Introdução à escrita de plugins para o RKWard

Thomas Friedrichsmeier
Meik Michalke
Tradução: Marcus Gama

Introdução à escrita de plugins para o RKWard

Conteúdo

Lista de Tabelas

Resumo

Este é um guia para escrever plugins para o RKWard.

Capítulo 1

Introdução

Este documento descreve como escrever seus próprios plugins. A documentação cresceu bastante ao longo do tempo. Não se assuste com isso. Recomendamos que você leia os quatro passos básicos (conforme descrito abaixo) para ter uma ideia geral de como as coisas funcionam. Depois disso, você pode consultar o sumário para verificar quais tópicos avançados podem ser relevantes para você.

Para perguntas e comentários, escreva para a lista de discussão de desenvolvimento do RKWard.

Você não precisa ler isto para usar o RKWard. Este documento trata da extensão do RKWard. Ele é direcionado a usuários avançados, ou pessoas dispostas a ajudar a aprimorar o RKWard.

Escrever um plugin padrão é basicamente um processo de quatro etapas:

- Colocar uma nova Ação na hierarquia do menu
- Descrever a aparência e o comportamento da GUI do plugin
- Definir como o código R deve ser gerado a partir das configurações que o usuário faz na GUI
- Adicionar uma página de ajuda ao seu plugin

Essas etapas serão tratadas em sequência.

Alguns conceitos avançados podem ser usados nessas quatro etapas, mas são tratados em capítulos separados, para simplificar:

- Lógica da GUI
- Embutindo plugins dentro de plugins
- Conceitos úteis para a criação de várias séries de plugins semelhantes

Além disso, nenhum dos capítulos mostra todas as opções, mas sim apenas os conceitos básicos. Uma referência completa das opções é fornecida separadamente.

Capítulo 2

Preliminares: O que são plugins no RKWard? Como eles funcionam?

Claro que a primeira pergunta que você pode ter é: quais partes da funcionalidade do RKWard são implementadas usando plugins? Ou: o que os plugins podem fazer?

Uma maneira de responder a isso é: desmarque todos os arquivos `.pluginmap` em **Configurações** → **Configurar RKWard** → **Plugins** e veja o que está faltando. Uma resposta um pouco mais útil: a maioria das funções estatísticas reais acessíveis pela GUI são implementadas usando plugins. Além disso, você pode criar GUIs bastante flexíveis para todos os tipos de operações usando plugins.

O paradigma básico por trás dos plugins do RKWard é aquele que explicaremos neste documento: um arquivo XML descreve a aparência da GUI. Um arquivo JavaScript adicional é usado para gerar a sintaxe R a partir das configurações da GUI. Ou seja, os plugins não precisam realizar cálculos estatísticos. Em vez disso, eles geram a sintaxe R necessária para executar esses cálculos. A sintaxe R é então enviada ao backend R para avaliação e, normalmente, um resultado é exibido na janela de saída.

Continue a leitura nos próximos capítulos para ver como isso é feito.

Capítulo 3

Criando entradas no menu

Ao criar um novo plugin, você precisa informá-lo ao RKWard. Portanto, a primeira coisa a fazer é escrever um arquivo `.pluginmap` (ou modificar um existente). O formato do `.pluginmap` é XML. Vou mostrar um exemplo (e, claro, certifique-se de ter configurado o RKWard para carregar seu `.pluginmap` -- **Configurações** → **Configurar RKWard** → **Plugins**):

> DICA
> Após ler este capítulo, dê uma olhada também no pacote rkwarddev. Ele fornece algumas funções R para criar a maioria das tags XML do RKWard para você.

```
<!DOCTYPE rkpluginmap>
```

O doctype não é realmente interpretado, mas defina-o como *"rkpluginmap"* mesmo assim.

```
<document base_prefix="" namespace="meusplugins" id="meupluginmap">
```

O atributo *base_prefix* pode ser usado se todos os seus plugins estiverem em um diretório comum. Basicamente, você pode omitir esse diretório dos nomes de arquivo especificados abaixo. É seguro deixá-lo como *""*.

Como você verá abaixo, todos os plugins recebem um identificador único, *id*. O *namespace* é uma forma de organizar esses IDs e reduzir a probabilidade de criar identificadores duplicados acidentalmente. Internamente, basicamente o namespace e, em seguida, uma '::' são adicionados a todos os identificadores que você especificar neste `.pluginmap`. Em geral, se você pretende distribuir seus plugins em um pacote R, é uma boa prática usar o nome do pacote como parâmetro *namespace*. Os plugins distribuídos com a distribuição oficial RKWard têm *namespace="rkward"*.

O atributo *id* é opcional, mas especificar um id para o seu `.pluginmap` permite que outras pessoas façam com que seus `.pluginmap`s carreguem o seu `.pluginmap` automaticamente (consulte a seção sobre dependências).

```
<componentes>
```

Componentes? Não estamos falando de plugins? Sim, mas no futuro, os plugins serão apenas uma classe especial de componentes. O que fazemos aqui, então, é registrar todos os componentes/plugins com RKWard. Vejamos um exemplo de entrada:

```
<component type="standard" id="t_test_duas_vars" file="t_test_duas_vars.xml ↩
    " label="t-Test com duas variáveis" />
```

Primeiro, o atributo *type*: Deixe-o com *"standard"* por enquanto. Outros tipos ainda não foram implementados. O *id* já foi mencionado. Cada componente precisa de um identificador único (em seu namespace). Escolha um que seja facilmente reconhecível. Evite espaços e caracteres especiais. Eles não são proibidos, até o momento, mas podem ter significados especiais. Com o atributo *file*, você especifica onde a descrição do próprio plugin está localizada. Isso é relativo ao diretório onde o arquivo .pluginmap está localizado e ao *base_prefix* acima. Finalmente, dê ao componente um rótulo. Este rótulo será exibido onde quer que o plugin seja colocado no menu (ou, no futuro, talvez em outros lugares também).

Normalmente, um arquivo .pluginmap contém vários componentes, então aqui estão mais alguns:

```
<component type="standard" id="teste_nao_implementado" file="means/ ↩
    nao_implementado.xml" />
                <component type="standard" id="t_test_ficticio" file="means ↩
                    /ttests/ficticio.xml" label="Este é um t-test fictício"  ↩
                    />
                <component type="standard" id="descritivo" file="descritivo ↩
                    .xml" label="Estatísticas descritivas" />
                <component type="standard" id="corr_matriz" file=" ↩
                    corr_matriz.xml" label="Correlação da matriz" />
                <component type="standard" id="simples_anova" file=" ↩
                    simples_anova.xml" label="Simples Anova" />
        </components>
```

OK, este foi o primeiro passo. O RKWard agora sabe que esses plugins existem. Mas como invocá-los? Eles precisam ser colocados em uma hierarquia de menus:

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

Logo abaixo da tag **<hierarchy>**, você começa a descrição de em qual **<menu>** seus plugins devem estar. Com a linha acima, você basicamente diz que seu plugin deve estar no menu **Análise** (não necessariamente diretamente lá, mas em um submenu). O menu **Análise** é padrão no RKWard, então ele não precisa ser criado do zero. No entanto, se ele ainda não existir, você pode usar o atributo *label* para dar um nome a ele. Finalmente, o *id* identifica novamente este **<menu>**. Isso é necessário para que vários arquivos .pluginmap possam colocar seus plugins nos mesmos menus. Eles fazem isso procurando por um **<menu>** com o *id* fornecido. Se o ID ainda não existir, um novo menu será criado. Caso contrário, as entradas serão adicionadas ao menu existente.

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

Basicamente a mesma coisa aqui: Agora definimos um submenu para o menu **Análise**. Ele será chamado **Métodos**.

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

E um nível final na hierarquia do menu: Um submenu do submenu **Métodos**.

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

Agora, finalmente, este é o menu em que queremos colocar o plugin. A tag **<entry>** indica que este é realmente o item, e não outro submenu. O atributo *component* refere-se ao *id* que você atribuiu ao plugin/componente acima.

```
<entry component="teste_t_ficticio" />
                                </menu>
                                <entry component="teste_t_ficticio" />
                        </menu>
                        <menu id="frequency" label="Frequência" index="2"/>
```

Caso você tenha se perdido: Este é outro submenu do menu **Análise**. Veja a captura de tela abaixo. Vamos pular algumas opções que não estão visíveis, marcadas com [...].

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

Estas são as entradas finais visíveis nas capturas de tela abaixo.

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

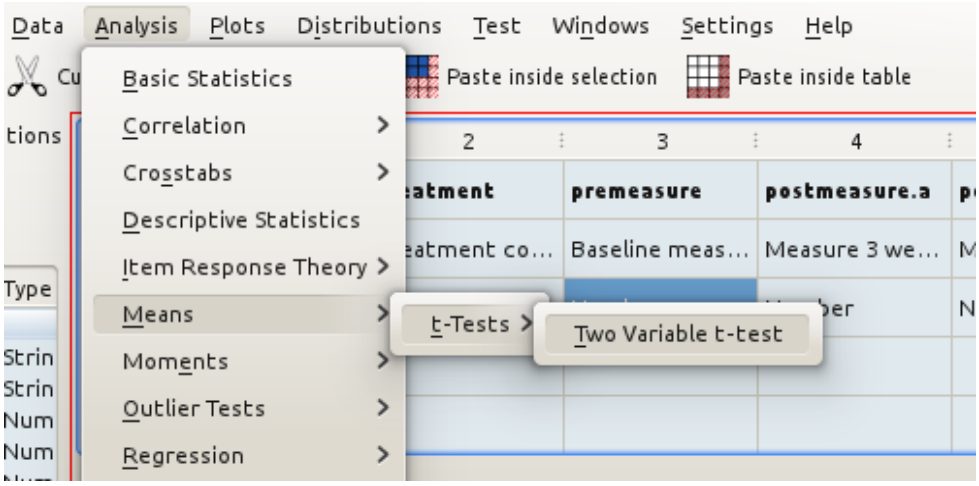
É claro que você também pode colocar seus plugins em menus diferentes de **Análise**.

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

Mesmo em menus padrão como **Arquivo**. Tudo o que você precisa é do `id` correto.

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

É assim que se faz. E esta captura de tela mostra o resultado:



Confuso? A maneira mais fácil de começar provavelmente é pegar alguns dos arquivos `.plugin map` existentes que acompanham a distribuição e modificá-los de acordo com suas necessidades. Além disso, se precisar de ajuda, não hesite em escrever para a lista de discussão de desenvolvimento.

3.1 Controlar a ordem dos itens do menu

Por padrão, todos os itens (entradas/submenus) dentro de um menu serão classificados alfabeticamente, automaticamente. Em *alguns* casos, você pode querer ter mais controle. Nesse caso, você pode agrupar os elementos da seguinte forma:

- Você pode definir grupos dentro de qualquer menu desta forma. Todos os elementos pertencentes ao mesmo grupo serão agrupados:

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

- Se você deseja que o grupo seja visualmente separado das outras entradas, use:

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

- Entradas, menus e grupos podem ser adicionados a um grupo específico, usando:

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

- Na verdade, também é possível definir grupos (sem linhas separadoras) implicitamente:

```
<entry component="primeiro" group="a"/>
                <entry component="terceiro"/>
                <entry component="segundo" group="a"/>
```

- Os nomes dos grupos são específicos para cada menu. O grupo ″a″ no menu ″Dados″ não entra em conflito com o grupo ″a″ no menu ″Análise″, por exemplo.
- O caso de uso mais comum é a definição de grupos na parte superior ou inferior de um menu. Para isso, existem grupos predefinidos ″superior″ e ″inferior″ em cada menu.
- As entradas dentro de cada grupo são classificadas em ordem alfabética. Os grupos aparecem na ordem de declaração (a menos que sejam anexados a outro grupo, é claro).
- Menus e entradas sem especificação de grupo formam logicamente um grupo (″″), também.

Capítulo 4

Definir a GUI

4.1 Definir um diálogo

No capítulo anterior você viu como registrar um plugin com o RKWard. O ingrediente mais importante foi especificar o caminho para um arquivo XML com uma descrição de como o plugin realmente se parece. Neste capítulo, você aprenderá como criar este arquivo XML.

> DICA
> Após ler este capítulo, dê uma olhada também no pacote rkwarddev. Ele fornece algumas funções R para criar a maioria das tags XML do RKWard para você.

Mais uma vez, vamos apresentar um exemplo. O exemplo é uma versão (ligeiramente simplificada) do teste t de duas variáveis.

```
<!DOCTYPE rkplugin>
```

O doctype não é realmente interpretado. Defina-o como *rkplugin*, mesmo assim.

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

Todos os plugins geram algum código. Atualmente, a única maneira de fazer isso é usando JS, como detalhado no próximo capítulo. Isso define onde procurar o código JS. O nome do arquivo é relativo ao diretório onde o plugin XML está localizado.

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

Geralmente é uma boa ideia também fornecer uma página de ajuda para o seu plugin. O nome do arquivo dessa página de ajuda é fornecido aqui, relativo ao diretório onde o plugin XML está localizado. A criação de páginas de ajuda está documentada aqui. Se você não fornecer um arquivo de ajuda, omita esta linha.

```
<dialog label="Teste t com duas variáveis">
```

Como você sabe, os plugins podem ter uma interface de diálogo, uma interface de assistente ou ambas. Aqui, começamos a definir uma interface de diálogo. O atributo *label* especifica o título do diálogo.

```
<tabbook>
                        <tab label="Configurações básicas">
```

Os elementos da GUI podem ser organizados usando um catálogo de abas. Aqui, definimos um catálogo de abas como o primeiro elemento da caixa de diálogo. Use **<tabbook>**[...]**</tabbook>** para definir o catálogo de abas e, em seguida, para cada página do catálogo de abas, use **<tab>**[...]**</tab>**. O atributo *label* no elemento **<tab>** permite especificar uma legenda para essa página do catálogo de abas.

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

As tags **<row>** e **<column>** especificam o layout dos elementos da GUI. Aqui, você indica que gostaria de colocar alguns elementos lado a lado (da esquerda para a direita). O atributo *id* não é estritamente necessário, mas o utilizaremos posteriormente, ao adicionar uma interface de assistente ao nosso plugin. O primeiro elemento a ser colocado na linha é:

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

Usando esta tag simples, você cria uma lista da qual o usuário pode selecionar variáveis. Você precisa especificar um *id* para este elemento, para que o RKWard saiba como encontrá-lo.

> ATENÇÃO
> Você NÃO pode usar um ponto (.) na string *id*.

```
<column>
```

Em seguida, aninhamos uma **<column>** dentro da linha. Ou seja, os elementos seguintes serão colocados uns acima dos outros (de cima para baixo), e todos ficarão à direita do **<varselector>**.

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

Esses elementos são a contraparte do **<varselector>**. Eles representam 'slots' nos quais o usuário pode inserir variáveis. Você notará que o *source* é definido com o mesmo valor que o *id* do **<varselector >**. Isso significa que os **<varslot>**s obterão suas variáveis do varselector. Os **<varslot>**s também precisam receber um *id*. Eles podem ter um *label* e podem ser definidos como *required*. Isso significa que o botão **Enviar** não será habilitado até que o **<varslot>** contenha um valor válido. Finalmente, o atributo *type* ainda não foi interpretado, mas será usado para garantir que apenas os tipos corretos de variáveis sejam permitidos no **<varslot>**.

