar){
      echo(", var.equal=TRUE")
    },
    if(conf.level != "0.95"){
      echo(", conf.level=", conf.level)
    },
    linebreaks=TRUE
  ),
  echo(")\n"),
  level=2
)

JS.print <- rk.paste.JS(echo("rk.print (res)\n"), level=2)
```

As you can see, rkwarddev also provides an R implementation of the `echo()` function. It returns exactly one character string with a valid JS version of itself. You might also notice that all of the R objects here are the ones we created earlier. They will automatically be replaced with their variable names, so this should be quite intuitive. Whenever you need just this replacement, the function `id()` can be used, which also will return exactly one character string from all the objects it was given (you could say it behaves like `paste()` with a very specific object substitution).

The `js()` function is a wrapper that allows you to use R's **if(){...} else {...}** conditions like you are used to. They will be translated directly into JS code. It also preserves some operators like **<**, **>=** or **||**, so you can logically compare your R objects without the need for quoting most of the time. Let's have a look at the resulting 'JS.calc' object, which now contains a character string with this content:

```
  echo("res <- t.test (x=" + vrslCompare + ", y=" + vrslAgainst + ",  ↩
      hypothesis=\"" + radUsngtstth + "\"");
  if(chcPardsmpl) {
    echo(", paired=TRUE");
  } else {}
  if(!chcPardsmpl && chcAssmqlvr) {
    echo(", var.equal=TRUE");
  } else {}
  if(spnCnfdnclv != "0.95") {
    echo(", conf.level=" + spnCnfdnclv);
  } else {}
  echo(")\n");
```

> **NOTE**
> Alternatively for `if()` conditions nested in `js()`, you can use the `ite()` function, which behaves similar to R's `ifelse()`. However, conditional statements constructed using `ite()` are usually harder to read and should be replaced by `js()` whenever possible.

15.2.3 Plugin map

This section is very short: We don't need to write a `.pluginmap` at all, as it can be generated automatically by `rk.plugin.skeleton()`. The menu hierarchy can be specified via the *`pluginmap`* option:

```
  [...]
  pluginmap=list(
    name="Two Variable t-Test",
    hierarchy=list("analysis", "means", "t-Test"))
  [...]
```

15.2.4 Help page

This section is very short as well: `rk.plugin.skeleton()` cannot write a whole help page from the information it has. But it can scan the XML document also for elements which probably deserve a help page entry, and automatically create a help page template for our plugin. All we have to do afterwards is to write some lines for each listed section.

> TIP
> The function `rk.rkh.scan()` can also scan existing XML files to create a help file skeleton.

15.2.5 Generate the plugin files

Now comes the final step, in which we'll hand over all generated objects to `rk.plugin.skeleton()`:

```
plugin.dir <- rk.plugin.skeleton("t-Test",
  xml=list(
    dialog=full.dialog,
    wizard=full.wizard),
  js=list(
    results.header="Two Variable t-Test",
    calculate=JS.calc,
    printout=JS.print),
  pluginmap=list(
    name="Two Variable t-Test",
    hierarchy=list("analysis", "means", "t-Test")),
  load=TRUE,
  edit=TRUE,
  show=TRUE)
```

The files will be created in a temporal directory by default. The last three options are not necessary, but very handy: *`load=TRUE`* will automatically add the new plugin to RKWards configuration (since it's in a temp dir and hence will cease to exist when RKWard is closed, it will automatically be removed again by RKWard during its next start), *`edit=TRUE`* will open all created files for editing in RKWard editor tabs, and *`show=TRUE`* will attempt to directly launch the plugin, so you can examine what it looks like without a click. You might consider adding *`overwrite=TRUE`* if you're about to run your script repeatedly (e.g. after changes to the code), as by default no files will be overwritten.

The result object 'plugin.dir' contains the path to the directory in which the plugin was created. This can be useful in combination with the function `rk.build.package()`, to build an actual R package to share your plugin with others -- e.g. by sending it to the RKWard development team to be added to our plugin repository.

15.2.6 The full script

To recapitulate all of the above, here's the full script to create the working t-test example. Adding to the already explained code, it also loads the package if needed, and it uses the `local()` environment, so all the created objects will not end up in your current workspace (except for 'plugin.dir'):

```
require(rkwarddev)

local({
  variables <- rk.XML.varselector(id.name="vars")
  var.x <- rk.XML.varslot("compare", source=variables, types="number",  ↩
      required=TRUE)
  var.y <- rk.XML.varslot("against", source=variables, types="number",  ↩
      required=TRUE)
  test.hypothesis <- rk.XML.radio("using test hypothesis",
    options=list(
      "Two-sided"=c(val="two.sided"),
      "First is greater"=c(val="greater"),
      "Second is greater"=c(val="less")
    )
  )
  check.paired <- rk.XML.cbox("Paired sample", value="1", un.value="0")
  basic.settings <- rk.XML.row(variables, rk.XML.col(var.x, var.y, test. ↩
      hypothesis, check.paired))

  check.eqvar <- rk.XML.cbox("assume equal variances", value="1", un.value ↩
      ="0")
  conf.level <- rk.XML.spinbox("confidence level", min=0, max=1, initial ↩
      =0.95)
  check.conf <- rk.XML.cbox("print confidence interval", val="1", chk=TRUE)
  conf.frame <- rk.XML.frame(conf.level, check.conf, rk.XML.stretch(),  ↩
      label="Confidence Interval")

  full.dialog <- rk.XML.dialog(
    label="Two Variable t-Test",
    rk.XML.tabbook(tabs=list("Basic settings"=basic.settings, "Options"= ↩
        list(check.eqvar, conf.frame)))
  )

  full.wizard <- rk.XML.wizard(
      label="Two Variable t-Test",
      rk.XML.page(
        rk.XML.text("As a first step, select the two variables you want to  ↩
            compare against
          each other. And specify, which one you theorize to be greater.  ↩
              Select two-sided,
          if your theory does not tell you, which variable is greater."),
        rk.XML.copy(basic.settings)),
      rk.XML.page(
        rk.XML.text("Below are some advanced options. It's generally safe  ↩
            not to assume the
          variables have equal variances. An appropriate correction will be ↩
               applied then.
          Choosing \"assume equal variances\" may increase test-strength,  ↩
              however."),
        rk.XML.copy(check.eqvar),
        rk.XML.text("Sometimes it's helpful to get an estimate of the  ↩
            confidence interval of
          the difference in means. Below you can specify whether one should ↩
```

```
              be shown, and
            which confidence-level should be applied (95% corresponds to a 5% ↩
                 level of
            significance)."),
          rk.XML.copy(conf.frame)))

  JS.calc <- rk.paste.JS(
    echo("res <- t.test (x=", var.x, ", y=", var.y, ", hypothesis=\"", test ↩
        .hypothesis, "\""),
    js(
      if(check.paired){
      echo(", paired=TRUE")
      },
      if(!check.paired && check.eqvar){
      echo(", var.equal=TRUE")
      },
      if(conf.level != "0.95"){
      echo(", conf.level=", conf.level)
      },
      linebreaks=TRUE
    ),
    echo(")\n"), level=2)

  JS.print <- rk.paste.JS(echo("rk.print (res)\n"), level=2)

  plugin.dir <<- rk.plugin.skeleton("t-Test",
    xml=list(
      dialog=full.dialog,
      wizard=full.wizard),
    js=list(
      results.header="Two Variable t-Test",
      calculate=JS.calc,
      printout=JS.print),
    pluginmap=list(
      name="Two Variable t-Test",
      hierarchy=list("analysis", "means", "t-Test")),
    load=TRUE,
    edit=TRUE,
    show=TRUE,
    overwrite=TRUE)
})
```

15.3 Adding help pages

If you want to write a help page for your plugin, the most straight forward way to do so is to add the particular instructions directly to the definitions of the XML elements they belong to:

```
variables <- rk.XML.varselector(
  id.name="vars",
  help="Select the data object you would like to analyse.",
  component="Data"
)
```

The text given to the *help* parameter can then be fetched by `rk.rkh.scan()` and written to the help page of this plugin component. For this to work technically, however, `rk.rkh.scan()` must

know which R objects belong to one plugin component. This is why you must also provide the *component* parameter and make sure it is identical for all objects belonging together.

Since you will usually combine many objects into one dialog and might also like to be able to re-use objects like the **<varslot>** in multiple components of your plugins, it is possible to globally define a component with the `rk.set.comp()`. If set, it is assumed that all the following objects used in your script belong to that particular component, until `rk.set.comp()` is being called again with a different component name. You can then omit the *component* parameter:

```
rk.set.comp("Data")
variables <- rk.XML.varselector(
  id.name="vars",
  help="Select the data object you would like to analyse."
)
```

To add global sections like **<summary>** or **<usage>** to the help page, you use functions like `rk.rkh.summary()` or `rk.rkh.usage()` accordingly. Their results are then used to set the list elements like *summary* or *usage* in the *rkh* parameter of `rk.plugin.component()`/`rk.plugin.skeleton()`.

15.4 Translating plugins

The rkwarddev package is capable of producing external plugins with full i18n support. For instance, all relevant functions generating XML objects offer an optional parameter to specify *i18n_context* or *noi18n_label*:

```
varComment <- rk.XML.varselector(id.name="vars", i18n=list(comment="Main ↩
    variable selector"))
varContext <- rk.XML.varselector(id.name="vars", i18n=list(context="Main ↩
    variable selector"))
cboxNoi18n <- rk.XML.cbox(label="Power", id.name="power", i18n=FALSE)
```

The above examples produce output like this:

```
# varComment
<!-- i18n: Main variable selector -->
  <varselector id="vars" />

# varContext
<varselector id="vars" i18n_context="Main variable selector" />

# cboxNoi18n
<checkbox id="power" noi18n_label="Power" value="true" />
```

There's also support for translatable JS code. In fact, the package tries add `i18n()` calls by default in places where this is usually helpful. The `rk.JS.header()` function is a good example:

```
jsHeader <- rk.JS.header("Test results")
```

This produces the following JS code:

```
new Header(i18n("Test results")).print();
```

But you can also manually mark strings in your JS code as translatable, by using the `i18n()` function just like you would if you wrote the JS file directly.

Appendix A

Reference

A.1 Types of properties/Modifiers

At some places in this introduction we've talked about 'properties' of GUI elements or otherwise. In fact there are several different types of properties. Usually you do not need to worry about this, as you can use common sense to connect any property to any other property. However, internally, there are different types of properties. What this matters for, is when fetching some special values in the JS-template. In getString (˝id˝)/getBoolean (˝id˝)/getList (˝id˝) statements you can also specify some so called 'modifiers' like this: `getString (´´id.modifier´´)`. This modifier will affect, in which way the value is printed. Read on for the list of properties, and the modifiers they each make available:

String properties

The most simple type of property, used to simply hold a piece of text. Modifiers:

No modifier (˝˝)
: The string as defined / set.

quoted
: The string in quoted form (suitable for passing to R as character).

Boolean properties

Properties that can either be on or off, true or false. For instance the properties created by <convert>-tags, also the property accompanying a <checkbox> (see below). The following values will be returned according to the given modifier:

No modifier (˝˝)
: By default the property will return 1 if it is true, and 0 otherwise. The recommended way to fetch boolean values is using `getBoolean()`. Note that for `getString()`, the string ˝0˝ will be returned when the property is false. This string would evaluate to true, not to false in JS.

˝labeled˝
: Returns the string ˝true˝ when true, ˝false˝, when false, or whichever custom strings have been specified (typically in a <checkbox>).

˝true˝
: Return the string as if the property was true, even if it is false

˝false˝
: Return the string as if the property was false, even if it is true

˝not˝
This actually returns another Boolean property, which is the reverse of the current (i.e. false if true, true if false)

˝numeric˝
Obsolete, provided for backwards compatibility. Same as no modifier ˝˝. Return ˝1˝ if the property is true, or ˝0˝ if it is false.

Integer properties

A property designed to hold an integer value (but of course it still returns a numeric character string to the JS-template). It does not accept any modifiers. Used in <spinbox>es (see below)

Real number properties

A property designed to hold a real number value (but of course it still returns a numeric character string to the JS-template). Used in <spinbox>es (see below)

No modifier (˝˝)
For `getValue() / getString()`, this returns the same as ˝formatted˝. In future versions, it will be possible to obtain a numeric representation, instead.

˝formatted˝
Returns the formatted number (as a string).

RObject properties

A property designed a selection of one or more R objects. Used most prominently in varselectors and varslots. The following values will be returned according to the given modifier:

No modifier (˝˝)
By default the property will return the full name of the selected object. If more than one object is selected, the object names will be separated by line breaks (˝\n˝).

˝shortname˝
Like above, but returns only short name(s) for the object(s). For instance an object inside a list would only be given the name it has inside the list, without the name of the list.

˝label˝
Like above, but returns the RKWard label(s) of the object(s) (if no label available, this is the same as shortname)

String list properties

This property holds a list of strings.

No modifier (˝˝)
For `getValue()/getString()`, this returns all strings separated by ˝\n˝. Any ˝\n˝ characters in each item are escaped as literal ˝\n˝. However, the recommended usage is to fetch the value with `getList()`, instead, which will return an array of strings.

˝joined˝
Returns the list as a single string, with items joined by ˝\n˝. In contrast to no modifier (˝˝), the individual strings are _not_ escaped.

Code properties

A property held by plugins that generated code. This is important for embedding plugins, in order to embed the code generated by the embedded plugin into the code generated by the embedding (top-level) plugin. The following values will be returned according to the given modifier:

No modifier (˝˝)
Returns the full code, i.e. the sections ˝preprocess˝, ˝calculate˝, ˝printout˝, and (but not ˝preview˝) concatenated to one string.

˝preprocess˝
Returns only the preprocess section of the code

˝calculate˝
Returns only the calculate section of the code

˝printout˝
Returns only the printout section of the code

˝preview˝
Returns the preview section of the code

A.2 General purpose elements to be used in any XML file (`.xml`, `.rkh`, `.pluginmap`)

<snippets>
Allowed as a direct child of the <document> node and only there. Should be placed near the top of the file. See section on using snippets. Only one <snippets> element may be present. Optional, no attributes.

<snippet>
Defines a single snippet. Allowed only as a direct child of the <snippets/> element. Attributes:

<id>
An identifier string for the snippet. Required.

<insert>
Insert the contents of a <snippet>. Allowed anywhere. Attributes:

<snippet>
The identifier string of the snippet to insert. Required.

<include>
Include the contents of another XML file (everything inside the <document> element of that file). Allowed anywhere. Attributes:

<file>
The filename, relative to the directory, the current file is in. Required.

A.3 Elements to be used in the XML description of the plugin

Properties held by the elements are listed in a separate section.

A.3.1 General elements

<document>

Needs to be present in each description.xml-file as the root-node. No special function. No attributes

<about>

Information about this plugin (author, licence, etc.). This element is allowed in both an individual plugin's .xml file, and in `.pluginmap` files. Refer to the `.pluginmap` file reference for reference details, the chapter on 'about' information for an introduction.

<code>

Defines where to look for the JS template to the plugin. Use only once per file, as a direct child of the document-tag. Attributes:

file

Filename of the JS template, relative to the directory the plugin-xml is in

<help>

Defines where to look for the help file for the plugin. Use only once per file, as a direct child of the document-tag. Attributes:

file

Filename of the help file, relative to the directory the plugin-xml is in

<copy>

Can be used as a child (direct or indirect) of the main layout elements, i.e. <dialog> and <wizard>. This is used to copy an entire block a xml elements 1:1. Attributes:

id

The ID to look for. The <copy> tag will look for a previous XML element that has been given the same ID, and copy it including all descendant elements.

copy_element_tag_name

In some few cases, you will want an almost literal copy, but change the tag-name of the element to copy. The most important example of this is, when you want to copy an entire <tab> from a dialog interface to the <page> of a wizard interface. In this case, you'd set copy_element_tag_name=˝page˝ to do this conversion automatically.

A.3.2 Interface definitions

<dialog>

Defines a dialog-type interface. Place the GUI-definition inside this tag. Use only once per file, as a direct child of the document-tag. At least one of ˝dialog˝ or ˝wizard˝ tags is required for a plugin. Attributes:

label

Caption for the dialog

recommended

Should the dialog be used as the ˝recommended˝ interface (i.e. the interface that will be shown by default, unless the user has configured RKWard to default to a specific interface)? This attribute does not currently have an effect, as it is implicitly ˝true˝, unless the wizard is recommended.

<wizard>

Defines a wizard-type interface. Place the GUI-definition inside this tag. Use only once per file, as a direct child of the document-tag. At least one of ″dialog″ or ″wizard″ tags is required for a plugin. Accepts only <page> or <embed>-tags as direct children. Attributes:

label

Caption for the wizard

recommended

Should the wizard be used as the ″recommended″ interface (i.e. the interface that will be shown by default, unless the user has configured RKWard to default to a specific interface)? Optional, defaults to ″false″.

A.3.3 Layout elements

All elements in this section accept an attribute id=″identifierstring″. This attribute is optional for all elements. It can be used, for example, to hide/disable the entire layout element and all the elements contained therein (see chapter GUI logic). The id-string may not contain ″.″ (dot) or ″;″ (semicolon), and should generally be limited to alphanumeric characters and the underscore (″_″). Only the additional attributes are listed.

<page>

Defines a new page inside a wizard. Only allowed as a direct child of a <wizard> element.

<row>

All direct children of a ″row″ tag will be placed left-to-right.

<column>

All direct children of a ″column″ tag will be placed top-to-bottom.

<stretch>

By default, elements in the GUI take up all the space that's available. For instance, if you have two columns side by side, the left one is packed with elements, but the right one only contains a lonely **<radio>**, the **<radio>** control will expand vertically, even though it does not really need the available space, and it will look ugly. In this case you really want to add a ″blank″ below the **<radio>**. For this, use the <stretch> element. It will simply use up some space. Don't overuse this element, usually it's a good idea for GUI elements to get all the available space, only sometimes will the layout become spaced out. The <stretch> element does not take any arguments, not even an ″id″. Also you can place no children inside the <stretch> element (in other words, you'll only ever use it as ″<stretch/>″)

<frame>

Draws a frame/box around its direct children. Can be used to visually group related options. Layout inside a frame is top-to-bottom, unless you place a <row> inside. Attributes:

label

Caption for the frame (optional)

checkable

Frames can be made checkable. In this case, all contained elements will be disabled when the frame is unchecked, and enabled, when it is checked. (optional, defaults to ″false″)

checked

For checkable frames only: Should the frame be checked by default? Defaults to ″true″. Not interpreted for non-checkable frames.