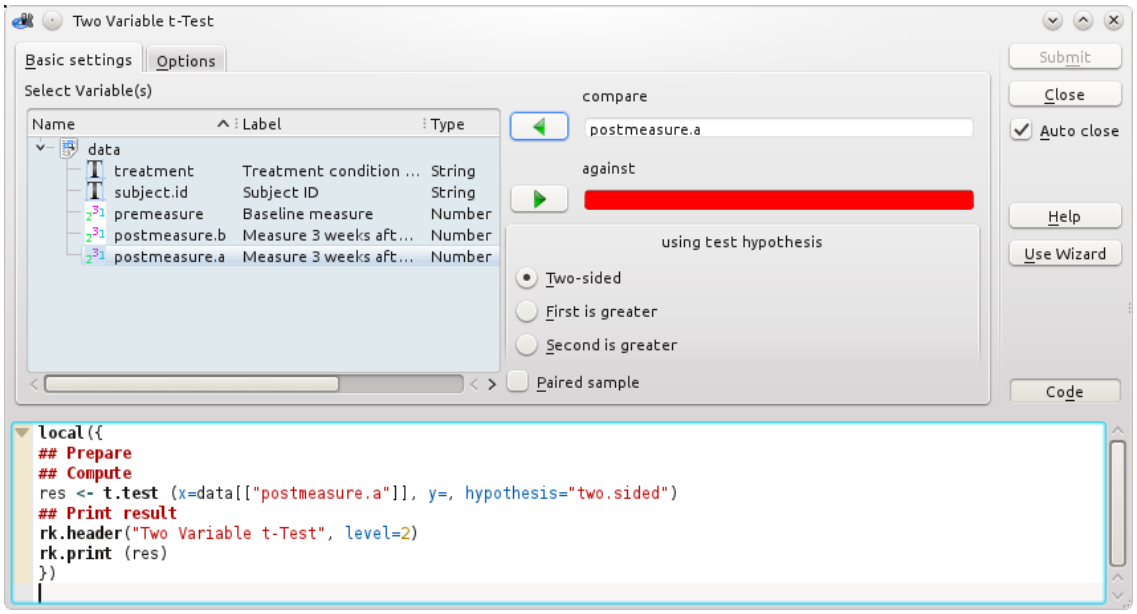
Caso você esteja se perguntando sobre o atributo *i18n_context*: Ele serve para fornecer contexto e auxiliar na tradução correta da palavra "against", usada como rótulo do **<varslot>**, mas não afeta diretamente a funcionalidade do plugin. Mais informações sobre isso em um capítulo separado.

```
<radio id="hypothesis" label="usar hipótese de teste">
                                                                <option value="two. ↩
                                                                    sided" label=" ↩
                                                                    Dois lados"/>
                                                                <option value=" ↩
                                                                    greater" label=" ↩
                                                                    Primeiro é maior ↩
                                                                    "/>
                                                                <option value="less ↩
                                                                    " label="Segundo ↩
                                                                     é maior"/>
                                                    </radio>
```

Aqui, você define um grupo de botões exclusivos **<radio>**. O grupo possui um *label* e um *id*. Cada **<option>** (botão) possui um *label* e recebe um *value*. Este é o valor que o elemento **<radio>** retornará quando a opção for selecionada.

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

Cada tag precisa ser fechada. Colocamos todos os elementos que queríamos (os dois **<varslots>** e o **<radio>**) no **<column>**. Colocamos todos os elementos que queríamos (o **<varselector>** e o **<column>** com esses elementos) no **<row>**. E colocamos todos os elementos que queríamos na primeira página no **<tabbook>**. Ainda não terminamos de definir o **<tabbook>** (mais páginas virão), e é claro que também há mais por vir no **<dialog>**. Mas esta captura de tela mostra basicamente o que fizemos até agora:



Observe que não especificamos os botões **Submit**, **Close**, etc., nem a visualização do código. Esses elementos são gerados automaticamente. Mas, é claro, ainda precisamos definir a segunda página do **<tabbook>**:

```
<tab label="Options">
                                    <checkbox id="varequal" label="assumir ↩
                                        variâncias iguais" value=", var.equal= ↩
                                        TRUE"/>
```

Por padrão, os elementos serão dispostos de cima para baixo, como em uma **<column>**. Como é isso que queremos aqui, não precisamos declarar explicitamente um layout **<row>** ou **<column>**. O primeiro elemento que definimos é uma caixa de seleção. Assim como os **<radio><option>**s, a caixa de seleção tem um *label* e um *value*. O *value* é o que é retornado se a caixa de seleção estiver marcada. Obviamente, a caixa de seleção também precisa de um *id*.

```
<frame label="Intervalo de confiança" id="frame_conf_int">
```

Aqui está mais um elemento de layout: Para sinalizar que os dois elementos abaixo pertencem ao mesmo conjunto, desenhamos uma **<frame>** (caixa). Esse quadro pode ter um *label* (legenda). Como o quadro é apenas um elemento de layout passivo, ele não precisa de um *id*, mas ainda assim definimos um aqui, pois nos referiremos a ele mais tarde, ao definir uma interface de assistente adicional.

```
<checkbox id="confint" label="imprimir intervalo de confidência" value="1"  ↩
    checked="true"/>
                                        <spinbox type="real" id="conflevel" ↩
                                             label="nível de confidência"  ↩
                                            min="0" max="1" initial="0.95"/>
                                </frame>
```

Dentro do **<frame>**, colocamos outro **<checkbox>** (usando *checked="true"*, sinalizamos que a caixa de seleção deve estar marcada por padrão) e um **<spinbox>**. O spinbox permite que o usuário selecione um valor entre *"min"* e *"max"*, com o valor padrão/inicial *"0.95"*. Definir o parâmetro *type* como *"real"* indica que números reais são aceitos em oposição a *type="integer"*, que aceitaria apenas números inteiros.

> NOTA
> Também é possível, e muitas vezes preferível, tornar o próprio **<frame>** selecionável, em vez de adicionar um **<checkbox>** dentro dele. Consulte a referência para obter detalhes. Isso não foi feito aqui, para fins ilustrativos.

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

Isso é tudo para a segunda página do **<tabbook>**, todas as páginas do **<tabbook>** e todos os elementos do **<dialog>**. Terminamos de definir a aparência do diálogo.

```
</document>
```

Finalmente, fechamos a tag **<document>**, e isso é tudo. A GUI está definida. Você pode salvar o arquivo agora. Mas como a sintaxe R é gerada a partir das configurações da GUI? Abordaremos isso no próximo capítulo. Primeiro, porém, vamos examinar a adição de uma interface de assistente e algumas considerações gerais.

4.2 Adicionar uma interface de assistente

Na verdade, não precisamos definir uma interface adicional **<wizard>**, mas aqui está como isso seria feito. Para adicionar uma interface de assistente, você adicionará uma tag **<wizard>** no mesmo nível da tag **<dialog>**:

```
<wizard label="Teste t com duas variáveis">
                        <page id="firstpage">
                                <text>Como primeiro passo, selecione as  ↩
                                    duas variáveis que deseja comparar.
                                        E especifique qual você irá  ↩
                                            teorizar como a maior. Selecione ↩
                                             dois lados,
                                        se sua teoria não te informa qual  ↩
                                            variável é maior.</text>
                                <copy id="main_settings_row"/>
                        </page>
```

Parte disso é bastante autoexplicativo: adicionamos uma tag **<wizard>** com um *label* para o assistente. Como um assistente pode conter várias páginas que são exibidas uma após a outra, em seguida, definimos a primeira **<page>** e inserimos uma nota explicativa **<text>** nela. Depois, usamos uma tag **<copy>**. O que isso faz é, na verdade, nos poupar de definir novamente o que

já escrevemos para o **<dialog>**: a tag copy procura outra tag com o mesmo *id* anteriormente no XML. Isso é definido na seção **<dialog>** e é uma **<row>** na qual estão o **<varselector>**, **<varslots>** e o 'hypothesis' **<radio>** controle. Tudo isso é copiado 1:1 e inserido diretamente no elemento **<copy>**.

Agora a segunda página:

```
<page id="secondpage">
                                <text>Abaixo estão algumas opções avançadas ↩
                                    . É normalmente seguro não considerar  ↩
                                    que as
                                        variáveis possuam variâncias iguais ↩
                                            . Uma correção apropriada será  ↩
                                            aplicada então.
                                        Escolher "assumir variâncias iguais ↩
                                            " pode aumentar a força do teste ↩
                                            , no entanto.</text>
                                <copy id="varequal"/>
                                <text>Algumas vezes é útil estimar o  ↩
                                    intervalo de confidência da
                                        diferença em significado. Abaixo  ↩
                                            você pode especificar se uma  ↩
                                            deve ser exibida, e
                                        qual nível de confidência deve ser  ↩
                                            aplicado (95% corresponde a um  ↩
                                            nível de significância de
                                        5%).</text>
                                <copy id="frame_conf_int"/>
                        </page>
                </wizard>
```

Aqui, praticamente a mesma coisa. Adicionamos alguns textos e, entre eles, **<copy>** outras seções da interface de diálogo.

Você pode, é claro, fazer com que a interface do assistente tenha uma aparência muito diferente da caixa de diálogo simples, e até mesmo não usar a tag **<copy>**. Certifique-se, no entanto, de atribuir o mesmo *id* aos elementos correspondentes em ambas as interfaces. Isso não só é usado para transferir configurações da interface da caixa de diálogo para a interface do assistente e vice-versa, quando o usuário alterna entre as interfaces (o que ainda não acontece na versão atual do RKWard), mas também simplifica a escrita do seu modelo de código (veja abaixo).

4.3 Algumas considerações sobre design da GUI

Esta seção contém algumas considerações gerais sobre quais elementos da GUI usar e onde usá-los. Se esta for sua primeira tentativa de criar um plugin, sinta-se à vontade para pular esta seção, pois ela não é relevante para o funcionamento básico da GUI. Volte aqui mais tarde para ver se você consegue refinar a interface gráfica do seu plugin de alguma forma.

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

Os três elementos **<radio>**, **<checkbox>**, **<dropdown>**, todos têm uma função semelhante: selecionar uma entre várias opções. Obviamente, uma caixa de seleção (checkbox) permite escolher apenas entre duas opções: marcada ou desmarcada, portanto, você não pode usá-la se houver mais de duas opções para escolher. Mas quando usar cada um dos elementos? Algumas regras práticas:

Se você se deparar com a criação de um **<radio>** ou **<dropdown>** com apenas duas opções, pergunte-se se a pergunta é essencialmente uma questão de sim/não. Por exemplo, uma escolha entre 'ajustar resultados' e 'não ajustar resultados', ou entre 'remover valores ausentes' e 'manter valores ausentes'. Nesse caso, um **<checkbox>** é a melhor opção: ocupa pouco espaço, terá o menor número de palavras nos rótulos e é mais fácil de ler para o usuário. Existem pouquíssimas situações em que você deve escolher um **<radio>** ao invés de um **<checkbox>**, quando há apenas duas opções. Um exemplo disso seria: 'Método de cálculo: 'pearson'/'spearman". Aqui, outros métodos podem ser considerados, e eles não formam exatamente um par de opostos.

A escolha entre um **<radio>** e um **<dropdown>** é, em grande parte, uma questão de espaço. O **<dropdown>** tem a vantagem de ocupar pouco espaço, mesmo que haja muitas opções para escolher. Por outro lado, um **<radio>** tem a vantagem de tornar todas as opções possíveis visíveis ao usuário de uma só vez, sem precisar clicar na seta do menu suspenso. Geralmente, se houver seis ou mais opções para escolher, um **<dropdown>** é preferível. Se houver cinco ou menos opções, um **<radio>** é a melhor escolha.

Capítulo 5

Gerar código R a partir de configurações da GUI

5.1 Usar JavaScript em plugins do RKWard

Agora temos uma GUI definida, mas ainda precisamos gerar algum código R a partir dela. Para isso, precisamos de outro arquivo de texto, `code.js`, localizado no mesmo diretório que o arquivo `description.xml`. Você pode ou não estar familiarizado com o JavaScript (ou, para ser tecnicamente preciso: ECMA-script). A documentação sobre JS pode ser encontrada em abundância, tanto impressa quanto na internet (por exemplo: https://developer.mozilla.org/en/Core_JavaScript_1.5_Guide). Mas, para a maioria dos propósitos, você não precisará saber muito sobre JS, pois usaremos apenas alguns recursos básicos.

> DICA
> Após ler este capítulo, dê uma olhada também no pacote rkwarddev. Ele fornece algumas funções em R para criar código JavaScript comumente usado no RKWard. Ele também pode detectar automaticamente variáveis usadas em um arquivo XML de plugin e criar código JavaScript básico a partir disso para você começar.

> NOTA
> Os arquivos de plugin `.js` são considerados codificados em UTF-8. Verifique a codificação do seu editor caso esteja usando caracteres não-ASCII.

Para o teste t de duas variáveis, o arquivo `code.js` tem a seguinte aparência (com comentários entre as linhas):

5.1.1 preprocess()

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

O arquivo JS está organizado em três funções separadas: `preprocess()`, `calculate()` e `printout()`. Isso ocorre porque nem todo o código é necessário em todos os estágios. Atualmente, a função preprocess não é realmente usada em muitos lugares (normalmente você a omitirá completamente).

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');
}
```

Esta função gera a sintaxe real do R a ser executada a partir das configurações da GUI. Vamos analisar isso em detalhes: O código a ser usado é gerado usando a instrução `echo()`. Analisando a instrução `echo()` passo a passo, a primeira parte dela é

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

como texto simples. Em seguida, precisamos preencher o valor que o usuário selecionou como a primeira variável. Buscamos isso usando `getString (~x~)`, e o anexamos à string a ser 'exibida'. Isso imprime o valor do elemento GUI com `id=~x~`: nosso primeiro **<checkbox>**. Em seguida, acrescentamos um ', ' e fazemos o mesmo para buscar o valor do elemento `~y~` - o segundo **<checkbox>**. Para a hipótese (o grupo **<radio>**), e a **<checkbox>** das variâncias iguais, o procedimento é muito semelhante.

> NOTA
> Em vez de concatenar strings usando o operador **+**, você também pode usar um "template literal", como este (observe que a string está entre crases (`)):
>
> ```
> echo(`res <- t.test (x=${ getString("x") }, y=${ getString("y") },  ↩
>     hypothesis="${ getString("hypothesis") }"`); // etc.
> ```

Observe que, em vez de concatenar os trechos de saída com '+', você também pode usar várias instruções `echo()`. Tudo é impresso em uma única linha. Para inserir uma quebra de linha no código gerado, insira um `~\n~` na string exibida. Em teoria, você pode até mesmo gerar várias linhas com uma única instrução echo, mas, por favor, limite-se a uma linha (ou menos) de código gerado por cada `echo()`.

> NOTA
> Além de `getString()`, também existem as funções `getBoolean()`, que tentará retornar o valor como um booleano lógico (adequado para uso em uma instrução `if()`), e `getList()`, que tentará retornar dados em formato de lista em um `Array()` do JavaScript. Mostraremos exemplos dessas funções mais adiante.
> Ao analisar os plugins existentes, você também encontrará muitos plugins usando `getValue()`, em vez de `getString()`, e na verdade os dois são *quase* idênticos. No entanto, usar `getString()`, `getBoolean()` e `getList()` é a prática recomendada desde a versão 0.6.1.

A situação fica um pouco mais complexa para o nível de confiança. Por razões estéticas, não queremos especificar explicitamente o nível de confiança a ser usado, caso ele corresponda ao valor padrão. Portanto, em vez de imprimir o valor incondicionalmente, primeiro o armazenamos em uma variável. Em seguida, verificamos se essa variável difere de `0,95` e, em caso afirmativo, imprimimos um argumento adicional. Finalmente, exibimos um colchete de fechamento e uma quebra de linha: `~) ~`. Isso é tudo para a função de cálculo.

5.1.3 printout()

```
function printout () {
        echo ('rk.header (' + i18n ("Teste t com duas variáveis") + ')\n');
        echo ('rk.print (res)\n');
}
```

E isso era tudo o que havia na função printout na maioria dos casos. `rk.header()` imprime um cabeçalho padrão para os resultados. Observe que nos arquivos `.js`, você precisa marcar todas as strings traduzíveis manualmente, usando **i18n()** ou alguns comandos alternativos. Mais sobre isso no capítulo sobre internacionalização. Você também pode adicionar mais informações a isso, se desejar, por exemplo:

```
function printout () {
        new Header (i18n ("Teste t com duas variáveis"))
                  .addFromUI ("varequal")
                  .add (i18n ("Nível de confiança"), getString ("conflevel ↩
                      ")) // Nota: escrito assim para fins ilustrativos. ↩
                      Mais automático:
        //        .addFromUI ("conflevel")
                  .print ();
echo ('rk.print (res)\n');
}
```

`rk.print()` utiliza o pacote R2HTML para fornecer saída formatada em HTML. Outra função útil é `rk.results()`, que também pode gerar diferentes tipos de tabelas de resultados. Em caso de dúvida, use `rk.print()` e pronto. A classe JS `Header` é uma função auxiliar em nível de JS para gerar uma chamada para `rk.header()` (basta dar uma olhada no código gerado). Em alguns casos, você pode querer chamar `echo ('rk.header(...)')` diretamente para imprimir um cabeçalho para sua saída.

Observe que, internamente, a saída é apenas um documento HTML simples neste momento. Portanto, você pode ser tentado a adicionar HTML personalizado usando `rk.cat.output()`. Embora isso funcione, por favor, não faça isso. O formato de saída pode mudar (por exemplo, para ODF) no futuro, então é melhor não introduzir código específico de HTML. Em vez disso, mantenha as coisas simples com `rk.header()`, `rk.print()`, `rk.results()` e -- se necessário -- `rk.print.literal()`. Se esses não atenderem às suas necessidades de formatação, entre em contato conosco na lista de discussão para obter ajuda.

Parabéns! Você criou seu primeiro plugin. Continue lendo nos próximos capítulos para conhecer conceitos mais avançados.

5.2 Convenções, políticas e contexto

Existem muitas maneiras de escrever código R para uma determinada tarefa, e existem ainda mais maneiras de gerar esse código R a partir de JS. Como exatamente você fará isso, fica a seu critério. Ainda assim, existem algumas considerações que você deve seguir e informações básicas que você deve compreender.

5.2.1 Compreendendo o ambiente `local()`

Na maioria das vezes, você precisará criar um ou mais objetos R temporários no código gerado pelo seu plugin. Normalmente, você não quer que eles sejam colocados no espaço de trabalho do usuário, podendo até mesmo sobrescrever variáveis do usuário. Portanto, todo o código gerado pelo plugin é executado em um ambiente `local()` (consulte a página de ajuda

do R sobre a função `local()`). Isso significa que todas as variáveis que você criar serão temporárias e não serão salvas permanentemente.

Se o usuário solicitar explicitamente que uma variável seja salva, você precisará atribuir a esse objeto usando `.GlobalEnv$objectname <- valor`. Em geral, não use o operador `<<-`. Ele não atribuirá necessariamente um valor em .GlobalEnv.

Uma armadilha importante é usar `eval()`. Aqui, você precisa observar que, por padrão, 'eval' usará o ambiente atual para avaliação, isto é, o ambiente local. Isso funcionará bem na maioria das vezes, mas nem sempre. Portanto, se você precisar usar `eval()`, provavelmente desejará especificar o parâmetro *`envir`*: `eval(..., envir=globalenv())`.

5.2.2 Formatação do código

O mais importante é que o seu código R gerado funcione, mas ele também deve ser fácil de ler. Portanto, preste atenção também à formatação. Algumas considerações:

As instruções R de nível superior normais devem ser alinhadas à esquerda.

As declarações em um bloco inferior devem ser recuadas com uma tabulação (veja o exemplo abaixo).

Se você fizer cálculos muito complexos, adicione comentários aqui e ali, especialmente para marcar seções lógicas. Observe que existe uma função dedicada **comment()** para inserir comentários traduzíveis no código gerado.

Por exemplo, o código gerado pode ter esta aparência. O mesmo código sem indentação ou comentários seria bastante difícil de ler, apesar de sua modesta complexidade:

```
# primeiro determine a oscilação e a rotação
my.wobble <- wobble (x, y)
my.rotation <- wobble.rotation (my.wobble, z)

# o método de seleção deve ser escolhido de acordo com a rotação
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 Lidar com opções complexas

Muitos plugins podem fazer mais de uma coisa. Por exemplo, o plugin 'Estatísticas Descritivas' pode calcular média, amplitude, soma, produto, mediana, comprimento, etc. No entanto, normalmente o usuário escolherá apenas realizar alguns desses cálculos. Nesse caso, tente manter o código gerado o mais simples possível. Ele deve conter apenas partes relevantes para as opções que foram selecionadas. Para conseguir isso, aqui está um exemplo de um padrão de design comum de como você o usaria (em JS; aqui, ″domean″, ″domedian″ e ″dosd″ seriam elementos <checkbox>):

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

        if (getBoolean ("domean.state")) echo ("results$" + i18n ("Valor  ↩
            médio") + " <- mean (x)\n");
```

```
        if (getBoolean ("domedian.state")) echo ("results$" + i18n (" ↩
            Mediana") + " <- median (x)\n");
        if (getBoolean ("dosd.state")) echo ("results$" + i18n ("Desvio  ↩
            padrão") + " <- sd (x)\n");
        //...
}
```

5.3 Dicas e truques

Aqui estão alguns truques variados que podem tornar a escrita de plugins menos tediosa:

Se você precisar do valor de uma configuração da GUIem vários lugares no código do seu plugin, considere atribuí-lo a uma variável em JS e usá-la em vez de buscá-lo repetidamente com `getString()/getBoolean()/getList()`. Isso é mais rápido, mais legível e requer menos digitação.

```
function calculate () {
        var narm = "";        // na.rm=FALSE é o padrão em todas as funções ↩
             abaixo
        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");
        // ...
}
```

A função auxiliar simples `makeOption()` pode facilitar a omissão de parâmetros que já possuem seus valores padrão, em muitos casos:

```
function calculate () {
        var options
        //...
        // Isso não fará nada se VALUE for 0,95 (o padrão). Caso contrário, ↩
             adicionará ', conf.int=VALUE' às opções.
        options += makeOption ("conf.int", getString ("confint"), "0.95");
        //...
}
```

Capítulo 6

Escrevendo uma página de ajuda

Quando seu plugin estiver basicamente funcionando, chegou a hora de fornecer uma página de ajuda. Embora normalmente você não queira explicar todos os conceitos subjacentes em detalhes, você pode querer adicionar mais explicações para algumas das opções, e links para plugins e funções relacionados.

> DICA
> Após ler este capítulo, dê uma olhada também no pacote rkwarddev. Ele fornece algumas funções R para criar a maioria das tags XML do RKWard para você. Ele também é capaz de criar estruturas básicas de arquivos de ajuda a partir de arquivos XML de plugins existentes para você começar.

Você deve se lembrar de ter colocado isso dentro do seu plugin XML (se você ainda não colocou isso, faça isso agora):

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

Onde, obviamente, você substituiria `nomearquivo` por um nome mais apropriado. Agora é hora de criar este arquivo `.rkh`. Aqui está um exemplo autoexplicativo:

```
<!DOCTYPE rkhelp>
<document>
        <summary>
Nesta seção, você colocará algumas informações básicas e resumidas sobre o ↩
    que o plugin faz. Esta seção sempre aparecerá no topo da página de ajuda ↩
    .
        </summary>

        <usage>
A seção de uso pode conter informações um pouco mais práticas. Ela não ↩
    explica todas as configurações
em detalhes (isso é feito na seção "configurações").

Para iniciar um novo parágrafo, insira uma linha em branco, como mostrado ↩
    acima.
Esta linha, por outro lado, estará no mesmo parágrafo.
```

```
Em todas as seções, você pode inserir um código HTML simples, como <b> negrito<b> ou
texto <i>itálico<i>. Mantenha a formatação ao mínimo necessário.

A seção de uso é sempre a segunda seção exibida em uma página de ajuda.
        </usage>

        <section id="sectionid" title="Seção genérica" short_title="Genérica">
Se necessário, você pode adicionar seções adicionais entre as seções de uso e configurações.
No entanto, geralmente, você não precisará disso ao documentar plugins. O atributo "id"
fornece um ponto de ancoragem para acessar esta seção a partir do menu de navegação.
O atributo "short_title"
fornece um título curto para usar na barra de navegação. Isso é opcional. Por padrão,
o "title" principal será usado como cabeçalho da seção e como nome do link na
barra de navegação.

Em qualquer seção, você pode inserir links para mais informações. Para isso, adicione:

<link href="URL">nome do link</link>

Onde URL pode ser um link externo, como https://rkward.kde.org.

Vários URLs especiais são suportados nas páginas de ajuda:

<link href="rkward://pagina/caminho/id_pagina"/>

Este link leva a uma página de ajuda do rkward de nível superior (não para um plugin).

<link href="rkward://componente/[namespace/]id_componente"/>

Este link leva à página de ajuda de outro plugin. A parte [namespace/] pode ser omitida.
(neste caso, assume-se que rkward é o namespace padrão, por exemplo:
<link href="rkward://componente/import_spss"/> or
<link href="rkward://componente/rkward/import_spss"/> são equivalentes).
O id_componente é o mesmo que você especificou no .pluginmap.

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

Links para a página de ajuda do R sobre "rfunction".

Observe que os nomes dos links serão gerados automaticamente para esses tipos de links.
        </section>

        <settings>
                <caption id="id_da_aba_ou_quadro"/>
                <setting id="id_do_elemento">
Descrição do elemento da GUI identificado pelo ID fornecido
                </setting>
```

```
                <setting id="id_do_elementob" title="descrição">
Normalmente, o título do elemento da GUI será extraído da
definição XML do plugin automaticamente. Entretanto,
para alguns elementos da interface gráfica, esta descrição pode não ser  ↩
    suficiente para identificá-los de forma confiável.
Nesse caso, você pode adicionar um título explícito usando o atributo " ↩
    title".
                </setting>
                <setting id="id_do_elementoc">
Descrição do elemento da interface gráfica identificado por " ↩
    id_do_elementoc"
                </setting>
                [...]
        </settings>

        <related>
A seção "related" geralmente contém apenas alguns links, como:

<ul>
        <li><link href="rkward://rhelp/meam"/></li>
        <li><link href="rkward://rhelp/median"/></li>
        <li><link href="rkward://componente/compontente_relacionado"/></li>
</ul>
        </related>

<technical>
A seção técnica (opcional, sempre a última) pode conter alguns detalhes  ↩
    técnicos da implementação do plugin,
que são de interesse apenas para os desenvolvedores do RKWard. Isso é  ↩
    particularmente relevante
para plugins projetados para serem incorporados em muitos outros plugins e  ↩
    pode detalhar quais
opções estão disponíveis para personalizar o plugin incorporado e quais  ↩
    seções de código contêm qual
código R.
        </technical>
</document>
```

Capítulo 7

Interações lógicas entre elementos da GUI

7.1 Lógica da GUI

Todos os conceitos básicos para a criação de um plugin para o RKWard foram descritos nos capítulos anteriores. Esses conceitos básicos devem ser suficientes para muitos casos, senão para a maioria. No entanto, às vezes você deseja mais controle sobre o comportamento da GUI do seu plugin.

Por exemplo, suponha que você queira estender o exemplo de teste t usado nesta documentação para permitir ambas as comparações: comparar uma variável com outra variável (como mostrado) e comparar uma variável com um valor constante. Uma maneira de fazer isso seria adicionar um botão de opção que alterna entre os dois modos e adicionar uma caixa de seleção para inserir o valor constante a ser comparado. Considere este exemplo simplificado:

```
<!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="comparar"/>
                                <radio id="mode" label="Comparar contra">
                                        <option value="variable" checked=" ↩
                                            true" label="outra variável ( ↩
                                            selecionar abaixo)"/>
                                        <option value="constant" label="um  ↩
                                            valor constante (definir abaixo) ↩
                                            "/>
                                </radio>
                                <varslot id="y" types="number" source="vars ↩
                                    " required="true" label="variable"  ↩
                                    i18n_context="Nome; uma variável"/>
                                <spinbox id="constant" initial="0" label=" ↩
                                    constant" i18n_context="Nome; uma  ↩
                                    constante"/>
                        </column>
```

```
                </row>
        </dialog>
</document>
```

Até aqui tudo bem, mas existem alguns problemas com esta GUI. Primeiro, tanto o varslot quanto o spinbox são sempre exibidos, embora apenas um dos dois seja realmente usado. Pior ainda, o varslot sempre requer uma seleção válida, mesmo se você comparar com uma constante. Obviamente, se criarmos uma GUI multiuso como esta, queremos mais flexibilidade. Insira: a seção **<logic>** (inserida no mesmo nível que **<code>**, **<dialog>** ou **<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="Teste t">
        [...]
```

A primeira linha dentro da seção de lógica é uma tag **<convert>**. Basicamente, isso fornece uma nova propriedade booleana (ligado ou desligado, verdadeiro ou falso), que pode ser usada posteriormente. Essa propriedade (*˜varmode˜*) é verdadeira sempre que o botão de opção superior estiver selecionado e falsa sempre que o botão de opção inferior estiver selecionado. Como isso é feito?

Primeiro, em *sources*, as propriedades de origem para trabalhar são listadas (neste caso, apenas uma de cada; você poderia listar várias como *sources=˜mode.string;somethingelse˜*, então *˜varmode˜* seria verdadeiro apenas se *˜mode.string˜* e *˜somethingelse˜* forem iguais à string *˜variable˜*). Observe que, neste caso, não escrevemos apenas *˜mode˜* (como faríamos em `getString(˜mode˜)`), mas *˜mode.string˜*. Na verdade, este é o funcionamento interno de um controle remoto: Ele possui uma propriedade 'string', que armazena seu valor de string. 'getString(˝mode˝)' é apenas uma abreviação e equivalente a 'getString(˝mode.string˝)'. Consulte a referência para ver todas as propriedades dos diferentes elementos da GUI.

Em segundo lugar, definimos o modo de conversão para *mode=˜equals˜*. Isso significa que queremos verificar se a(s) fonte(s) é(são) igual(is) a um determinado valor. Finalmente, o valor padrão é aquele com o qual comparar. Portanto, com *standard=˜variable˜*, verificamos se a propriedade *˜mode.string˜* é igual à string *˜variable˜* (o valor da opção de rádio superior). Se for igual, a propriedade varmode será verdadeira; caso contrário, será falsa.

Agora vamos ao que interessa: Nós **<conect>** (conectamos) a propriedade *˜varmode˜* à propriedade y.visible, que controla se o elemento *˜y˜* do varslot é exibido ou não. Observe que qualquer elemento que se torne invisível é implicitamente não obrigatório. Portanto, se a opção de rádio superior estiver selecionada, o elemento *˜y˜* do varslot será obrigatório e visível. Caso contrário, ele não será obrigatório e permanecerá oculto.

Para o spinbox, queremos exatamente o inverso. Felizmente, não precisamos de outro **<convert>** para isso: propriedades booleanas podem ser negadas facilmente adicionando o modificador *˜not˜*, então **<connect>** *˜varmode.not˜* à propriedade visibility do spinbox. Na prática, ou o varslot é exibido e obrigatório, *ou* o spinbox é exibido e obrigatório - dependendo da opção selecionada no botão de opção. A GUI se altera de acordo com a opção do botão de opção. Experimente o exemplo, se quiser.

Para obter uma lista completa de propriedades, consulte a referência. No entanto, existe mais uma propriedade especial que todos os elementos GUI possuem: 'enabled'. Esta é ligeiramente

menos drástica do que 'visible'. Ela não mostra/oculta o elemento GUI mas apenas o ativa/desativa. Elementos desativados são normalmente exibidos em cinza e não reagem à entrada do usuário.

> NOTA
> Além de **<convert>** e **<connect>**, existem vários outros elementos para uso na seção **<logic>**. Por exemplo, construções condicionais também podem ser implementadas usando o elemento **<switch>**. Consulte a referência sobre elementos lógicos para obter detalhes.

7.2 Lógica da GUI com script

Embora conectar propriedades conforme descrito acima seja frequentemente suficiente, às vezes é mais flexível ou conveniente usar JS para programar a lógica da GUI. Dessa forma, o exemplo acima poderia ser reescrito como:

```
[...]
        <code file="code.js"/>
'
        <logic>
                <script><![CDATA[
                        // [...] qualquer código no nível superior é  ↩
                            chamado apenas uma vez.
                        gui.addChangeCommand("mode.string", function() {
                                // enquanto esta função anônima será  ↩
                                    chamada sempre que "mode.string" mudar
                                var varmode = (gui.getString("mode.string") ↩
                                     == "variable");
                                gui.setValue("y.enabled", varmode);
                                gui.setValue("constant.enabled", !varmode);
                        });
                ]]></script>
        </logic>

        <dialog label="Test t">
        [...]
```

Isso registra uma função anônima a ser chamada sempre que o valor da caixa de seleção *id="mode"* mudar. Dentro dessa função, definimos uma variável auxiliar *"varmode"* que é verdadeira quando o modo é *"variable"* e falsa quando é *"constant"*. Em seguida, usamos `gui.setValue()` para definir as propriedades 'enabled' de *"y"* e *"constant"*, da mesma forma que fizemos usando as instruções **<connect>** anteriormente.