<tabbook>

Organizes elements in a tabbook. Accepts only <tab>-tags as direct children.

<tab>

Defines a page in a tabbook. Place the GUI-definition for the tab inside this tag. May be used only as a direct child of a <tabbook> tag. A <tabbook> should have at least two defined tabs. Attributes:

label
Caption for the tab page (required)

<text>

Shows the text enclosed in this tag in the GUI. Some simple HTML-style markup is supported (notably <b>, <i>, <p>, and
). Please keep formatting to a minimum, however. Inserting a completely empty line adds a hard line break. Attributes:

type
Type of the text. One of ˝normal˝, ˝warning˝ or ˝error˝. This influences the look of the text (optional, defaults to normal)

A.3.4 Active elements

All elements in this section accept an attribute id=˝identifierstring˝. This attribute is required for all elements. Only the additional attributes are listed. The id-string may not contain ˝.˝ (dots).

<varselector>

Provides a list of available objects from which the user can select one or more. Requires one or more <varslot>s as a counterpart to be useful. Attributes:

label
Label for the varselector (optional, defaults to ˝Select variable(s)˝)

<varslot>

Used in conjunction with a ˝varselector˝ to allow the user to select one or more variables. Attributes:

label
Label for the varslot (recommended, defaults to ˝Variable:˝)

source
The varselector to fetch the selection from (required, unless you connect manually or using source_property)

source_property
An arbitrary property to copy values from, when the select button is clicked. If specified, this overrides the ˝source˝-attribute.

required
Whether - for submitting the code - it is required that this varslot holds a valid value. See required-property (optional, defaults to false)

multi
Whether the varslot holds only one (default, ˝false˝), or several objects

allow_duplicates
Whether the varslot may accept only unique objects (default, ˝false˝), or if the same object may be added several times.

min_vars
Only meaningful if multi=˝true˝: Minimum number of vars to be selected for the selection to be considered valid (optional, defaults to ˝1˝)

min_vars_if_any
: Only meaningful if multi=˝true˝: Some varslots may be considered valid, if, for instance, the varslot is either empty, or holds at least two values. This specifies how many variables have to be selected if any at all (2 in the example). (optional, defaults to ˝1˝)

max_vars
: Only meaningful if multi=˝true˝: Maximum number of variables to select (optional, defaults to ˝0˝, which means no maximum)

classes
: If you specify one or more R classnames (separated by spaces (˝ ˝)), here, the varslot will only accept objects belonging to those classes (optional, *use with great care*, the user should not be prevented from making valid choices, and R has *a lot* of different classes!)

types
: If you specify one or more variables types (separated by spaces (˝ ˝)), here, the varslot will only accept objects of those types. Valid types are ˝unknown˝, ˝number˝, ˝string˝, ˝factor˝, ˝invalid˝. (Optional, *use with great care*, the user should not be prevented from making valid choices, and RKWard does not always know the type of a variable)

num_dimensions
: The number of dimensions, an object needs to have. ˝0˝ (the default) means, any number of dimensions is acceptable. (optional, defaults to ˝0˝)

min_length
: The minimum length, an object needs to have in order to be acceptable. (optional, defaults to ˝0˝)

max_length
: The maximum length, an object needs to have in order to be acceptable. (optional, defaults to the largest integer number representable on the system)

<valueselector>

Provides a list of available strings (not R objects) to be selected in one or more accompanying <valueslot>s. String options can be defined using <option>-tags as direct children (see below), or set using dynamic properties. Attributes:

label
: Label for the valueselector (optional, defaults to no label)

<valueslot>

Used in conjunction with a <valueselector> to allow the user to select one or more string items. This element is mostly identical to <varslot>, and shares the same attributes, except for those which refer to properties of the acceptable items (i.e. classes, types, num_dimensions, min_length, max_length).

<radio>

Defines a group of radio-exclusive buttons (only one can be selected at a time). Requires at least two <option>-tags as direct children. No other tags are allowed as children. Attributes:

label
: Label for the radio control (recommended, defaults to ˝Select one:˝)

<dropdown>

Defines a group of options of which one and only one can be selected at the same time, using a dropdown list. This is functionally equivalent to a <radio>, but looks different. Requires at least two <option>-tags as direct children. No other tags are allowed as children. Attributes:

label
Label for the dropdown list (recommended, defaults to ″Select one:″)

<select>

Provides a list of available strings from which the user can select an arbitrary number. String options can be defined using <option>-tags as direct children (see below), or set using dynamic properties. Attributes:

label
Label for the <select> (optional, defaults to no label)

<option>

Can only be used as a direct child of a <radio>, <dropdown>, <valueselector> or <select> element. Represents one selectable option in a radio control or dropdown list. As <option> elements are always part of one of the selection elements, they do not normally have an ″id″ of their own, but see below. Attributes:

label
Label for the option (required)

value
The string value the parent element will return if this option is checked/selected (required)

checked
Whether the option should be checked/selected by default ″true″ or ″false″. In a <radio> or <dropdown>, only one option may be set to `checked=´´true´´`, and if no option is set to checked, the first option in the parent element will be checked/selected automatically. In a <select>, any number of options may be set to checked. (optional, default to ″false″)

id
Specifying the ″id″ parameter for the <option> elements is optional (and in fact it's recommended, not to set an ″id″, unless you really need one). However, specifying an ″id″ will allow you to enable/disable <option>s, dynamically, by connecting to the boolean property `id_of_radio.id_of_optionX.enabled`. Currently this works for options inside <radio> or <dropdown> elements, only; <valueselector> and <select> options do not currently support ids.

<checkbox>

Defines a check box, i.e. a single option that can either be set to on or off. Attributes:

label
Label for the check box (required)

value
The value the check box will return if checked (required)

value_unchecked
The value that will be returned if the check box is not checked (optional, defaults to ″″, i.e. an empty string)

checked
Whether the option should be checked by default ″true″ or ″false″ (optional, default to ″false″)

<frame>

The frame element is generally used as a pure layout element, and is listed in the section on layout elements. However, it can also be made checkable, thus acting like a simple check box at the same time.

<input>

Defines a free text input field. Attributes:

label
Label for the input field (required)

initial
Initial text of the text field (optional, defaults to ″″, i.e. an empty string)

size
One of ″small″, ″medium″, or ″large″. ″large″ defines a multi-line input field, ″small″, and ″medium″ are single line fields (optional, defaults to ″medium″)

required
Whether - for submitting the code - it is required that this input is not empty. See required-property (optional, defaults to false)

<matrix>
A table for entering matrix data (or vectors) in the GUI.

> NOTE
> This input element is *not* optimized for entering/editing large amounts of data. While there is no strict limit on the size of a <matrix>, in general it should not exceed around ten rows / columns. If you expect larger data, allow users to select it as an R object (which may be a good idea as an alternative option, in almost *every* instance where you use a matrix element).

Attributes:

label
Label for the table (required)

mode
One of ″integer″, ″real″, or ″string″. The type of data that will be accepted in the table (required)

min
Minimum acceptable value (for matrices of type ″integer″ or ″real″) (optional, defaults to the smallest representable value)

max
Maximum acceptable value (for matrices of type ″integer″ or ″real″) (optional, defaults to the largest representable value)

allow_missings
Whether missing (empty) values are allowed in the matrix. This is implied for matrices or mode ″string″ (optional, defaults to false).

allow_user_resize_columns
When set to true, the user can add columns by typing on the rightmost (inactive) cells (optional, defaults to true).

allow_user_resize_rows
When set to true, the user can add rows by typing on the bottommost (inactive) cells (optional, defaults to true).

rows
Number of rows in the matrix. Has no effect for allow_user_resize_rows=″true″.

> NOTE
> This can also be controlled by setting the ″rows″ property.

(optional, defaults to 2).

columns
Number of columns in the matrix. Has no effect for allow_user_resize_columns=″true″.

> NOTE
> This can also be controlled by setting the ″columns″ property.

(optional, defaults to 2).

min_rows
Minimum number of rows in the matrix. The matrix will refuse shrink below this size. (optional, defaults to 0; see also: `allow_missings`.).

min_columns
Minimum number of columns in the matrix. The matrix will refuse shrink below this size. (optional, defaults to 0; see also: `allow_missings`.).

fixed_height
Force the GUI element to stay at its initial height. Do not use in combination with matrices, where the number of rows may change in any way. Useful, esp. when creating a vector input element (columns=˝1˝). With this option set to true, no horizontal scroll bar will be shown, even in the matrix exceeds the available width (as this would affect the height). (optional, defaults to false).

fixed_width
Slightly misnamed: Assume the column count will not change. The last (or typically only) column will be stretched to take up the available width. Do not use in combination with matrices, where the number of columns may change in any way. Useful, esp. when creating a vector input element (rows=˝1˝). (optional, defaults to false).

horiz_headers