> NOTA
> Se a mesma função precisar ser invocada para alterações em vários elementos, você também pode passar um array dos respectivos *id*=s para **gui.addChangeCommand()**. Além disso, para maior conveniência, **gui.addChangeCommand()** retorna seu segundo parâmetro, o que é útil se você quiser se referir a essa função em outro lugar (por exemplo, para **gui.addChangeCommand()** chamá-la uma vez durante a inicialização). Exemplo:
>
> ```
> let update = gui.addChangeCommand(["mode.string", "y.available"],  ↩
>     function() {
>                 // faz algo a cada mudança de modo.string ou y. ↩
>                     disponível
>         });
>         update(); // faz o mesmo durante a inicialização
> ```

A abordagem com scripts para lógica da GUI torna-se particularmente útil quando você deseja alterar a opção disponível de acordo com o tipo de objeto que o usuário selecionou. Consulte a referência para obter as funções disponíveis.

Observe que a abordagem com script para a lógica da GUI pode ser combinada com instruções **<connect>** e **<convert>** se desejar. Observe também que a tag **<script>** permite especificar um nome de arquivo de script, além de ou como alternativa à inclusão do código de script. Normalmente, a inclusão do código de script, como mostrado acima, é mais conveniente.

Capítulo 8

Embutindo plugins dentro de plugins

8.1 Casos de uso para embutir

Ao escrever plugins, você frequentemente perceberá que está criando vários plugins que diferem apenas em alguns aspectos, mas têm muito mais em comum. Por exemplo, para plotagem, existem várias opções R genéricas que podem ser usadas com praticamente todos os tipos de gráficos. Você deveria criar uma interface gráfica e um modelo em JavaScript para essas opções repetidamente?

Obviamente, isso seria bastante trabalhoso. Felizmente, você não precisa fazer isso. Em vez disso, você cria a funcionalidade comum uma vez e, posteriormente, pode incorporá-la em vários plugins. Na verdade, é possível incorporar qualquer plugin em qualquer outro plugin, mesmo que o autor original do plugin incorporado nunca tenha imaginado que alguém desejaria incorporar seu plugin em outro.

8.2 Incorporar dentro de uma caixa de diálogo

Ok, chega de conversa. Como funciona? Simples: basta usar a tag **<embed>**. Aqui está um exemplo simplificado:

```
<dialog>
        <tabbook>
                <tab [...]>
                        [...]
                </tab>
                <tab label="Opções de plotagem" i18n_context="Opções ↩
                    relativas à plotagem">
                        <embed id="plotoptions" component="rkward:: ↩
                            plot_options"/>
                </tab>
                <tab [...]>
                        [...]
                </tab>
        </tabbook>
</dialog>
```

O que acontece aqui é que toda a GUI ou o plugin de opções de plotagem (exceto, é claro, os elementos padrão como o botão **Enviar**, etc.) é incorporado diretamente ao seu plugin (experimente!).

Como você pode ver, a sintaxe da tag **<embed>** é bastante simples. Ela recebe um *id* como a maioria dos elementos. O componente especifica qual plugin incorporar, conforme definido no arquivo `.pluginmap` (*˜rkward::plot_options˜* é o resultado da concatenação do namespace 'rkward', um separador '::' e o nome do componente 'plot_options').

8.3 Geração de código ao incorporar

Até aqui tudo bem, mas e o código gerado? Como o código para o plugin de incorporação e o plugin incorporado são mesclados? No código JS do plugin de incorporação, basta escrever algo como:

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

Basicamente, estamos buscando o código gerado pelo plugin incorporado assim como buscamos qualquer outra configuração da GUI. Aqui, a string *˜plotoptions.code.printout˜* pode ser analisada para: 'A seção de impressão do código gerado do elemento com o *id* plotoptions' (plotoptions é o ID que demos para a tag **<embed>** acima). E sim, se você quiser controle avançado, pode até buscar os valores de elementos individuais da GUI dentro do plugin incorporado (mas não o contrário, pois o plugin incorporado não sabe nada sobre o que está ao seu redor).

8.4 Incorporação dentro de um assistente

Se o seu plugin fornecer um GUI de assistente, a incorporação funciona basicamente da mesma maneira. Geralmente você usará:

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

Se o plugin incorporado fornecer uma interface de assistente, suas páginas serão inseridas diretamente entre *˜page12˜* e *˜page13˜* do seu plugin. Se o plugin incorporado fornecer apenas uma interface de diálogo, uma única nova página será adicionada entre suas páginas *˜page12˜* e *˜page13˜*. O usuário nunca perceberá.

8.5 Menos incorporação embutida: Botão de Opções adicionais

Embora incorporar elementos seja interessante, tenha cuidado para não exagerar. Muitas funções dentro de uma GUI dificultam a localização das opções relevantes. É claro que, às vezes, você pode querer incorporar muitas opções (como todas as opções de `plot()`), mas como elas são realmente opcionais, você não quer que elas fiquem em destaque na sua GUI.

Uma alternativa é incorporar essas opções 'como um botão':

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

Neste caso, um único botão será adicionado ao seu plugin, rotulado como **Especificar opções de plotagem**. Ao pressionar esse botão, uma caixa de diálogo separada será exibida, com todas as opções do plugin incorporado. Mesmo que esta GUI incorporada não esteja visível na maior parte do tempo, você pode obter suas configurações conforme descrito acima.

> CUIDADO
> Provavelmente, a abordagem do 'botão' só deve ser usada para plugins que nunca podem ser inválidos (por configurações ausentes/incorretas). Caso contrário, o usuário não conseguiria enviar o código, mas poderia ter dificuldade em descobrir o motivo, que está oculto atrás de algum botão.

8.6 Incorporação/definição de plugins incompletos

Alguns plugins — e, aliás, o plot_options usado como exemplo acima é um deles — não são completos por si só. Eles simplesmente não têm os elementos de GUI para selecionar alguns valores importantes. Eles foram projetados para serem usados apenas incorporados a outros plugins.

Até que ponto o plugin plot_options está incompleto? Bem, para algumas configurações de opções, ele precisa saber o nome dos objetos/expressões para os eixos x e y (na verdade, funcionará bem se tiver apenas um deles, mas precisa de pelo menos um para funcionar corretamente). No entanto, ele não possui um mecanismo para selecionar esses objetos ou inseri-los de qualquer outra forma. Então, como ele sabe sobre eles?

Na seção de lógica do plugin plot_options, existem duas linhas adicionais, ainda não abordadas:

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

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

Isso define duas propriedades adicionais no plugin plot_options, cujo único propósito é ser conectado a algumas propriedades (ainda desconhecidas) do plugin de incorporação. No plugin plot_options, essas duas propriedades são simplesmente usadas como quaisquer outras e, por exemplo, existem chamadas para `getString(˜xvar˜)` no modelo JS do plot_options.

Agora, para o plugin incompleto, não há como saber onde ele será incorporado e quais serão os nomes das configurações relevantes no plugin de incorporação. Portanto, precisamos adicionar duas linhas adicionais na seção de lógica do plugin de incorporação:

```
<logic>
                [...]

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

Em princípio, isso não é novidade; já abordamos as instruções **<connect>** no capítulo da seção de lógica da GUI. Basta conectar os valores em dois varlots (chamados *˜xvarslot˜* e *˜yvarslot˜* neste exemplo) às propriedades 'external' receptoras do plugin incorporado. É só isso. Todo o resto é tratado automaticamente.

Capítulo 9

Lidar com muitos plugins semelhantes

9.1 Visão geral das diferentes abordagens

Às vezes, você pode querer desenvolver plugins para uma série de funções semelhantes. Por exemplo, considere os gráficos de distribuição. Eles geram código bastante similar e, claro, é desejável que as interfaces gráficas sejam semelhantes entre si. Finalmente, grandes seções dos arquivos de ajuda podem ser idênticas. Apenas alguns parâmetros são diferentes para cada plugin.

A abordagem ingênua para isso é desenvolver um plugin e, basicamente, copiar e colar todo o conteúdo dos arquivos `.js`, `.xml` e `.rkh`, alterando apenas as poucas partes que são diferentes. No entanto, e se, algum tempo depois, você encontrar um erro de ortografia que foi copiado e colado em todos os plugins? E se você quiser adicionar suporte para um novo recurso? Você teria que visitar todos os plugins novamente e alterar cada um deles. Um processo cansativo e tedioso.

Uma segunda abordagem seria usar incorporação. No entanto, em alguns casos, isso não se presta bem ao problema em questão, principalmente porque os 'pedaços' que você pode incorporar são às vezes muito grandes para serem úteis, e isso impõe algumas restrições ao layout. Para esses casos, os conceitos de incluir arquivos `.js`, incluir arquivos `.xml` e snippets podem ser muito úteis (mas veja as considerações sobre quando é preferível usar incorporação).

Uma palavra de cautela, antes de começar a ler: esses conceitos podem simplificar o gerenciamento de muitos plugins semelhantes e melhorar a manutenção e a legibilidade desses plugins. No entanto, exagerar pode facilmente levar ao efeito contrário. Use com cautela.

9.2 Usando a instrução include do JavaScript

Você pode facilmente incluir um arquivo de script em outro em plugins do RKWard. O valor disso se torna imediatamente óbvio se algumas seções do seu código JS forem semelhantes entre os plugins. Você pode simplesmente definir essas seções em um arquivo `.js` separado e incluí-lo em todos os arquivos `.js` do plugin. Por exemplo, como em:

```
// este é um arquivo chamado "common_functions.js"

function doCommonStuff () {
        // talvez recebe algumas opções, etc.
        // ...
```

```
        comment ("Este é o código R que você deseja incluir em diversos  ↩
            plugins\n");
        // ...
}
```

```
// este é o do seu dos seus arquivos .js do plugin normal

// inclui as funções comuns
include ("common_functions.js");

function calculate () {
        // faz alguma coisa
        // ...

        // inserir o código comum
        doCommonStuff ();
}
```

Observe que, às vezes, é ainda mais útil inverter isso e definir o 'esqueleto' das funções `preprocess()`, `calculate()` e `printout()` em um arquivo comum, e fazer com que essas funções chamem de volta as partes que são diferentes entre os plugins. Por exemplo:

```
// este é um arquivo chamado "common_functions.js"

function calculate () {
        // faz algumas coisas que são o mesmo para todos os plugins
        // ...

        // adiciona algo que é diferente entre os plugins
        getSpecifics ();

        // ...
}
```

```
// este é o do seu dos seus arquivos .js do plugin normal

// inclui as funções comuns
include ("common_functions.js");

// nota: nenhuma função calculate() é definida aqui.
// está em common_functions.js, ao invés.

function getSpecifics () {
        // imprime algum código R
}
```

Uma questão que você deve ter em mente ao usar essa técnica é o escopo de variáveis. Consulte o manual do JavaScript sobre escopos de variáveis.

Essa técnica é muito utilizada nos plugins de plotagem de distribuição e de distribuição CLT, então talvez você queira procurar exemplos lá.

9.3 Incluir arquivos `.xml`

Basicamente, o mesmo recurso de inclusão de arquivos também está disponível para uso nos arquivos `.xml`, `.pluginmap` e `.rkh`. Em qualquer lugar nesses arquivos, você pode inserir uma tag

<include>, como mostrado abaixo. O efeito é que todo o conteúdo desse arquivo XML (para ser preciso: tudo dentro da tag **<document>** desse arquivo) é incluído literalmente neste ponto do arquivo. Observe que você só pode incluir outro arquivo XML.

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

O atributo *file* é o nome do arquivo relativo ao diretório em que o arquivo atual está localizado.

9.4 Usar <snippets>

Embora a inclusão de arquivos conforme mostrado na seção anterior seja bastante poderosa, ela se torna mais útil quando usada em combinação com o comando **<snippets>**. Snippets são, na verdade, seções menores que você pode inserir em outro ponto do arquivo. Um exemplo ilustra isso melhor:

```
<document>
        <snippets>
                <snippet id="note">
                        <frame>
                                <text>
                Isto será inserido em dois lugares na GUI
                                </text>
                        </frame>
                </snippet>
        </snippets>
        <dialog label="test">
                <column>
                        <insert snippet="note"/>
                        [...]
                        <insert snippet="note"/>
                </column>
        </dialog>
</document>
```

Portanto, você define o trecho ou snippet em um local no topo do arquivo XML e então você o **<insert>** (insere) em qualquer(is) local(is) que desejar.

Embora este exemplo não seja muito útil por si só, considere combiná-lo com um arquivo `.xml` usando **<include>**. Observe que você também pode inserir trechos de código para o arquivo `.rkh` no mesmo arquivo. Basta adicionar o arquivo lá também usando **<include>** e inserir o trecho de código relevante usando **<insert>**:

```
<!-- Este é o arquivo chamado "common_snippets.xml" -->
<document>
        <snippet id="common_options">
                <spinbox id="something" [...]/>
                [...]
        </snippet>
        <snippet id="common_note">
                <text>Uma observação importante para este tipo de plugin</ ↩
                    text>
        </snippet>
```

```
        <snippet id="common_help">
                <setting id="something">Isto faz alguma coisa</setting>
                [...]
        </snippet>
</document>
```

```
<!-- Este é o arquivo .xml do plugin -->
<document>
        <snippets>
                <!-- Importa os snippets comuns-->
                <include file="common_snippets.xml"/>
        </snippets>

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

Semelhante à inclusão em JS, a abordagem inversa costuma ser ainda mais útil:

```
<!-- Este é um arquivo chamado "common_layout.xml" -->
<document>
        <column>
                <insert snippet="note">
                [...]
                <insert snippet="plugin_parameters">
        </column>
        [...]
</document>
```

```
<!-- Este é o arquivo .xml do plugin -->
<document>
        <snippets>
                <snippet id="note">
                        <text>A nota usada para este plugin específico</ ↩
                            text>
                </snippet>

                <snippet id="plugin_parameters">
                        <frame label="Parâmetros específicos deste plugin">
                                [...]
                        </frame>
                </snippet>
        </snippets>

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

Finalmente, também é possível **<insert>** (inserir) trechos de código (snippets) em outros trechos, desde que: a) haja apenas um nível de aninhamento, e b) a seção de **<snippets>** esteja localizada no topo do arquivo (antes da inserção de um snippet aninhado); isso ocorre porque as instruções **<insert>** são resolvidas de cima para baixo.

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

À primeira vista, **<include>** e **<snippets>** oferecem funcionalidades bastante semelhantes a embedding: Permitem reutilizar partes do código entre plugins. Então, qual é a diferença entre essas abordagens e quando usar cada uma?

A principal diferença entre esses conceitos é que os plugins incorporáveis são um pacote mais compacto. Eles combinam uma GUI completa, código para gerar código R a partir dela e uma página de ajuda. Em contraste, os métodos include e insert permitem um controle muito mais preciso, mas com menor modularidade.

Ou seja, um plugin que incorpora outro plugin normalmente não precisa saber muito sobre os detalhes internos do plugin incorporado. Um ótimo exemplo é o plugin plot_options. Plugins que desejam incorporá-lo não precisam necessariamente conhecer todas as opções fornecidas, ou como elas são fornecidas. Isso é bom, pois, caso contrário, uma alteração no plugin plot_options poderia tornar necessário ajustar todos os plugins que o incorporam (e muito). Em contraste, os plugins include e insert expõem todos os detalhes internos, e os plugins que os utilizam precisarão -- por exemplo -- saber os IDs exatos e talvez até mesmo o tipo dos elementos usados.

Portanto, a regra prática é a seguinte: incluir e inserir são ótimos se as opções relevantes forem necessárias apenas para um grupo claramente limitado de plugins. Plugins incorporados são melhores se o grupo de plugins para os quais eles podem ser úteis não estiver claramente definido e se a funcionalidade puder ser facilmente modularizada. Outra regra prática: se você puder colocar as partes comuns em um único 'bloco', faça isso e use a incorporação. Se você precisar de muitos pequenos trechos para definir as partes comuns, use **<snippets>**. Uma última maneira de ver isso: se todos os plugins fornecerem funcionalidades muito semelhantes, incluir e inserir provavelmente é uma boa ideia. Se eles compartilharem apenas um ou dois 'módulos' em comum, a incorporação provavelmente é melhor.

Capítulo 10

Conceitos para uso em plugins especializados

Este capítulo contém informações sobre alguns tópicos que são úteis apenas para certas classes de plugins.

10.1 Plugins que geram um gráfico

Criar um gráfico a partir de um plugin é fácil. No entanto, existem algumas armadilhas sutis a evitar, bem como algumas funcionalidades genéricas excelentes que você deve conhecer. Esta seção mostra os conceitos básicos e conclui com um exemplo canônico que você deve seguir sempre que criar plugins de gráficos.

10.1.1 Desenhar um gráfico na janela de saída

Para desenhar um gráfico na janela de saída, use `rk.graph.on()` imediatamente antes de criar o gráfico e `rk.graph.off()` imediatamente depois. Isso é semelhante a, por exemplo, chamar `postscript()` e `dev.off()` em uma sessão R regular.

É importante ressaltar, no entanto, que você deve *sempre* chamar `rk.graph.off()` após chamar `rk.graph.on()`. Caso contrário, o arquivo de saída ficará corrompido. Para garantir que `rk.graph.off()` seja realmente chamado, você deve envolver todos os comandos R entre as duas chamadas em uma instrução `try()`. Nunca ouviu falar disso? Não se preocupe, é fácil. Tudo o que você precisa fazer é seguir o padrão mostrado no exemplo, abaixo.

10.1.2 Adicionando funcionalidade de pré-visualização

> NOTA
> Esta seção discute a adição de funcionalidades de pré-visualização a plugins que produzem gráficos. Existem seções separadas sobre pré-visualizações de saída (HTML), pré-visualizações de dados (importados) e pré-visualizações personalizadas. No entanto, recomenda-se que você leia esta seção primeiro, pois a abordagem é semelhante em todos os casos.

Um recurso muito útil para todos os plugins que geram um gráfico é fornecer uma pré-visualização com atualização automática. Para isso, você precisará de duas coisas: Adicionar uma caixa de seleção **<preview>** à sua definição GUI e ajustar o código gerado para a pré-visualização.

Adicionar uma caixa de seleção **<preview>** é simples. Basta colocar o seguinte em algum lugar da sua GUI. Isso cuidará de toda a mágica nos bastidores da criação de um dispositivo de visualização, atualizando a visualização sempre que as configurações forem alteradas, etc. Exemplo:

> NOTA
> A partir da versão 0.6.5 do RKWard, elementos **<preview>** são tratados de forma especial em diálogos de plugins (não em assistentes): eles serão colocados na coluna de botões, independentemente de onde estejam definidos na interface do usuário. Ainda é uma boa prática defini-los em um local adequado no layout, para garantir a compatibilidade com versões anteriores.

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

E isso conclui a definição da GUI.

Ajustar o modelo JS requer apenas um pouco mais de trabalho. Aqui, você precisará garantir que somente o próprio gráfico seja gerado e exibido em um dispositivo na tela, em vez de ser direcionado para a saída. Ou seja, sem impressão de cabeçalhos, `rk.graphics.on()` ou chamadas semelhantes. Para auxiliar você nisso, o RKWard chamará as funções `preprocess()`, `calculate()` e `printout()` com um parâmetro adicional definido como *`true`* ao gerar o código para uma pré-visualização. (O parâmetro é omitido ao gerar o código final. Em JavaScript, isso será avaliado como *`false`* quando usado dentro de uma instrução `if`.) Veja o exemplo abaixo para o padrão típico que você usará.

Alternativamente, caso precise de mais controle do que isso, você pode adicionar uma nova função chamada `preview()` ao seu modelo JS e gerar o código necessário para uma pré-visualização ali (provavelmente, pelo menos em parte, chamando novamente `calculate()`, etc).

10.1.3 Opções genéricas do gráfico

Você deve ter notado que a maioria dos plugins de plotagem no RKWard oferece uma ampla gama de opções genéricas como, por exemplo, personalizar títulos de eixos ou margens de figuras. Adicionar essas opções ao seu plugin é fácil. Elas são fornecidas por um plugin incorporável chamado **rkward::plot_options**. Incorpore-o na interface do seu plugin desta forma:

```
<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>
```

Isso adicionará um botão à sua interface de usuário para abrir uma janela com opções de plotagem. A seção de lógica é apenas um exemplo. Ela permite algum controle sobre o plugin de opções de plotagem. Leia mais na página de ajuda do plugin plot_options (link disponível na página de ajuda de qualquer plugin que forneça as opções genéricas).

Em seguida, você precisa garantir que o código correspondente às suas opções de plotagem seja adicionado ao código gerado para o seu gráfico. Para fazer isso, obtenha as propriedades **code.preprocess**, **code.printout** e **code.calculate** do plugin de opções de plotagem incorporado e insira-as em seu código, conforme mostrado no exemplo, abaixo.

10.1.4 Um exemplo canônico

Aqui está um exemplo de arquivo .JS que você deve usar como modelo sempre que criar um plugin de plotagem:

```
function preprocess () {
    // o "algumpacote" é necessário para criar o gráfico
    echo ("require (algumpacote)\n");
  }

  function printout (is_preview) {
    // Se "is_preview" é definido para false/undefined, ele gera o código  ↩
        completo, incluindo headers.
    // Se "is_preview" é definido para true, somente o essencial será  ↩
        gerado.

    if (!is_preview) {
      echo ('rk.header (' + i18n ("An example plot") + ')\n\n');
      echo ('rk.graph.on ()\n');
    }
    // somente a seção a seguir será gerada para is_preview==true

    // lembre-se: tudo entre between rk.graph.on() e rk.graph.off() deve  ↩
        ser circundado dentro de uma declaração try():
    echo ('try ({\n');
    // insira qualquer código de opção-configuração que deve ser executado  ↩
        antes dos comandos de plotagem.
    // O código em si é fornecido para incorporar o plugin de opções de  ↩
        gráfico. printIndentedUnlessEmpty() takes care of pretty formatting.
    printIndentedUnlessEmpty ('\t', getString ("plotoptions.code.preprocess ↩
        "), '', '\n');

    // cria de fato o gráfico. plotoptions.code.printout fornece a parte  ↩
        das opções genéricas do gráfico
    // que deve ser adicionado à chamada de plotagem em si.
    echo ('plot (5, 5' + getString ("plotoptions.code.printout") + ')\n');
```

```
    // insere qualquer código de configuração-opção que deve ser executado ↩
        após a plotagem em si.
    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 Pré-visualizações de dados, resultados e outras informações

10.2.1 Pré-visualizações de saída (HTML)

> NOTA
> Esta seção discute a adição de funcionalidade de pré-visualização a plugins que criam impressões de saída/HTML. Recomenda-se que você leia a seção separada sobre pré-visualizações de gráficos, antes desta seção.

Criar uma pré-visualização da saída HTML é um procedimento quase idêntico ao de criar uma pré-visualização de um gráfico. Nesse caso, basta garantir que o comando **preview()** gere os comandos relevantes **rk.print()/rk.results()**. No entanto, geralmente é uma boa prática omitir as declarações de cabeçalho na pré-visualização. Aqui está um exemplo simplificado:

```
<!-- In the plugin's XML file -->>
        <dialog label="Importar dados CSV" >
                <browser id="file" type="file" label="Nome do arquivo"/>
                <!-- [...] -->>
                <preview id="preview" mode="output"/>
        </dialog>
>
```

Observe a especificação de *mode="output"* no elemento **<preview>**.

```
// No arquivo JS do plugin
        function preview () {
                // gera o código usado para a pré-visualização
                printout (true);
        }

        function printout (is_preview) {
                // somente gera um header se is_preview==false
                if (!is_preview) {
                        new Header ("Esta é uma descrição").print ();
                }
                echo ('rk.print (result)');
        }
```

10.2.2 Pré-visualização dos dados (importados)

> NOTA
> Esta seção discute a adição de funcionalidade de pré-visualização a plugins que criam (ou importam) dados. Recomenda-se que você leia a seção separada sobre pré-visualizações de gráficos antes desta seção.

Criar uma pré-visualização de dados importados (qualquer tipo de dado que o comando **rk.edit()** possa manipular) é muito semelhante a criar uma pré-visualização de gráfico. O exemplo simplificado a seguir deve ajudar a ilustrar como criar uma pré-visualização de dados:

```
<!-- No arquivo XML do plugin -->>
        <dialog label="Importar dados CSV" >
                <browser id="file" type="file" label="Nome do arquivo"/>
                <!-- [...] -->>
                <preview id="preview" active="true" mode="data"/>
        </dialog>
>
```

Observe que o elemento **<preview>** especifica *mode="data"* desta vez. *active="true"* simplesmente torna a pré-visualização ativa por padrão.

```
// No arquivo JS do plugin
        function preview () {
                // gera o código usado para a pré-visualização
                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 () {
                // [...]
        }
```

Novamente, a função **preview()** gera quase o mesmo código R que a função **calculate()**, então criamos uma função auxiliar **doCalculate()** para isolar as partes comuns. O mais importante a observar é que você precisará atribuir os dados importados a um objeto chamado *preview_data* (dentro do ambiente local atual). *Todo o resto acontecerá automaticamente* (grosso modo, o RKWard chamará **rk.edit(preview_data)**, dentro de uma chamada para **.rk.with.window.hints()**).

NOTA
Embora as pré-visualizações sejam um ótimo recurso, elas consomem recursos. No caso de pré-visualizações de dados, pode haver situações em que elas causem problemas de desempenho significativos. Isso pode ocorrer ao importar conjuntos de dados enormes (que são grandes demais para serem abertos para edição na janela do editor do RKWard), mas também conjuntos de dados ″normais″ podem ser importados incorretamente, criando um número enorme de linhas ou colunas. *É altamente recomendável limitar o parâmetro preview_data* a uma dimensão que forneça uma pré-visualização útil, sem o risco de criar problemas de desempenho perceptíveis (por exemplo, 50 linhas por 50 colunas devem ser mais do que suficientes na maioria dos casos).

10.2.3 Pré-visualizações personalizadas

O elemento **<preview>** pode ser usado para criar pré-visualizações para qualquer tipo de janela de ″documento″ que possa ser anexada à área de trabalho do RKWard. Além de plots e data windows, isso inclui arquivos HTML, scripts R e janelas de resumo de objetos. Para estes últimos, você precisará usar **<preview mode=″custom″>** .

Se você leu as seções que descrevem a pré-visualização de gráficos e a pré-visualização de dados, você já deve ter uma ideia geral do procedimento, mas as pré-visualizações ″personalizadas″ exigem um pouco mais de trabalho manual nos bastidores. A função R mais importante a ser analisada é **rk.assign.preview.data()**, neste exemplo. A seguinte listagem resumida mostra como o seu código R (pré-visualização) gerado poderia ser para um plugin que cria um arquivo de texto como saída:

```
## A ser gerado na seção de código do preview() de um plugin
        pdata <- rk.get.preview.data("ALGUMAID")
        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)$filename)
                })
                rk.assign.preview.data("ALGUMAID", pdata)
        }
        try ({
                cat ("Isto é um teste", pdata$filename)
                rk.edit.files(file=pdata$filename)
        })
```

Aqui você deve obter o valor *ALGUMAID* da propriedade *id* do elemento **<preview>**, ou seja, usando **getString (″preview.id″)** no arquivo .js do plugin.

10.3 Plugins dependentes do contexto

Até agora, presumimos que todos os plugins são sempre relevantes e estão localizados no menu principal. No entanto, alguns plugins só fazem sentido (ou adicionalmente) em um determinado contexto. Por exemplo, um plugin para exportar o conteúdo de um dispositivo gráfico X11 é obviamente mais útil quando localizado no menu de um dispositivo X11, e não na barra de menus principal. Além disso, esse plugin deve saber o número do dispositivo em que deve operar, sem precisar perguntar ao usuário sobre isso.

Chamamos esses plugins de dependentes de contexto. Consequentemente, no arquivo `.pluginmap`, eles não são (ou não apenas) colocados na **<hierarchy>** de comandos principal, mas sim em um elemento **<context>**. Até o momento, apenas dois contextos diferentes são suportados

(mais serão adicionados posteriormente): x11 e importação de arquivos. Abordaremos esses contextos em sequência. Mesmo que você esteja interessado apenas no contexto de importação, leia também a seção sobre o contexto x11, pois este é um pouco mais detalhado.

10.3.1 Contexto do dispositivo X11

Para usar um plugin no contexto de um dispositivo x11 - isto é, colocá-lo na barra de menus da janela que aparece quando você chama `x11()` no console, primeiro declare-o como de costume no arquivo `.pluginmap`:

```
<document [...]>
        <components>
                [...]
                <component id="meu_plugin__x11" file="meu_plugin_x11.xml"  ↩
                    label="Um plugin de contexto X11"/>
                [...]
        </components>
```

No entanto, você não precisa defini-lo na hierarquia (você pode, se também fizer sentido como um plugin de nível superior):

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

Em vez disso, adicione uma definição do contexto "x11" e adicione-a aos menus aqui:

```
<context id="x11">
                [...]
                <menu id="edit">
                        [...]
                        <entry id="meu_plugin_X11"/>
                </menu>
        </context>
</document>
```

Na seção logic do XML do plugin, você pode declarar duas propriedades **<external>**: *devnum* e *context*. *context* (se declarado) será definido como *"x11"* quando o plugin for invocado neste contexto. *devnum* será definido como o número do dispositivo gráfico no qual operar. E isso é tudo.

10.3.2 Contexto de importação de dados

Antes de ler esta seção, certifique-se de ler a seção sobre o Contexto do dispositivo X11, pois ela explica os conceitos básicos.

O contexto *"import"* é usado para declarar plugins de filtro de importação de arquivos. Você simplesmente os coloca em um contexto com *id="import"* no arquivo `.pluginmap`. No entanto, há um detalhe adicional ao declarar esses plugins: para oferecer uma caixa de diálogo de seleção de arquivos unificada para todos os tipos de arquivo suportados, você precisa declarar uma informação adicional em seu componente:

```
<document [...]>
        <components>
                [...]
```

```
                <component id="meu_plugin_importar_xyz" file=" ↩
                    meu_plugin_importar_xyz.xml" label="Importar arquivos  ↩
                    XYZ">
                        <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>
```

A linha de atributo simplesmente diz que as extensões de nome de arquivo associadas para os arquivos XYZ são `*.xyz` ou `*.zyx`, e que o filtro deve ser rotulado como 'Arquivos de dados XYZ' na caixa de diálogo de seleção de arquivos.

Você pode declarar duas propriedades **<external>** em seu plugin. *`filename`* será definido com o nome do arquivo selecionado e *`context`* será definido com *`"import"`*.

10.4 Consultando o R para obter informações

Em alguns casos, você pode querer obter mais informações do R para serem apresentadas na interface do usuário do seu plugin. Por exemplo, você pode querer oferecer uma seleção dos níveis de um fator que o usuário selecionou para análise. Desde a versão 0.6.2 do RKWard, isso é possível. Antes de começarmos, é importante que você esteja ciente de algumas ressalvas:

O código R executado dentro da lógica da interface do usuário do plugin é avaliado no loop de eventos do R, o que significa que ele pode ser executado *enquanto* outros cálculos estão em andamento. Isso garante que a interface do usuário do seu plugin seja utilizável, mesmo enquanto o R estiver ocupado realizando outras tarefas. No entanto, isso torna muito importante que seu código não tenha efeitos colaterais. Em particular:

- *Não* faça nenhuma atribuição em .GlobalEnv ou qualquer outro ambiente não local.
- *Não* imprima nada no arquivo de saída.
- *Não* plote nada na tela.
- Em geral, *não* faça nada que tenha efeitos colaterais. Seu código *pode ler informações*, mas não *"fazer"* nada.