Strings to use for the horizontal header, separated by ˝;˝. The header will be hidden, if set to ˝˝. (optional, defaults to column number).

vert_headers
Strings to use for the vertical header, separated by ˝;˝. The header will be hidden, if set to ˝˝. (optional, defaults to row number).

<optionset>

A UI for repeating a set of options for an arbitrary number of items (introduction to optionsets). Attributes:

min_rows
If specified, the set will be marked invalid, unless it has at least this number of rows (optional, integer).

min_rows_if_any
Like min_rows, but will only be tested, if there is at least one row (optional, integer).

max_rows
If specified, the set will be marked invalid, unless it has at most this number of rows (optional, integer).

keycolumn
Id of the column to act as keycolumn. An optionset with a (valid) keycolumn will act as a ˝driven˝ optionset. An optionset with no keycolumn will allow manual insertion / removal of items. The keycolumn must be marked as external. (optional, defaults to no keycolumn).

Child-elements:

<optioncolumn>

Declares one optioncolumn of the set. For each value that you want to fetch from the optionset, you must declare a separate <optioncolumn>. Attributes:

id
The id of the optioncolumn (required, string).

external
Set to true, if the optioncolumn is controlled from outside the optionset (optional, boolean, defaults to false).

label
If given, the optioncolumn will be displayed in a column by that label (optional, string, defaults to not displayed).

connect
The property to connect this optioncolumn to, given as id inside the <content>-area. For external <optioncolumn>s, the corresponding value will be set to the externally set value. For regular (non-external) <optioncolumn>s, the corresponding row of the <optioncolumn>-property, will be set when the property changes inside the content-area. (optional, string, defaults to not connected).

default
Only for external columns: The value to assume for this column, if no value is known for an entry. Rarely useful. (Optional, defaults to empty string)

<content>
Declare the content / UI of the set. No attributes. All usual active, passive, and layout elements are allowed as childname elements. In addition, in earlier versions of RKWard (up to 0.6.3), the special child-element **<optiondisplay>** was allowed. This is obsolete in RKWard 0.6.4, and should simply be removed from existing plugins.

<logic>
Optional specification of UI logic to apply *inside* the contents region the optionset. See the reference on <logic>

<browser>
An element designed to select a single filename (or directory name). Note that this field will take any string, even though it is meant to be used for files, only:

label
Label for the browser (optional, defaults to ˝Enter filename˝)

initial
Initial text of the browser (optional, defaults to ˝˝, i.e. an empty string)

type
One of ˝file˝, ˝dir˝, or ˝savefile˝. To select an existing file, existing directory, or non-existing file, respectively (optional, defaults to ˝file˝)

allow_urls
Whether (non-local) urls can be selected (optional, defaults to ˝false˝)

filter
File type filter, e.g. (˝*.txt *.csv˝ for .txt and .csv files). A separate entry for ˝All files˝ is added, automatically (optional, defaults to ˝˝, i.e. All files)

required
Whether - for submitting the code - it is required that the field is not empty. Note that this does not necessarily mean, the selected filename is valid! See required-property (optional, defaults to true)

<saveobject>
An element designed to select the name of an R object to save to (i.e. generally not already existing, in contrast to a varslot):

label
Label for the input (optional, defaults to ˝Save to:˝)

initial
Initial text of the input (optional, defaults to ˝my.data˝)

required
Whether - for submitting the code - it is required that the field holds a permissible object name. See required-property (optional, defaults to true)

checkable
In many use cases, saving to an R object is optional. In these cases, a check box can be integrated into the saveobject-element using this attribute. When set to true, the saveobject will be activated / deactivated by the check box. See the active-property of saveobject (optional, defaults to false)

checked
For checkable saveobject-elements, only: Whether the control is checked/enabled by default (optional, defaults to false)

<spinbox>

A spinbox in which the user can select a numeric value, using either direct keyboard input or small up/down arrows. Attributes:

label
Label for the spinbox (recommend, default to ˝Enter value:˝)

min
The lowest value the user is allowed to enter in the spinbox (optional, defaults to the lowest value technically representable in the spinbox)

max
The largest value the user is allowed to enter in the spinbox (optional, defaults to the highest value technically representable in the spinbox)

initial
The initial value shown in the spinbox (optional, defaults to ˝0˝)

type
One of ˝real˝ or ˝integer˝. Whether the spinbox will accept real numbers or only integers (optional, defaults to ˝real˝)

default_precision
Only meaningful if the spinbox is of type=˝real˝. Specifies the default number of decimal places shown in the spinbox (only this many trailing zeros will be shown). When the user presses the up/down arrows, this decimal place will be changed. The user may still be able to enter values with a higher precision, however (see below) (optional, defaults to ˝2˝)

max_precision
The maximum number of digits that can be meaningfully represented (optional, defaults to ˝8˝)

<formula>

This advanced element allows the user to select a formula/set of interactions from selected variables. For instance for a GLM, this element can be used to allow the user to specify the interaction-terms in the model. Attributes:

fixed_factors
The ID of the varslot holding the selected fixed factors (required)

dependent
The ID of the varslot holding the selected dependent variable (required)

<embed>

Embed a different plugin into this one (see chapter on embedding). Attributes:

component
The registered name of the component to embed (see chapter on registering components) (required)

as_button
If set to ˝true˝, only a pushbutton will be placed in the embedding GUI, the embedded GUI will only be shown (in a separate window) when the pushbutton is pressed (optional, default is ˝false˝)

label
Only meaningful if as_button=˝true˝: The label of the button (recommend, default is ˝Options˝)

<preview>

Checkbox to toggle preview functionality. Note that starting with version 0.6.5 of RKWard **<preview>** preview elements are special- cased in plugin dialogs (not wizards): They will be placed in the button-column, irrespective of where exactly they are defined in the UI. It is still a good idea to define them at a sensible place in the layout, for backwards compatibility. Attributes:

label
Label of the box (optional, default is ˝Preview˝)

mode
Type of preview. Supported types are ˝plot˝ (see chapter on graph previews), ˝output˝ (see chapter on (HTML) output previews), ˝data˝ (see data previews), and ˝custom˝ (see custom previews). (optional, default is ˝plot˝)

placement
Placement of the preview: ˝attached˝ (to the main workplace), ˝detached˝ (standalone window), ˝docked˝ (attached to the plugin dialog) and ˝default˝ (currently this is the same as ˝docked˝, but might become user-configurable at some point). In general, it is recommended to leave this as the default setting for best UI-consistency (optional, default is ˝default˝)

active
Whether the preview is active by default. In general, only docked previews should be made active by default, and even for these, there is a reason why the default is in-active previews (optional, default is ˝false˝)

A.3.5 Logic section

<logic>

The containing element for the logic section. All elements below are allowed only inside the <logic> element. The <logic> element is allowed only as a direct child of the <document> element (at most once per document), or of <optionset> elements (at most once per optionset). The document's logic section applies to both <dialog> and <wizard> GUIs in the same way.

<external>

Creates a new (string) property that is supposed to be connected to an outside property if the plugin gets embedded. See section on ˝incomplete˝ plugins. Attributes:

id
The ID of the new property (required)

default
The default string value of the new property, i.e. the value used, if the property is not connected to an outside property (optional, defaults to an empty string)

<i18n>

Creates a new (string) property that is supposed to be provide an i18n'ed label. Attributes:

id
The ID of the new property (required)

label
The label. This will be translated. (required)

<set>

Set a property to a fixed value (of course, if you additionally connect the property to some other property, the value does not remain fixed). For instance, if you embed a plugin, but want to hide some of its elements, you might set the visibility property of those elements to false. Useful esp. for embedded/embedding plugins. Note: If there are several <set> elements for a single `id`, the latest one to be defined takes precedence. This will sometimes be useful to rely on when using <include>d parts. Attributes:

id
: The ID of the property to set (required)

to
: The string value to set the property to (required). Note: For boolean properties such as visibility, enabledness, you'll typically set the to attribute to either to=˝true˝ or to=˝false˝.

<convert>

Create a new boolean properties that depends on the state of one or more different properties. Attributes:

id
: The ID of the new property (required)

sources
: The ids of the properties this property will depend on. One or more properties may be specified, separated by ˝;˝ (required)

mode
: The mode of conversion/operation. One of ˝equals˝, ˝notequals˝, ˝range˝, ˝and˝, ˝or˝. If in mode equals, the property will only be true, if the value of all of its sources equals the attribute standard (see below). If in at mode notequals, the property will only be true, if the value of all of its sources are different from the attribute standard (see below). If in mode range, the sources have to be numeric (integer or real). The property will only be true, if all sources are in the range specified by the attributes min and max (see below). If in mode and, the sources have to be boolean properties. The property will only be true, if all the sources are true simultaneously. If in mode or, the sources have to be boolean properties. The property will only be true, if at least one of the sources is true. (required)