Com isso em mente, aqui está o padrão geral. Você usará isso dentro de uma seção de lógica de interface do usuário com o script scripted UI logic:

```
<script><![CDATA[
                                let update = gui.addChangeCommand (" ↩
                                    variable", function () {
                                        gui.setValue ("selector.enabled",  ↩
                                            0);
```

```
                                    variable = gui.getValue ("variable ↩
                                        ");
                                    if (variable == "") return;

                                    new RCommand('levels (' + variable  ↩
                                        + ')', "myid").then(result =
> {
                                            gui.setValue ("selector. ↩
                                                enabled", 1);
                                            gui.setListValue ("selector ↩
                                                .available", result);
                                    }).catch(msg =
> {
                                            if (msg === "outdated")  ↩
                                                return;  // comando foi  ↩
                                                cancelado, pois um novo  ↩
                                                está prestes a chegar -> ↩
                                                benigno
                                            // possivelmente outros  ↩
                                                erros de tratamento, msg ↩
                                                 carrega os avisos e  ↩
                                                mensagens de erro  ↩
                                                produzidos,
                                            // se o comando falhou  ↩
                                                porexemplo:
                                            gui.setListValue ("selector ↩
                                                .available", Array (" ↩
                                                ERROR:", msg));
                                    });
                                });
                ]]></script>
```

Aqui, *variable* é uma propriedade que contém o nome de um objeto (por exemplo, dentro de um **<varslot>**). Sempre que isso mudar, você precisará atualizar a exibição dos níveis dentro do **<valueselector>**, chamado *selector*. A função chave aqui é a instrução construtora **new RCommand()**, que recebe como primeiro parâmetro a string de comando a ser executada. Observe que o comando é executado de forma assíncrona, o que torna as coisas um pouco mais complexas. Por um lado, você precisa garantir que seu **<valueselector>** permaneça desabilitado enquanto não contiver informações atualizadas. Em segundo lugar, como o usuário pode fazer alterações rapidamente, mais de um comando pode ter sido gerado antes de recebermos qualquer resultado. Portanto, você precisará garantir que execute apenas o comando mais recente.

Para lidar com a assincronia, o que é retornado aqui é um objeto **Promise**. Mais informações sobre esse poderoso recurso do JavaScript podem ser encontradas na internet. O importante a saber aqui é que adicionar uma instrução **.then()** permite especificar o que acontecerá quando o comando for concluído, e uma instrução **.catch()** pode ser usada para lidar com quaisquer erros. Novamente, lembre-se de que o bloco **.then()** não é executado imediatamente. Para entender as implicações disso, pode ser útil, durante o desenvolvimento, inserir um **Sys.sleep(1);** em seu comando R, para ver o que acontece, quando um comando não é concluído imediatamente.

Finalmente, para lidar com a geração de múltiplos comandos, você pode especificar um segundo argumento para **new RCommand()**, ("myid", neste exemplo). Quaisquer comandos com o mesmo identificador (escolhido livremente) serão entendidos como pertencentes à mesma fila. O RKWard garantirá então que apenas o comando mais recente acione o bloco **.then()** enquanto quaisquer comandos obsoletos chegarão no bloco **.catch()**. Aqui, os comandos obsoletos podem ser identificados, pois a string "outdated" é passada como seu valor, enquanto para quaisquer outros erros possíveis, os avisos e mensagens de erro são repassados.

Note que este exemplo é um tanto simplificado. Na realidade, você deve tomar precauções adicionais, por exemplo, para evitar colocar uma quantidade excessiva de níveis no seletor. A boa

notícia é que provavelmente você não precisa fazer tudo isso sozinho. O exemplo acima foi retirado do plugin **rkward::level_select**, por exemplo, que você pode simplesmente incorporar em seu próprio plugin. Isso permite até mesmo que você especifique uma expressão diferente para executar no lugar de **levels()**.

> NOTA
> Em versões anteriores do RKWard, os comandos R eram executados usando uma função um pouco mais complexa: **doRCommand()**. Você ainda pode encontrar isso em alguns plugins, mas não é recomendável usá-la em novos códigos.

10.5 Referenciando o objeto atual ou o arquivo atual

Para muitos plugins, é desejável trabalhar com o objeto 'atual'. Por exemplo, um plugin de 'classificação' poderia pré-selecionar os dados. O nome do objeto atual está disponível para os plugins como uma propriedade predefinida chamada `current_object`. Você pode se conectar a essa propriedade da maneira usual. Se nenhum objeto for atual, a propriedade equivale a uma string vazia. Da mesma forma, o URL do arquivo de script atual é acessível como uma propriedade predefinida chamada `current_filename`. Essa propriedade está vazia se nenhum arquivo de script estiver sendo editado ou se o arquivo de script ainda não tiver sido salvo.

Atualmente, o parâmetro `current_object` só pode ser da classe `data.frame`, mas não confie nisso, pois isso será estendido para outros tipos de dados no futuro. Se você estiver interessado apenas em objetos `data.frame`, conecte-se à propriedade `current_dataframe`. Como alternativa, você pode impor requisitos de tipo usando restrições apropriadas em seus comandos **<varslot>**s, ou usando scripts de lógica GUI.

10.6 Opções repetidas (um conjunto de)

Às vezes, você deseja repetir um conjunto de opções para um número arbitrário de itens. Por exemplo, suponha que você queira implementar um plugin para classificar um data.frame. Você pode querer permitir a classificação por um número arbitrário de colunas (em caso de empate entre as primeiras colunas). Isso pode ser facilmente implementado permitindo que o usuário selecione várias variáveis em um **<varslot>** com `multi="true"`. Mas se você quiser estender isso, por exemplo, permitindo que o usuário especifique para cada variável se ela deve ser convertida para caractere/numérica ou se a classificação deve ser crescente ou decrescente, você precisa de mais flexibilidade. Outros exemplos seriam plotar várias linhas em um único gráfico (permitindo selecionar objeto, estilo de linha, cor da linha etc. para cada linha) ou especificar um mapeamento para recodificar de um conjunto de valores antigos para novos valores.

Digite o **<conjunto de opções>**. Vamos analisar um exemplo simples primeiro:

```
<dialog [...]>
        [...]
        <optionset id="set" min_rows="1">
                <content>
                        <row>
                                <input id="firstname" label="Nome(s)  ↩
                                    fornecido(s)" size="small">
                                <input id="lastname" label="Nome de família ↩
                                    " size="small">
                                <radio id="gender" label="Gênero">
                                        <optioncolumn label="Masculino"  ↩
                                            value="m"/>
```

```
                                        <optioncolumn label="Feminino" ↩
                                            value="f"/>
                                    </radio>
                            </row>
                    </content>

                    <optioncolumn id="firstnames" label="Nome(s) fornecido(s)" ↩
                        connect="firstname.text">
                    <optioncolumn id="lastnames" label="Nome de família" ↩
                        connect="lastname.text">
                    <optioncolumn id="gender" connect="gender.string">
            </optionset>
            [...]
</dialog>
```

Aqui, criamos uma interface de usuário para especificar um número de pessoas (por exemplo, autores). A interface requer pelo menos uma entrada (*min_rows="1"*). Dentro do elemento **<optionset>**, começamos especificando o **<content>**, isto é, os elementos que pertencem ao conjunto de opções. Você já deve estar familiarizado com a maioria dos elementos dentro do **<content>**.

Em seguida, especificamos as variáveis de interesse que queremos ler do conjunto de opções em nosso arquivo JS. Como lidaremos com um número arbitrário de itens, não podemos simplesmente ler `getString("firstname")` em JS. Em vez disso, para cada valor de interesse, especificamos um **<optioncolumn>**. Para a primeira optioncolumn no exemplo, '**<connect="firstname.text">** significa que o conteúdo do elemento "firstname" é lido para cada item. Os itens **<optioncolumn>** para os quais um *label* é fornecido serão exibidos na tela, em uma coluna correspondente a esse rótulo. Em JS, agora podemos buscar os primeiros nomes de todos os autores usando `getList("set.firstname")`, `getList("set.lastnames")` para os sobrenomes e `getList("set.gender")` para um array de strings "m"/"f".

Observe que não há restrições quanto ao que você pode colocar dentro de um **<optionset>**. Você pode até usar componentes embedded. Assim como com qualquer outro elemento, tudo o que você precisa fazer é coletar as variáveis de saída de interesse em uma **<optioncolumn>**-specification. No caso de plugins incorporados, isso geralmente é uma seção da propriedade "code". Por exemplo:

```
<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>
```

Claro que você também pode usar lógica de interface do usuário dentro de um optionset. Existem duas opções para fazer isso: Você pode fazer isso estabelecendo uma conexão (ou script) na seção principal **<logic>** do seu plugin, como de costume. No entanto, você acessará os elementos da interface do usuário na região de conteúdo como (por exemplo) "set.contents.firstname.XYZ". Observe o prefixo "set" (o *id* que você atribuiu ao conjunto e "contents"). Alternativamente, você pode adicionar uma seção **<logic>** separada como um elemento filho do seu **<optionset>**. Nesse caso, os *id*s serão endereçados em relação à região de conteúdo, por exemplo, "firstname.XYZ".

Somente o elemento **<script>** não é permitido na seção de **<logic>** do plugin.Se você quiser usar scripts, terá que utilizar a seção principal **<logic>** do plugin.

> NOTA
> Ao criar scripts de lógica em um conjunto de opções, tudo o que você pode fazer é acessar a região de conteúdo *atual*. Portanto, normalmente, só faz sentido conectar elementos dentro da região de conteúdo entre si. Conectar uma propriedade fora do **<optionset>** a uma propriedade dentro da região de conteúdo pode ser útil para inicialização. No entanto, modificar a região de conteúdo após a inicialização *não* se aplicará aos itens que o usuário já definiu. Apenas ao item atualmente selecionado no conjunto.

10.6.1 ″Driven″ optionsets

Até agora, consideramos um **<optionset>** que fornece botões para adicionar/remover itens. No entanto, em alguns casos, é muito mais natural selecionar itens fora do **<optionset>** e fornecer opções para personalizar alguns aspectos de cada item apenas em um **<optionset>**. Por exemplo, suponha que você queira permitir que o usuário plote vários objetos em um único gráfico. Para cada objeto, o usuário deve poder especificar a cor da linha. Você poderia resolver isso colocando um **<varselector>** e um **<varslot>** dentro da área **<content>**, permitindo que o usuário selecione um item por vez. No entanto, isso significará muito menos cliques para o usuário se você usar um **<varslot multi=″true″″>** *fora* do **<optionset>**. Em seguida, você conectará essa seleção de objetos a um conjunto de opções chamado ″controlado″. Veja como:

```
<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>
```

Começaremos analisando o exemplo abaixo. Você notará que duas especificações de **<optioncolumn>** têm *external=″true″*. Isso indica à RKWard que elas são controladas de fora do **<optionset>**. Aqui, o único propósito da coluna de opções ″varnames″ é fornecer rótulos de fácil leitura na exibição do conjunto de opções (ela está conectada ao modificador ″shortname″ da propriedade que contém os objetos selecionados). O propósito da coluna de opções ″vars″ é servir como a coluna ″chave″, conforme especificado por **<optionset keycolumn=″vars″...>**. Isso

significa que para cada entrada nesta lista, o conjunto oferecerá um conjunto de opções, e as opções estão logicamente vinculadas a essas entradas. Esta coluna está conectada à propriedade que contém os objetos selecionados no **<varslot>**. Ou seja, para cada objeto selecionado ali, o **<optionset>** permitirá especificar a cor da linha.

> NOTA
> A coluna externa também pode ser `connect` (conectada) a propriedades dentro da região **<content>**. No entanto, é importante observar que as colunas de opções declaradas como `external="true"` nunca devem ser modificadas de dentro do **<optionset>**, e as colunas de opções declaradas como `external="false"` (o padrão) nunca devem ser modificadas de fora do **<optionset>**.

10.6.2 Alternativas: Quando não usar conjuntos de opções

Optionsets são uma ferramenta poderosa, mas às vezes podem causar mais danos do que benefícios, pois adicionam uma complexidade considerável, tanto da perspectiva de um desenvolvedor de plugins quanto da perspectiva de um usuário. Portanto, pense duas vezes antes de usá-los. Aqui estão algumas dicas:

- Para alguns casos simples, o elemento **<matrix>** pode fornecer uma alternativa leve e útil.
- Não sobrecarregue seu plugin com muitas funcionalidades. Demos o exemplo de usar um conjunto de opções (optionset) para que um plugin desenhe várias linhas em um único gráfico. Mas, em geral, não é uma boa ideia criar um plugin que gere gráficos individuais para cada item em um conjunto de opções. Em vez disso, faça com que o plugin gere um único gráfico, e o usuário possa chamá-lo várias vezes.
- Se você não espera mais do que dois ou três itens em um conjunto, considere repetir as opções manualmente.

Capítulo 11

Lidar com dependências e problemas de compatibilidade

11.1 Compatibilidade com a versão do RKWard

Fazemos o possível para garantir que os plugins desenvolvidos para uma versão antiga do RKWard continuem funcionando em versões posteriores do RKWard. No entanto, o inverso nem sempre é verdadeiro, pois novos recursos são adicionados. Como nem todos os usuários estão executando a versão mais recente do RKWard, isso significa que seu plugin pode não funcionar para todos.

Quando você estiver ciente de tais problemas de compatibilidade, certifique-se de documentar esse fato em seu arquivo `.pluginmap`, usando o elemento **<dependencies>**. O elemento **<dependencies>** pode ser especificado como um filho direto do elemento <document> do `.pluginmap`, ou como um elemento filho de definições individuais de **<component>**. No primeiro caso, as dependências se aplicam a *todos* plugins no mapa. No segundo caso, apenas ao(s) **<component>**(s) individual(is). Você também pode misturar dependências "globais" e "específicas". Nesse caso, as dependências "globais" são adicionadas às do componente individual.

Vejamos um pequeno exemplo:

```
<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>
```

Neste exemplo, sabe-se que todos os plugins requerem pelo menos a versão 0.5.0c do RKWard. Um plugin, com *id="myplugin"* é fornecido em duas variantes alternativas. A primeira versão, simplificada, será usada para versões do RKWard anteriores à 0.6.1. A segunda utiliza recursos que são novos no RKWard 0.6.1, e será usada somente a partir do RKWard 0.6.1 em diante.

Fornecer variantes alternativas como esta é uma maneira muito amigável de usar novos recursos, mantendo o suporte para versões anteriores do RKWard. As versões alternativas devem compartilhar o mesmo `id` (caso contrário, serão exibidos avisos) e só podem ser definidas *dentro do mesmo* arquivo `.pluginmap`.

Plugins que não são compatíveis com a versão em execução do RKWard, e que não vêm com uma versão alternativa serão ignorados com um aviso.

> NOTA
> Na verdade, a versão 0.6.1 do RKWard é a primeira a interpretar dependências e a reportar erros de dependência. Portanto, ao contrário do que o exemplo possa sugerir, especificar versões anteriores nas dependências não terá efeito direto (mas ainda pode ser uma boa ideia para fins de documentação).

Às vezes será possível até mesmo lidar com problemas de incompatibilidade de versão *dentro* de um único arquivo `.pluginmap` usando o elemento **<dependency_check>**, descrito na seção seguinte.

11.2 Compatibilidade de versão do R

Semelhante a `rkward_min_version` e `rkward_max_version`, o elemento **<dependencies>** permite a especificação dos atributos `R_min_version` e `R_max_version`. No entanto, existem as seguintes diferenças:

- Plugins que não atendem ao requisito de versão R não são ignorados atualmente ao ler um arquivo `.pluginmap`. O usuário ainda pode chamar o plugin e não verá nenhum aviso imediato (em versões futuras, provavelmente uma mensagem de aviso será exibida).
- Consequentemente, também *não* é possível definir versões alternativas de um plugin dependendo da versão em execução do R.
- No entanto, muitas vezes é fácil obter compatibilidade com versões anteriores, como mostrado abaixo. Se você estiver ciente de problemas de compatibilidade com o R, considere usar este método, em vez de definir uma dependência em uma versão específica do R.