standard
: Only meaningful in modes equals or notequals: the string value to compare against (required if in one of these modes)

min
: Only meaningful in mode range: the minimum value to compare against (optional, defaults to the lowest floating point number representable on the machine)

max
: Only meaningful in mode range: the maximum value to compare against (optional, defaults to the largest floating point number representable on the machine)

require_true
: If set to ˝true˝, the property will become required, and will only be considered valid, if its state is true/on. Hence, if the property is false, it will block the **Submit** button (optional, defaults to ˝false˝).

> CAUTION
> If you use this, make sure the user can easily detect what's wrong, such as by showing an explanatory <text>.

<switch>

Create a new property that will relay to different target properties (or fixed strings) based on the value of a condition property. This allows to create logic similar to `if()` or `switch()` constructs. Attributes:

id
The ID of the new property (required)

condition
The id of the condition property (required)

Child elements:

<true>
If the condition property is boolean, you can specify the two child elements <true> and <false> (and only these). (Required, if <false> is also given)

<false>
If the condition property is boolean, you can specify the two child elements <true> and <false> (and only these). (Required, if <true> is also given)

<case>
If the condition property is not boolean, you can supply an arbitrary number of <case>-elements, one for each value of the condition property that you want to match (at least one such element is required, if the condition property is not boolean)

<default>
If the condition property is not boolean, the optional <default>-element, allows to specify the behavior, if no <case> element is matches the value of the condition property (optional, allowed only once, in combination with one or more <case> elements).

Child elements <true>, <false>, <case>, and <default> take the following attributes:

standard
Only for <case>-elements: The value to match the condition property against (required, string).

fixed_value
A fixed string that should be supplied as the value of the <switch> property, if the current condition matches (required, if dynamic_value is not supplied).

dynamic_value
The *id* of the target property that should be supplied as the value of the <switch> property, if the current condition matches (required, if fixed_value is not supplied).

<connect>
Connects two properties. The client property will be changed whenever the governor property changes (but not the other way around!). Attributes:

client
The ID of the client property, i.e. the property that will be adjusted (required)

governor
The ID of the governor property, i.e. the property that will adjusts the client property. This may include a modifier (required)

reconcile
If ˝true˝, the client property will make adjust the governor property on connection in such a way that the governor property will only accept values that are also acceptable by the client (e.g. suppose the governor is a numeric property with min value ˝0˝, and the client is a numeric property with min value ˝100˝. The min of both properties will be adjusted to 100, if reconcile=˝true˝). Generally works only for properties of the same basic type (optional, default to ˝false˝)

<dependency_check>
Creates a boolean property that is true, if the specified dependencies are met, false otherwise. The xml syntax of the element is the same as that of the **<dependencies>** element, described in the `.pluginmap` reference. As of RKWard 0.6.1, only the RKWard and R version specifications are taken into account, not dependencies on packages or pluginmaps.

<script>

Define script code to control UI logic. See the section on scripted GUJI logic for details. The script code to run can be given either using the ´´file´´ attribute, or as a (commented) text of the element. The **<script>** element is not allowed in the **<logic>** section of an optionset. Attributes:

file
File name of the script file. (required)

A.4 Properties of plugin elements

All layout elements, and all active elements hold the following properties, accessible via ˝id_of_element.name_of_property˝:

visible
Whether the GUI element is visible or not (boolean)

enabled
Whether the GUI element is enabled or not (boolean)

required
Whether the GUI element is required (to hold a valid setting) or not. Note that any element which is disabled or hidden is also implicitly non-required (boolean).

In addition to this, some elements have additional properties you can connect to. Most active elements also have a ˝default˝ property whose value will be returned on calls to `getBoolean/getString/getList (´´...´´)`, if no specific property was named, as described below.

<text>
Default property is text

text
The text displayed (text)

<varselector>
No default property

selected
The objects currently selected. You probably do not want to use this. Used internally (RObject)

root
The root/parent object of the objects offered for selection (RObject)

<varslot>
Default property is ˝available˝

available
All objects held in the varslot (RObject)

selected
Of the objects held in the varslot, those that are currently selected. You probably do not want to use this. Used internally (RObject)

source
A copy of the objects selected in the corresponding varselector. You probably do not want to use this. Used internally (RObject)

<valueselector>

Default property is ˝selected˝

selected
The strings currently selected. Modifier ˝labeled˝ to retrieve the corresponding labels. In a <valueselector> you probably do not want to use this, directly (only in a <select>). (read/write StringList)

available
The list of string values to select from. (read/write StringList)

labels
Labels to display for the string values. (read/write StringList)

<valueslot>

Same as <varslot>, but the properties are lists of strings, instead of RObjects.

<radio>

Default property is ˝string˝

string
The value of the currently selected option (string)

number
The number of the currently selected option (options are numbered top-to-bottom, starting at 0) (integer)

<dropdown>

Same as <radio>

<select>

Same as <valueselector>

<option>

No default property. ˝enabled˝ is the *only* property, and it is not currently available for options inside a <select> or <valueselector>. <option> does not have the ˝visible˝ or ˝required˝ properties.

enabled
Whether this single option should be enabled or disabled. In most cases you will enable/disable the entire <radio< or <dropdown<, instead. But this can be used to dynamically set the enabledness of a single option inside a <radio< or <dropdown< (bool)

<checkbox>

Default property is ˝state.labeled˝, which means that the values specified by the `value`, and `value_unchecked`-attributes are returned, *not* the displayed label of the check box.

state
State of the check box (on or off). Note that useful modifiers of this property (as of all boolean properties) are ˝not˝ and ˝labeled˝ (see types of properties). However, often it is most useful to connect to the property with no modifier, i.e. ˝*checkbox_id*.state˝, which will return the state of the check box in a format suitable for use in an if statement (0 or 1). (boolean)

<frame>

Default property is ˝checked˝, if - and only if - the frame is checkable. For non-checkable frames, there is no default property.

checked
Available for checkable frames, only: state of the check box (on or off). Note that useful modifiers of this property (as of all boolean properties) are ˝not˝ and ˝numeric˝ (see types of properties). (boolean)

<input>
Default property is ˝text˝

text
Current text in the input field (string)

<matrix>
Default property is ˝cbind˝.

rows
Number of rows in the matrix (integer). If the matrix allows the user to add / remove rows, this property should be treated as read-only. Otherwise, changing it, will change the size of the matrix.

columns
Number of columns in the matrix (integer). If the matrix allows the user to add / remove columns, this property should be treated as read-only. Otherwise, changing it, will change the size of the matrix.

tsv
Data in the matrix in tsv format (string; read-write). Note that compared to the usual tsv layout, *columns*, not rows, are separated by newline characters, and cells within a column are separated by tabulator characters.

0,1,2...
The data from a single column (0 for leftmost column). `getValue()`/`getString()` returns this as a single string, separated by ˝\n˝. However, the recommended way to get this is using `getList()`, which returns this column as an array of strings.

row.0,row.1,row.2...
The data from a single row (0 for topmost row). `getValue()`/`getString()` returns this as a single string, separated by ˝\n˝. However, the recommended way to get this is using `getList()`, which returns this row as an array of strings.

cbind
Data in a format suitable for pasting to R, wrapped in a cbind statement (string; read-only).

<optionset>
No default property.

row_count
Number of items in the optionset (integer). Read-only.

current_row
Currently active item in the optionset (integer). -1 for no active item. Read-write.

optioncolumn_ids
For each <optioncolumn> you define, a string list property will be created with the specified id.

<browser>
Default property is ˝selection˝

selection
Current text (selected file name) in the browser (string)

<saveobject>

Default property is ˝selection˝

selection
Full name of the selected object (string; read-only - to set this programmatically, use ˝parent˝, and ˝objectname˝)

parent
The parent object of the selected object. This is always an existing R object of a type that can contain other objects (e.g. a list or data.frame). When set to an empty string or an invalid object, ˝.GlobalEnv˝ is assumed (RObject)

objectname
The base-name of the selected object, i.e. the string entered by the user (changed to a valid R name, if necessary) (string)

active
For checkable saveobjects, only: Whether the control is checked/enabled. Always true for non-checkable saveobjects (bool)

<spinbox>

Default property is either ˝int˝ or ˝real.formatted˝ depending on the spinbox's mode

int
Integer value held by the spinbox, or nearest integer, if in real mode (integer)

real
Real value held by the spinbox (and integer, if in integer) (real)

<formula>

Default property is ˝model˝