Em muitos casos, é facilmente possível fornecer funcionalidade reduzida, se um determinado recurso não estiver disponível na versão em execução do R. Considere o seguinte exemplo curto de um arquivo de plugin `.xml`:

```
<dialog [...]>
        <logic>
                <dependency_check id="ris210" R_min_version="2.10.0"/>
                <connect client="compression.xz.enabled" governor="ris210 ↩
                    "/>
        </logic>
        [...]
        <radio id="compression" label="Método abrangente">
                <option label="Nenhum" value="">
                <option label="gzip" value="gzip">
                <option id="xz" label="xz" value="xz">
        </radio>
        [...]
</dialog>
```

Neste exemplo, a opção de compressão "xz" será simplesmente desativada quando a versão do runtime do R for anterior à 2.10.0 (que não suportava compressão xz). O elemento **<dependency_check>** suporta os mesmos atributos que o elemento **<dependencies>** nos arquivos `.pluginmap`. Ele cria uma propriedade booleana, que é verdadeira se as dependências especificadas forem atendidas e falsa caso contrário.

11.3 Dependências de pacotes R

É possível definir dependências em pacotes R específicos, mas a partir da versão 0.6.1 do RKWard, essas dependências não são verificadas nem instaladas/carregadas automaticamente. Elas são exibidas nos arquivos de ajuda do plugin. Aqui está um exemplo de definição:

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

> NOTA
> Certifique-se sempre de adicionar as chamadas apropriadas de `require()`, se o seu plugin precisar que determinados pacotes sejam carregados.

> NOTA
> Se você distribuir seu `.pluginmap` como um pacote R, e todos os plugins dependerem de um pacote específico, então você deve definir essa dependência no nível do pacote R. Definir dependências para pacotes R no nível do `.pluginmap` do RKWard é mais útil se apenas alguns de seus plugins precisarem da dependência, a dependência não estiver disponível no CRAN, ou seu `.pluginmap` não for distribuído como um pacote R.

11.4 Dependências de outros `.pluginmap` do RKWard

Se seus plugins dependem de plugins definidos em outro `.pluginmap` (ou seja, *não*faz parte do seu pacote), você pode definir essa dependência assim:

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

Atualmente, o plugin não carrega, instala ou sequer avisa sobre a falta de `.pluginmap`, mas pelo menos as informações sobre as dependências (e onde obtê-las) serão exibidas na página de ajuda do plugin. Você não precisa (e não deve) declarar dependências de `.pluginmap` que são distribuídos com a distribuição oficial do RKWard, ou de `.pluginmap` que estão dentro do seu próprio pacote. Além disso, se um `.pluginmap` necessário for distribuído como um pacote R, declare uma

dependência do pacote (como mostrado na seção anterior), em vez de declarar dependências do plugin.

Para garantir que os `.pluginmap` necessários sejam realmente carregados, use a tag **<require>** (consulte a reference para obter detalhes).

11.5 Um exemplo

Para esclarecer como as definições de dependência podem ser misturadas, aqui está um exemplo combinado:

```
<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>
```

Capítulo 12

Traduções de plugin

Até agora, usamos alguns conceitos sobre traduções ou ″i18n″ (abreviação de ″internacionalização″, que tem 18 caracteres entre i e n no inglês) de passagem. Neste capítulo, apresentamos uma explicação mais detalhada da funcionalidade de i18n para plugins do RKWard. Na maioria dos casos, você *não* precisará de tudo isso em seus plugins. No entanto, pode ser uma boa ideia ler este capítulo na íntegra, pois a compreensão desses conceitos deve ajudá-lo a criar plugins totalmente traduzíveis e que permitam traduções de alta qualidade.

12.1 Considerações gerais

Um ponto importante a entender sobre traduções de software, em contraste com traduções de outros materiais de texto, é que os tradutores frequentemente terão bastante dificuldade em obter uma visão completa *do que* estão traduzindo. As traduções de software são necessariamente baseadas em fragmentos de texto bastante curtos: cada rótulo que você atribui a uma **<option>** em um **<radio>**, cada string que você marca para tradução em uma chamada de função **i18n()**, formará uma ″unidade de tradução″ separada. Em essência, cada fragmento será apresentado ao tradutor isoladamente. Bem, não completamente isolado, pois tentamos fornecer ao tradutor o máximo de contexto significativo que puder ser extraído automaticamente. Mas, em alguns momentos, os tradutores precisarão de contexto adicional para entender uma string, especialmente quando as strings são curtas.

12.2 i18n em arquivos xml do RKWard

Para arquivos XML do RKWard, a internacionalização (i18n) geralmente funciona sem problemas. Se você estiver escrevendo seu próprio **.pluginmap** (por exemplo, para um plugin externo), você precisará especificar um `po_id` ao lado do `id` do pluginmap. Isso define o ″catálogo de mensagens″ a ser usado. Em geral, isso deve ser definido de forma idêntica ao `id` do seu **.pluginmap**, mas se você fornecer vários **.pluginmap**s relacionados em um único pacote, provavelmente desejará especificar um `po_id` comum em seus mapas. O `po_id` de um arquivo **.pluginmap** é herdado por todos os plugins declarados nele, a menos que declare um `po_id` diferente.

Para plugins e páginas de ajuda, você não precisa especificar para o RKWard quais strings devem ser traduzidas, pois isso geralmente fica evidente pelo seu uso. No entanto, como explicado acima, você deve ficar atento a strings que podem ser ambíguas ou que precisam de alguma explicação para serem traduzidas corretamente. Para strings que podem ter significados diferentes, forneça um `i18n_context` como este:

```
<checkbox id="scale" label="Escala" i18n_context="Mostra a escala"/>
<checkbox id="scale" label="Escala" i18n_context="Escala o gráfico"/>
```

Fornecer *i18n_context* fará com que as duas strings sejam traduzidas separadamente. Caso contrário, elas compartilhariam uma única tradução. Além disso, o contexto é mostrado ao tradutor. O atributo *i18n_context* é compatível com todos os elementos que podem ter strings traduzíveis, em algum lugar, incluindo elementos que contêm texto dentro deles (por exemplo, elementos **por exemplotext>**).

Em outros casos, a sequência a ser traduzida tem um único significado não ambíguo, mas ainda pode precisar de alguma explicação. Nesse caso, você pode adicionar um comentário que será mostrado aos tradutores. Exemplos podem incluir:

```
<!-- i18n: Não, isto não é um erro de digitação para screen plot! -->
<component id="scree_plot" label="Scree plot"/>

<!-- i18n: Se possível, por favor, faça com que esta string seja curta. Ter ↩
     mais de cerca de 15 caracteres
fica muito feio neste ponto, e o significado deve ser bastante auto- ↩
    evidente para o
usuário (seleção de uma lista de valores exibida ao lado deste elemento)  ↩
    -->
<valueslot id="selected" label="Pegue um"/>
```

Observe que tais comentários devem preceder o elemento ao qual se aplicam e devem começar com ″i18n:″ ou ″TRANSLATORS:″.

Finalmente, em casos raros, você pode querer excluir certas strings da tradução. Isso pode fazer sentido, por exemplo, se você oferecer uma escolha entre vários nomes de função R em um controle **<radio>**. Nesse caso, você não quer que esses nomes sejam traduzidos (mas, dependendo do contexto, você deve considerar fornecer um rótulo descritivo).

```
<radio id="transformation" label="Função R a aplicar">
  <option id="as.list" noi18n_label="as.list()"/>
  <option id="as.vector" noi18n_label="as.vector()"/>
  [...]
</radio>
```

Observe que você omitirá o atributo *label*, e especificará *noi18n_label* em vez disso. Além disso, observe que, ao contrário de *i18n_context* e comentários, o uso de *noi18n_label* tornará seu plugin incompatível com versões do RKWard anteriores à 0.6.3.

12.3 i18n nos arquivos e seções js do RKWard

Em contraste com os arquivos `.xml`, tornar os arquivos `.js` de um plugin traduzíveis requer mais trabalho personalizado. A principal diferença aqui é que não há uma maneira automática adequada de determinar se uma string deve ser exibida como uma string legível por humanos ou como um trecho de código. Portanto, você precisa marcá-la você mesmo. Já mostramos exemplos disso ao longo do texto. Aqui está uma descrição mais completa das funções de internacionalização (i18n) disponíveis no código JavaScript e algumas dicas para casos mais complexos:

i18n (msgid, [...])

A função mais importante. Marca a string para tradução. A string (traduzida ou não) é retornada entre aspas duplas (″). Um número arbitrário de marcadores de posição pode ser usado na string, como mostrado abaixo. Usar esses marcadores de posição em vez de concatenar pequenas substrings é muito mais fácil para os tradutores:

```
i18n ("Comparar objetos %1 e %2", getString ('x'), getString ('y'));
```

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

O mesmo que **i18n()**, mas fornecendo adicionalmente uma mensagem de contexto:

```
i18nc ("nome próprio, não estado de espírito", "Teste de humor");
```

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

Semelhante a **i18n()**, mas para mensagens que podem ser diferentes no singular ou plural (e algumas linguagens diferenciam ainda mais formas numéricas). Observe que, assim como com **i18n()**, você pode usar um número arbitrário de substituições, mas a primeira ('%1') é obrigatória e deve ser um número inteiro.

```
i18np ("Comparando um único par", "Comparando %1 pares distintos",  ↩
    n_pairs);
```

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

i18np() com mensagem de contexto adicionada.

comment (comentário, [endentação])

Exibe um comentário de código, marcado para tradução. Ao contrário das outras funções i18n(), esta não é colocada entre aspas, mas um '#' é adicionado a cada linha do comentário.

```
comment ("Transpor a matriz");
                        echo ('x <- t (x)\n');
```

Para adicionar comentários aos tradutores (consulte acima para uma discussão sobre as diferenças entre comentário e contexto), adicione um comentário começando com ″i18n:″ ou ″translators:″ diretamente acima da chamada de comentário **i18n()**. Exemplo:

```
// i18n: A grafia está correta: Scree plot.
                echo ('rk.header (' + i18n ("Scree plot") + ')\n');
```

12.3.1 i18n e aspas

Na maioria dos casos, você não precisará se preocupar com o comportamento do i18n() em relação às aspas. Como, normalmente, as strings traduzíveis são literais de string, colocá-las entre aspas é a coisa certa a fazer e economiza digitação. Além disso, em funções como **makeHeaderCode()/Header()** que geralmente colocam seus argumentos entre aspas, as strings i18n() são protegidas contra aspas duplicadas. Essencialmente, isso funciona enviando a string traduzida primeiro por meio de **quote()** (para colocá-la entre aspas) e, em seguida, por meio de **noquote()** (para protegê-la de aspas adicionais). Caso você precise de uma string traduzível que não esteja entre aspas, use **i18n(noquote ("Minha mensagem"))**. Caso você precise que uma string traduzível seja citada uma segunda vez, envie-a através de: **quote()**, *duas vezes*.

Dito isso, geralmente não é uma boa ideia tornar partes como nomes de funções ou nomes de variáveis traduzíveis. Para começar, o R, a linguagem de programação, é inerentemente em inglês e não há internacionalização da linguagem em si. Comentários de código são um caso à parte, mas você deve usar a função **comment()** para eles. Em segundo lugar, tornar partes sintaticamente relevantes do código gerado traduzíveis significa que as traduções podem, na verdade, quebrar seu plugin. Por exemplo, se um tradutor desavisado traduzir uma string destinada a ser um nome de variável em duas palavras distintas com um espaço entre elas.

12.4 Manutenção da tradução

Agora que você tornou seu plugin traduzível, como você realmente o traduz? Em geral, você só precisa se preocupar com isso ao desenvolver um plugin externo. Para plugins no repositório principal do RKWard, toda a mágica já foi feita para você. Aqui está o fluxo de trabalho básico para plugins externos. Observe que você precisa ter as ferramentas "gettext" instaladas:

- Marque todas as strings, fornecendo contexto e comentários conforme necessário
- Execute **python3 scripts/update_plugin_messages.py --extract-only /caminho/para/meu.pluginmap**. O script scripts/update_plugin_messages.py não faz parte das versões de código-fonte, mas pode ser encontrado em um repositório de código-fonte.
- Distribua o arquivo **rkward__*POID*.pot** resultante aos seus tradutores. Para plugins externos, é recomendado colocá-lo em uma subpasta "po" em inst/rkward.
- O tradutor abre o arquivo em uma ferramenta de tradução como o **lokalize**. Na verdade, mesmo que você não vá preparar nenhuma tradução, você mesmo deveria experimentar este passo. Examine as strings extraídas em busca de problemas/ambiguidades.
- O tradutor salva a tradução como **rkward__*POID*.*xx*.po** (onde *xx* é o código do idioma) e a envia de volta para você.
- Copie **rkward__*POID*.*xx*.po** para suas fontes, ao lado de **rkward__*POID*.pot**. Execute **python3 scripts/update_plugin_messages.py /path/to/my.pluginmap** (Observação: sem *--extract-only*, desta vez). Isso irá mesclar a tradução com quaisquer alterações de string intermediárias, compilar a tradução e instalá-la em ***DIRETÓRIO_DO_MAPA_DE_PLUGIN*/po/*xx*/LC_MESSAGES/rkward__*POID*.mo** (onde *xx* é o código do idioma, novamente).
- Você também deve incluir a tradução não compilada (ou seja, **rkward__*POID*.*xx*.po**) em sua distribuição, no subdiretório "po".
- Para qualquer atualização do seu plugin, execute **python3 scripts/update_plugin_messages.py /caminho/para/meu.pluginmap** para atualizar o arquivo .pot mas também os arquivos .po existentes e os catálogos de mensagens compilados.

12.5 Escrevendo traduções para plugin

Presumimos que você conheça sua profissão de tradutor ou esteja disposto a se aprofundar nela em outro lugar. Algumas palavras específicas sobre traduções de plugins:

- Os plugins do RKWard não eram traduzíveis até a versão 0.6.3 e, em sua maioria, não foram escritos com a internacionalização (i18n) em mente antes disso. Portanto, você encontrará strings ambíguas e outros problemas de i18n com mais frequência do que em outros projetos mais maduros. Por favor, não ignore esses problemas, mas nos informe (ou aos mantenedores dos plugins) para que possamos corrigi-los.
- Muitos plugins RKWard se referem a termos altamente especializados, desde manipulação de dados e estatística, mas também de outras áreas da ciência. Em muitos casos, uma boa tradução exigirá pelo menos conhecimento básico dessas áreas. Em alguns casos, não há uma boa tradução para um termo técnico, e a melhor opção pode ser deixar o termo sem tradução ou incluir o termo em inglês entre parênteses. Não se concentre muito na marca de 100% de traduções, concentre-se em fornecer uma boa tradução, mesmo que isso signifique pular algumas strings (ou até mesmo pular alguns catálogos de mensagens por inteiro). Outros usuários podem preencher quaisquer lacunas em termos técnicos.

Capítulo 13

Informações de autor, licença e versão

Você criou um conjunto de plugins e está se preparando para compartilhar seu trabalho. Para garantir que os usuários saibam do que se trata o seu trabalho, sob quais termos eles podem usá-lo e distribuí-lo, e com quem devem entrar em contato para problemas ou ideias, você deve adicionar algumas informações *sobre* seus plugins. Isso pode ser feito usando o elemento **<about>**. Ele pode ser usado tanto no `.pluginmap` quanto em arquivos `.xml` de plugins individuais (em ambos os casos, como um filho direto da tag document). Quando especificado no `.pluginmap`, ele se aplicará a todos os plugins. Se **<about>** for especificado em ambos os locais, as informações de **<about>** no arquivo `.xml` do plugin substituirão as do arquivo do `.pluginmap`. Você também pode adicionar um elemento **<about>** às páginas .rkh que não estão conectadas a um plugin, se necessário.

Aqui está um exemplo de arquivo `.pluginmap` com apenas algumas explicações. Em caso de dúvida, mais informações podem ser encontradas na documentação.

```
<document
        namespace="rkward"
        id="SquaretheCircle_rkward"
>
        <about
                name="Quadrado para Círculo"
                shortinfo="Quadrado para círculo usando a compensação de ↩
                    Heisenberg."
                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>
```

A maior parte disso deve se explicar por si só, então não vamos discutir cada elemento de tag individualmente. Mas vamos analisar alguns detalhes que provavelmente precisam de comentários para facilitar a compreensão.