model
The current model string (string)

table
The data.frame holding the required variables. If variables from only one data.frame are used, the name of that data.frame is returned. Otherwise a new data.frame is constructed as required (string)

labels
If variables from multiple data.frames are involved, their names may get mangled (for instance, if both data.frames contain a variable named ˝x˝). This returns a list with the mangled names as indices and the descriptive label as value (string)

fixed_factors
The fixed factors. You probably do not want to use this. Used internally (RObject)

dependent
The dependent variable(s). You probably do not want to use this. Used internally (RObject)

<embed>

No default property

code
The code generated by the embedded plugin (code)

<preview>

Default property is ˝state˝

state
Whether the preview box is checked (not necessarily whether the preview has already been shown) (boolean)

<convert>
This element (used in the <logic> section) is special, in that is technically *is* a property, instead of just holding one or more properties. It is of boolean kind. Note that useful modifiers of this property (as of all boolean properties) are ˝not˝ and ˝numeric˝ (see types of properties)

<switch>
This element (used in the <logic> section) is special, in that is technically *is* a (string) property, instead of just holding one or more properties. It allows to switch between several target properties depending on the value of a condition property, or to re-map values of the condition property. Any modifiers that you supply are passed on to the target properties, thus, e.g. if all target properties are RObject properties, you can use the ˝shortname˝ modifier on the switch, too. However, if the target properties are of different types, using modifiers may lead to errors. For `fixed_value`s, any modifier is dropped, silently. Note that target properties, when accessed through a switch, are always read-only!

A.5 Embeddable plugins shipped with the official RKWard release

A number of embeddable plugins is shipped with RKWard, and can be used in your own plugins. Detailed documentation is currently available only in these plugins source or help files. However, here is a list to give you a quick overview of what is available:

ID	Pluginmap	Description	Example usage
rkward::plot_options	embedded.pluginmap	Provides a wide range of options for plots. Most plotting plugins utilize this.	Plots->Barplot, most other plotting plugins
rkward::color_chooser	embedded.pluginmap	Very simple plugin for specifying a color. Current implementation provides a list of color names. Future implementations may provide more elaborate color picking.	Plots->Histogram
rkward::plot_stepfun_options	embedded.pluginmap	Step function plot options	Plots->ECDF plot
rkward::histogram_options	embedded.pluginmap	Histogram (plot) options	Plots->Histogram
rkward::barplot_embed	embedded.pluginmap	Barplot options	Plots->Barplot
rkward::one_var_tabulation	embedded.pluginmap	Provides tabulation on a single variable.	Plots->Barplot
rkward::limit_vector_length	embedded.pluginmap	Limit the length of a vector (to the n largest or smallest elements).	Plots->Barplot

rkward::level_select	embedded.pluginmap	Provides a <valueselector> filled with the levels (or unique values) of a vector.	Data->Recode Categorical data
rkward::multi_input	embedded.pluginmap	Combines spinbox, input and radio controls to provide input for character, numeric, logical data.	Data->Recode Categorical data

Table A.1: Standard embeddable plugins

A.6 Elements for use in `.pluginmap` files

<document>

Needs to be present in each `.pluginmap` file as the root-node (exactly once). Attributes:

base_prefix

Filenames specified in the `.pluginmap` file are assumed to be relative to the directory of the `.pluginmap` file + the prefix you specify here. Useful, esp., if all your components are located below a single subdirectory.

namespace

A namespace for the component ids. When looking up components for embedding, the components will be retrievable via a string ˝namespace::component_id˝. Set to ˝rkward˝ for now.

id

An optional identifier string for this `.pluginmap`. Specifying this allows third authors to refer to and load your `.pluginmap` from theirs (see chapter on handling dependencies).

priority

One of `´´hidden´´`, `´´low´´`, `´´medium´´`, or `´´high´´`. `.pluginmap`s with priority ˝medium˝ or ˝high˝ are activated, automatically, when RKWard first finds them. Use `priority=´´hidden´´` for `.pluginmap`s that are not meant to be activated, directory (only meant for inclusion). In the current implementation this does not actually hide the `.pluginmap`, however. (Optional, defaults to ˝medium˝).

<dependencies>

This element, specifying dependencies, is allowed as a direct child of the <document> element (once), and as a child of <component> elements (once for each <component> element). Specifies the dependencies that must be met in order to use the plugin(s). See the chapter on dependencies for an overview. Attributes:

rkward_min_version, rkward_max_version

Minimum and maximum allowed version of RKWard. Version specifications may include non-numeric suffixes, like ˝0.5.7z-devel1˝. If a specified dependency is not met, the plugin(s) it applies to *will be ignored*. More information. Optional; if not specified, no minimum / maximum version of RKWard will be required.

R_min_version, R_max_version
Minimum and maximum allowed version of R. Version specifications may *not* include non-numeric suffixes, like ˝0.5.7z-devel1˝. The R version dependency will be shown on the plugins' help pages, but does not have any direct effect, as of RKWard 0.6.1. More information. Optional; if not specified, no minimum / maximum version of R will be required.

Child elements:

<package>
Adds a dependency on a specific R package. Attributes:

name
Package name (required).

min_version, max_version
Minimum / maximum allowed version (optional).

repository
Repository where the package can be found. Optional, but highly recommended, if the package is not available on CRAN.

<pluginmap>
Adds a dependency on a specific RKWard `.pluginmap`. Attributes:

name
Id string of the required `.pluginmap` (required).

min_version, max_version
Minimum / maximum allowed version (optional).

url
URL where the `.pluginmap` can be found. Required.

<about>
May be present exactly once as a direct child of the <document> element. Contains meta information on the `.pluginmap` (or plugin). See the chapter on 'about' information for an overview. Attributes:

name
User visible name. Optional. Does not have to be the same as the ˝id˝.

version
Version number. Optional. The format is not restricted, but to be on the safe side, please follow common versioning schemes such as ˝x.y.z˝.

releasedate
Release date specification. Optional in format ˝YYYY-MM-DD˝.

shortinfo
A *short* description of the plugin / `.pluginmap`. Optional.

url
URL where more information can be found. Optional, but recommended.

copyright
Copyright specification, e.g. ˝2012-2013 by John Doe˝. Optional, but recommended.

licence
License specification, e.g. ˝GPL˝ or ˝BSD˝. Please make sure to accompany your files with a complete copy of the relevant licence! Optional, but recommended.

category
Category of plugin(s), e.g. ˝Item response theory˝. As of RKWard 0.6.1, no categories are predefined. Optional.

Child elements:

<author>
Adds information on an author. Attributes:

name, given, family
Either specify the full name for *name*, or specify both *given* and *family*, separately.

role
Author role description (optional).

email
E-mail address, where author can be contacted. Required. May be set to the rkward-devel mailing list, if you are subscribed, and your plugin is meant to be included in the official RKWard release.

url
URL with more information on the author, e.g. homepage (optional).

<components>
Needs to be present exactly once as a direct child of the <document> element. Contains the individual <component>-elements described below. No attributes.

<component>
One or more <component> elements should be given as direct children of the <components> element (and only there). Registers a component/plugin with rkward. Attributes:

type
For future extension: The type of component/plugin. Always set to ˝standard˝ for now (the only type currently supported).

id
The ID by which this component can be retrieved (for placing it in the menu (see below), or for embedding). See <document>-namespace above.

file
Required at least for components of type=˝standard˝: The filename of the XML file describing the GUI.

label
The label for this component, when placed in the menu hierarchy.

<attribute>
Defines an attribute of a component. Only meaningful for import plugins so far. Only allowed as a direct child of <component>. Attributes:

id
Id of the attribute

value
Value of the attribute

labels
Label associated with the attribute

<hierarchy>
Needs to be present exactly once as a direct child of the <document> element. Describes where the components declared above should be placed in the menu hierarchy. Accepts only <menu> elements as direct children. No attributes.

<menu>
One or more <menu> elements should be given as direct children of the <hierarchy> element. Declares a new (sub-)menu. If a menu by the given ID (see below) already exists, the two menus are merged. The <menu> element is allowed either as a direct child of the <hierarchy> element (top level menu), or as the direct child on any other <menu> element (sub-menu). Conversely, the <menu> element accepts other <menu> elements or <entry> elements as children. Attributes:

id
An identifier string of the menu. Useful, when menu definitions are read from several `.pluginmap` files, to make sure plugins can be placed in the same menu(s). Some menu-ids such as ˝file˝ refer to predefined menus (in this case the ˝File˝ menu). Be sure to check with existing `.pluginmap` files to use consistent ids.

label
A label for the menu.

group
Allows to control ordering of menu entries. See menu item ordering. Optional.

<entry>

A menu entry, i.e. a menu option to invoke a plugin. May be used only as a direct child of a <menu> element, accepts no child elements. Attributes:

component
The ID of the component that should be invoked, when this menu entry is activated.

group
Allows to control ordering of menu entries. See menu item ordering. Optional.

<group>

Declares a group of items in the menu. See menu item ordering. Attributes:

id
The name of this group.

separated
Optional. If set to ˝true˝ the item in this group will be visually separated from surrounding items.

group
The name of the group to append this group to (optional).

<context>

Declares the entries in a context. Only allowed as a direct child of the <document> tag. Accepts only <menu> tags as direct children. Attributes:

id
The ID of the context. Only two contexts are implemented so far: ˝x11˝ and ˝import˝.

<require>

Include another `.pluginmap` file. This `.pluginmap` file is only loaded once, even if it is <require>d from several other files. The most important use case is to include a pluginmap file, which declares some components, which are embedded by components declared in this `.pluginmap`. <require>-elements are only allowed as direct children of the <document>-node. Attributes:

file
The filename of the `.pluginmap` to include. This is seen relative to the directory of the current `.pluginmap` file + the base_prefix (see above, <document>-element). If you do not know the relative path to the `.pluginmap` to be included, use the *`map`* attribute to refer to it by id, instead.

map
To include a `.pluginmap` file from a different package (or an RKWard `.pluginmap` from your external `.pluginmap`), you can refer to it by its *`namespacename::id`*, as specified in the required `.pluginmaps` <document> element. Inclusion will fail, if no `.pluginmap` by that id is known (e.g. not installed on the user's system). You should use this method for including `.pluginmaps` outside your package, only. For maps inside your package, specifying a relative path (*`file`* attribute) is faster, and more reliable.

A.7 Elements for use in .rkh (help) files

<document>

Needs to be present in each `.xml` file as the root-node (exactly once). No attributes.

<title>

Title of the help page. This is *not* interpreted for help pages for a plugin (this takes the title from the plugin itself), only for stand-alone pages. No attributes. The text contained within the <title> tag will become the caption of the help page. May only be defined once, as a direct child of the <document> node.

<summary>

A short summary of the help page (or what this plugin is used for). This will always be shown at the top of the help page. No attributes. The text contained within the <summary> tag will be displayed. Recommended but not required. May only be defined once, as a direct child of the <document> node.

<usage>

A slightly more elaborate summary of the usage. This will always be shown directly after the <summary>. No attributes. The text contained within the <usage> tag will be displayed. Recommended for plugin help pages, but not required. May only be defined once, as a direct child of the <document> node.

<section>

A general purposes section. May be used any number of times as a direct child of the <document> node. These sections are displayed in the order of their definition, but all *after* the <usage> section and *before* the <settings> section. The text contained within the <section> tag will be displayed.

id

An identifier needed to jump to this section from the navigation bar (or a link). Needs to be unique within the file. Required, no default.

title

The title (caption) of this section. Required, no default.

short_title

A short title suitable to be displayed in the navigation bar. Optional, defaults to the full title.

<settings>

Defines the section containing reference on the various GUI settings. Only meaningful and only used for plugin related help pages. Use as a direct child of the <document>. May contain only <setting> and <caption> elements as direct children. No attributes.

<setting>

Explains a single setting in the GUI. Only allowed as a direct child of the <settings> element. The text contained within the element is displayed.

id

The ID of the setting in the plugin .xml. Required, no default.

title

An optional title for the setting. If omitted (omission is recommended in most cases), the title will be taken from the plugin .xml.

<caption>

A caption to visually group several settings. May only be used as a direct child of the <settings> element.

id
The ID of the corresponding element (typically a <frame>, <page> or <tab>) in the plugin .xml.

title
An optional title for the caption. If omitted (omission is recommended in most cases), the title will be taken from the plugin .xml.

<related>
Defines a section containing links to further related information. Will always be displayed after the <settings> section. No attributes. The text contained within the <related> tag will be displayed. Typically this will contain an HTML-style list. Recommended for plugin help pages, but not required. May only be defined once, as a direct child of the <document> node.

<technical>
Defines a section containing technical information of no relevance to end users (such as internal structure of the plugin). Will always be displayed last in a help page. No attributes. The text contained within the <related> tag will be displayed. Not required, and not recommended for most plugin help pages. May only be defined once, as a direct child of the <document> node.

<link>
A link. Can be used in any of the sections described above.

href
The target url. Note that several rkward specific URLs are available. See section on writing help pages for details.

<label>
Inserts the value of a UI label. Can be used in any of the sections described above.

id
The id of the element in the plugin, of which to copy the `label`-attribute.

<various html tags>
Most basic html tags are allowed within the sections. Please keep manual formatting to a minimum, however.

A.8 Functions available for GUI logic scripting

Class ˝Component˝
Class which represents a single component or component-property. The most important instance of this class is the variable ˝gui˝ which is predefined as the root property of the current component. The following methods are available for instances of class ˝Component˝:

absoluteId(base_id)
Returns the absolute ID of *base_id*, or - if base_id is omitted - the identifier of the component.

getValue(id)
Discouraged. Use `getString(), getBoolean(), or getList()`, instead. Returns the value of the given child property. Returns the value of this property, if ID is omitted.

getString(id)
Returns the value of the given child property as a string. Returns the value of this property, if ID is omitted.

getBoolean(id)

Returns the value of the given child property as a boolean (if possible). Returns the value of this property, if ID is omitted.

getList(id)

Returns the value of the given child property as an array of strings (if possible). Returns the value of this property, if ID is omitted.

setValue(id, value)

Set the value of the given child property to *value*.

getChild(id)

Return an instance of the child-property with the given *id*.

addChangeCommand(id, command)

Execute *command* whenever the child property given by *id* changes.

Class ˝RObject˝

Class which represents a single R object. An instance of this class can be obtained by using **makeRObject(objectname)**. The following methods are available for instances of class ˝RObject˝:

> WARNING
> If any commands are still pending in the backend, the information provided by these methods can be out-of-date by the time that the plugin code is run. Do *not* rely on it for critical operations (risking loss of data).

getName()

Returns the absolute name of the object.

exists()

Returns whether the object exists. You should generally check this before using any of the methods listed below.

dimensions()

Returns an array of dimensions (similar to **dim()** in R).

classes()

Returns an array of classes (similar to **class()** in R).

isClass(class)

Returns true, if the object is of class *class*.

isDataFrame()

Returns true, if the object is a data.frame.

isMatrix()

Returns true, if the object is a matrix.

isList()

Returns true, if the object is a list.

isFunction()

Returns true, if the object is a function.

isEnvironment()

Returns true, if the object is an environment.

isDataNumeric()

Returns true, if the object is a vector of numeric data.

isDataFactor()

Returns true, if the object is a vector of factor data.

isDataCharacter()
Returns true, if the object is a vector of character data.

isDataLogical()
Returns true, if the object is a vector of logical data.

parent()
Returns an instance of ˝RObject˝ representing the parent of this object.

child(childname)
Returns an instance of ˝RObject˝ representing the child *childname* of this object.

Class ˝RObjectArray˝

An array of RObject instances. An instance of this class can be obtained by using **makeRObjectArray(objectnames)**. It is particularly useful when dealing with varslots which allow to select multiple objects.

include()-function

include(filename) can be used to include a separate JS file.

doRCommand()-function

doRCommand(command, callback) can be used to query R for information. Please read the section on querying R from inside a plugin for details, and caveats.

Appendix B

Troubleshooting during plugin development

So you've read all the documentation, did everything right, and still cannot get it to work? Don't worry, we'll work it out. First thing to do is: Activate **RKWard Debug Messages** - window (available from the **Windows** - menu, or right click on one of the tool bars), and then start your plugin, again. As a general rule of thumb, you should not see any output in the messages window when your plugin gets invoked, or at any other time. If there is one, it's likely related to your plugin. See if it helps you.

If everything seems fine on the console, try to increase the debug-level (from the command line, using **rkward --debug-level 3**, or by setting debug level to 3 in **Settings** → **Configure RKWard** → **Debug**). Not all messages shown at higher debug levels necessarily indicate a problem, but chance are, your problem shows up somewhere between the messages.

If you still cannot find out what's wrong, don't despair. We know this is complicated stuff, and - after all - possibly you've also come across a bug in RKWard, and RKWard needs to be fixed. Just write to the development mailing list, and tell us about the problem. We'll be happy to help you.

Finally, even if you found out how to do it on your own, but found the documentation to be not-so-helpful or even wrong in some respects, please tell us on the mailing list as well, so we can fix/improve the documentation.

Appendix C

License