O elemento *category* em **<about>** pode ser definido de forma bastante livre, mas deve ser significativo, pois ele é usado para organizar plugins em grupos. Todos os outros atributos nesta tag de abertura são obrigatórios e devem ser preenchidos com conteúdo relevante.

Pelo menos um **<author>** com um endereço de e-mail válido e a função 'aut' ('author') também devem ser fornecidos. Caso seu plugin cause problemas ou alguém queira expressar sua gratidão, você poderá entrar em contato com alguém envolvido. Para mais informações sobre outras funções válidas, como 'ctb' para contribuidores de código ou 'cre' para manutenção de pacotes, consulte a documentação do R sobre person().

> NOTA
> Lembre-se de que você pode usar **<include>** e/ou **<insert>** para repetir informações em vários arquivos `.xml` (por exemplo, informações sobre um autor que esteve envolvido com vários plugins). Mais informações.

> DICA
> Você não precisa escrever este código XML manualmente. Se você usar a função `rk.plugin.skeleton()` do pacote rkwarddev e fornecer todas as informações necessárias através da opção *about*, ela criará automaticamente um arquivo `.pluginmap` com uma seção <about> funcional para você.

Capítulo 14

Compartilhar seu trabalho com outras pessoas

14.1 Plugins externos

A partir da versão 0.5.5, o RKWard oferece uma maneira prática de instalar plugins adicionais de terceiros que não pertencem ao pacote principal em si. Chamamos esses plugins de 'plugins externos'. Eles vêm na forma de um pacote R e podem ser gerenciados diretamente por meio dos recursos usuais de gerenciamento de pacotes do R e/ou do RKWard.

Esta seção da documentação descreve como os plugins externos devem ser empacotados, para que o RKWard possa usá-los. A criação do plugin em si é, obviamente, idêntica às seções anteriores. Ou seja, você provavelmente deve primeiro escrever um plugin funcional e, em seguida, voltar aqui para aprender como distribuí-lo.

Como os plugins externos são um recurso relativamente novo, os detalhes sobre isso provavelmente mudarão em versões futuras. Você pode contribuir com suas ideias para melhorar o processo.

> DICA
> Esta documentação explica os detalhes dos plugins externos para que você possa aprender como eles funcionam. Além disso, dê uma olhada também no pacote rkwarddev, que foi projetado para automatizar grande parte do processo de escrita.

14.2 Por que usar plugins externos?

O número de pacotes para estender a funcionalidade do R é imenso e continua crescendo. Por um lado, queremos incentivar você a escrever plugins até mesmo para as tarefas mais especializadas que você precisa resolver. Por outro lado, o usuário médio não deve se perder em enormes árvores de menus cheias de termos estatísticos desconhecidos. Portanto, pareceu razoável manter o gerenciamento de plugins no RKWard bastante modular também. A equipe do RKWard mantém seu próprio repositório público de pacotes em https://files.kde.org/rkward/R/ , destinado a hospedar seus plugins externos.

Como regra geral, plugins que parecem servir a um propósito amplamente utilizado (por exemplo, testes t) devem se tornar parte do pacote principal, enquanto aqueles que atendem a um grupo bastante limitado de pessoas com interesses específicos devem ser fornecidos como um pacote opcional. Para você, como autor de plugin, a melhor prática é simplesmente começar com um plugin externo.

14.3 Estrutura de um pacote de plugins

Para que os plugins externos sejam instalados e funcionem corretamente, eles devem seguir algumas diretrizes estruturais em relação à sua hierarquia de arquivos.

14.3.1 Hierarquia de arquivos

Vamos dar uma olhada na hierarquia de arquivos prototípica de um plugin elaborado. Você não precisa incluir todos esses diretórios e/ou arquivos para que um plugin funcione (continue lendo para saber o que é absolutamente necessário), considere isso um exemplo de 'boa prática':

```
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
```

> **NOTA**
> Neste exemplo, todas as ocorrências de plugin_name, testsuite_name e sometest_name devem ser substituídas por seus nomes corretos, respectivamente. Além disso, ll é um marcador para uma abreviação de idioma (por exemplo, 'de', 'en' ou 'es').

> **DICA**
> Você não precisa criar essa hierarquia de arquivos manualmente. Se você usar a função `rk.plugin.skeleton()` do pacote rkwarddev, ela criará automaticamente todos os arquivos e diretórios necessários para você, exceto o diretório `po`, que é criado e gerenciado pelo script de tradução.

14.3.1.1 Componentes de um plugin básico

É obrigatório incluir pelo menos três arquivos: um `.pluginmap`, um arquivo .xml com a descrição do plugin e um arquivo .js com a descrição do plugin. Ou seja, até mesmo o diretório "plugins" é opcional. Ele pode ajudar a organizar seus arquivos, principalmente se você incluir mais de um plugin/diálogo no arquivo, o que não é problema algum. Você pode ter quantos diretórios achar necessários para os arquivos de plugin, contanto que eles tenham nomes semelhantes aos do `.pluginmap`, respectivamente. Também é possível incluir vários arquivos `.pluginmap`, se isso atender às suas necessidades, mas você deve incluí-los todos em 'plugin_name.pluginmap'.

Cada pacote R deve ter um arquivo `DESCRIPTION` válido, que também é crucial para o RKWard o reconhecer como um provedor de plugin. A maior parte das informações que ele contém também é necessária nas Meta-informações do plugin e possivelmente nas dependências, mas em um formato diferente (a documentação do R explica o arquivo DESCRIPTION em detalhes).

Além do conteúdo geral de um arquivo `DESCRIPTION`, certifique-se de incluir também a linha 'Aprimora: rkward'. Isso fará com que o RKWard verifique automaticamente o pacote em busca de plugins, caso ele esteja instalado. Um exemplo de arquivo `DESCRIPTION` se parece com isto:

```
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"))
```

> **DICA**
> Você não precisa escrever este arquivo manualmente. Se você usar a função `rk.plugin.skeleton()` do pacote rkwarddev e fornecer todas as informações necessárias através da opção 'about', ela criará automaticamente um arquivo `DESCRIPTION` funcional para você.

14.3.1.2 Informações adicionais (opcional)

`ChangeLog`, `README`, `AUTHORS`, `LICENSE` devem ser autoexplicativos e são totalmente opcionais. Na verdade, eles não serão interpretados pelo RKWard, portanto, servem para conter informações adicionais que podem ser relevantes, por exemplo, para distribuidores. A maior parte do conteúdo relevante (créditos do autor, termos da licença por exemplo) já estará incluída nos arquivos do plugin (consulte a seção sobre meta-informações). Observe que todos esses arquivos também podem ser colocados em algum lugar no diretório "inst", se você quiser que eles estejam presentes não apenas no arquivo de origem, mas também no pacote instalado.

14.3.1.3 Teste automatizado de plugins (opcional)

Outro diretório opcional é "tests", que se destina a fornecer arquivos necessários para testes automatizados de plugins. Esses testes são úteis para verificar rapidamente se seus plugins ainda funcionam com novas versões do R ou do RKWard. Se você quiser incluir testes, é altamente recomendável restringir-se ao esquema de nomenclatura e hierarquia mostrados aqui. Ou seja, os testes devem residir em um diretório chamado `tests`, que inclui um arquivo `testsuite.R` e uma pasta com padrões de teste nomeados de acordo com o conjunto de testes apropriado. Você pode, no entanto, fornecer mais de um conjunto de testes; nesse caso, se não quiser anexá-los todos em um único arquivo `testsuite.R`, você pode dividi-los em, por exemplo; um arquivo para cada conjunto de testes e criar um arquivo `testsuite.R` com chamadas `source()` para cada arquivo de conjunto de testes. Em ambos os casos, crie subdiretórios separados com padrões de teste para cada conjunto de testes definido.

A vantagem de manter essa estrutura é que os testes de plugins podem ser executados simplesmente chamando `rktests.makplugintests()` do pacote rkwardtests sem argumentos adicionais. Consulte a documentação online em Testes Automatizados de Plugins para obter mais detalhes.

14.4 Construindo o pacote do plugin

Como explicado anteriormente, os plugins do RKWard externos são, na verdade, pacotes R, e, portanto, o processo de empacotamento é idêntico. Ao contrário dos pacotes R "reais", um pacote de plugin puro não carrega nenhum código R adicional (embora você possa, é claro, adicionar plugins RKWard a pacotes R usuais também, usando os mesmos métodos explicados aqui). Isso deve tornar ainda mais fácil a criação de um pacote funcional, desde que você tenha um arquivo `DESCRIPTION` válido e siga a hierarquia de arquivos explicada nas seções anteriores.

A maneira mais fácil de realmente construir e testar seu plugin é usar o comando R na linha de comando, por exemplo:

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

> DICA
> Você não precisa compilar o pacote dessa forma na linha de comando. Se você usar a função `rk.build.package()` do pacote rkwarddev, ela compilará e/ou verificará o pacote do seu plugin para você.

Capítulo 15

Desenvolvimento de plugins com o pacote rkwarddev

15.1 Visão geral

A criação de plugins externos envolve a escrita de arquivos em três linguagens (XML, JavaScript e R) e a criação de uma hierarquia padronizada de diretórios. Para facilitar bastante o trabalho dos desenvolvedores de plugins, estamos fornecendo o pacote rkwarddev. Ele oferece diversas funções simples em R para criar o código XML para todos os elementos de diálogo, como abas, caixas de seleção, listas suspensas ou navegadores de arquivos, além de funções para criar código JavaScript e arquivos de ajuda do RKWard para começar. A função `rk.plugin.skeleton()` cria a árvore de diretórios esperada e todos os arquivos necessários em seus respectivos locais.

Este pacote não é instalado por padrão, mas precisa ser instalado manualmente a partir do repositório próprio do RKWard. Você pode fazer isso usando a GUI (**Configurações** → **Configurar pacotes**), ou a partir de qualquer sessão do R em execução:

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

O rkwarddev depende de outro pequeno pacote chamado 'XiMpLe', que é um analisador e gerador XML muito simples e também presente no mesmo repositório.

A documentação completa em formato PDF também pode ser encontrada lá. Uma introdução mais detalhada ao trabalho com o pacote pode ser encontrada navinheta do rkwarddev.

15.2 Exemplo prático

Para que você tenha uma ideia de como 'criar um script para um plugin' funciona, em comparação com a abordagem direta que você viu nos capítulos anteriores, criaremos o plugin completo do teste t mais uma vez -- desta vez apenas com as funções R do pacote rkwarddev.

> DICA
> O pacote adicionará uma nova caixa de diálogo GUI ao RKWard em **Arquivo** → **Exportar** → **Criar script de plugin do RKWard**. Como o nome sugere, você pode criar esqueletos de plugins para edição posterior com ele. Esta caixa de diálogo foi gerada por um script rkwarddev que você pode encontrar no diretório 'demo' do pacote instalado e nas fontes do pacote, como um exemplo adicional. Você também pode executá-lo chamando `demo("skeleton_dialog")`

15.2.1 Descrição da GUI

Você notará imediatamente que o fluxo de trabalho é consideravelmente diferente: Ao contrário de escrever o código XML diretamente, você não começa com a definição do **<document>**, mas diretamente com os elementos do plugin que você gostaria de ter na caixa de diálogo. Você pode atribuir cada elemento da interface -- sejam caixas de seleção, menus suspensos, slots de variáveis ou qualquer outra coisa -- a objetos R individuais e, em seguida, combinar esses objetos para formar a GUI propriamente dita. O pacote possui funções para cada tag XML que pode ser usada para definir a GUI do plugin, e a maioria delas tem até o mesmo nome, apenas com o prefixo `rk.XML.*`. Por exemplo, definir um elemento **<varselector>** e dois elementos **<varslot>** para as variáveis *"x"* e *"y"* do teste t de exemplo pode ser feito da seguinte forma:

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

O detalhe mais interessante é provavelmente *source=variables*: Uma característica importante do pacote é que todas as funções podem gerar IDs automáticos, então você não precisa se preocupar em pensar em valores de *id* ou em lembrá-los para se referir a um elemento específico do plugin. Você pode simplesmente fornecer os objetos R como referência, pois todas as funções que precisam de um ID de algum outro elemento também podem lê-lo desses objetos. `rk.XML.varselector()` é um pouco especial, pois geralmente não tem um conteúdo específico para gerar um ID (pode ter, mas apenas se você especificar um rótulo), então precisamos definir um nome de ID. Mas `rk.XML.varslot()` não precisaria dos argumentos *id.name* aqui, então isso seria suficiente:

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

Para recriar o código de exemplo exatamente como estava, você teria que definir todos os valores de ID manualmente. Mas como o pacote facilitará nossa vida, a partir de agora não nos preocuparemos mais com isso.

> **DICA**
> rkwarddev é capaz de muita automação para ajudar você a criar seus plugins. No entanto, pode ser preferível não usá-lo em sua totalidade. Se seu objetivo é produzir um código que não apenas funcione mas também possa ser facilmente lido e comparado ao seu script gerador por um ser humano, você deve considerar sempre definir IDs úteis com *id.name*. Nomear seus objetos R de forma idêntica a esses IDs também ajudará a obter um código de script fácil de entender.

Se você quiser ver como o código XML do elemento definido se parece se você o exportasse para um arquivo, basta chamar o objeto pelo nome. Então, se você agora chamar 'var.x' na sua sessão do R, você deverá ver algo como isto:

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

Algumas tags só são úteis no contexto de outras. Portanto, por exemplo, você não encontrará uma função para a tag **<option>**. Em vez disso, tanto os botões de opção quanto as listas suspensas são definidos incluindo suas opções como uma lista nomeada, onde os nomes representam os rótulos a serem exibidos na caixa de diálogo, e seu valor é um vetor nomeado que pode ter duas entradas: *val* para o valor de uma opção e o booleano *chk* para especificar se esta opção está marcada por padrão.

```
test.hypothesis <- rk.XML.radio("usar hipótese de teste",
        options=list(
                "Dois lados"=c(val="two.sided"),
                "Primeiro é maior"=c(val="greater"),
                "Segundo é maior"=c(val="less")
        )
)
```

O resultado é o seguinte:

```
<radio id="rad_usngtsth" label="usar hipótese de teste">
        <option label="Dois lados" value="two.sided" />
        <option label="Primeiro é maior" value="greater" />
        <option label="Segundo é maior" value="less" />
</radio>
```

Tudo o que falta nos elementos da aba 'Configurações básicas' é a caixa de seleção para amostras pareadas e a estruturação de todos esses elementos em linhas e colunas:

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

A `rk.XML.cbox()` é uma rara exceção onde o nome da função não contém o nome completo da tag, para economizar digitação para este elemento frequentemente usado. Isto é o que `basic.settings` agora contém:

```
<row id="row_vTFSPP10TF">
        <varselector id="vars" />
        <column id="clm_vrsTFSPP10">
                <varslot id="vrsl_compare" label="comparar" source="vars"  ↩
                    types="number" required="true" />
                <varslot id="vrsl_against" label="contra" i18n_context=" ↩
                    comparar contra" source="vars" types="number" required=" ↩
                    true" />
                <radio id="rad_usngtsth" label="usar hipótese de teste">
                        <option label="Dois lados" value="two.sided" />
                        <option label="Primeiro é maior" value="greater" />
                        <option label="Segundo é maior" value="less" />
                </radio>
                <checkbox id="chc_Pardsmpl" label="Amostras pareadas" value ↩
                    ="1" value_unchecked="0" />
        </column>
</row>
```

De maneira semelhante, as próximas linhas criarão objetos R para os elementos da aba 'Opções', introduzindo funções para caixas de seleção, molduras e espaços:

```
check.eqvar <- rk.XML.cbox("assumir variâncias iguais", value="1", un.value ↩
    ="0")
conf.level <- rk.XML.spinbox("nível de confiança", min=0, max=1, initial ↩
    =0.95)
check.conf <- rk.XML.cbox("imprimir intervalo de confiança", val="1", chk= ↩
    TRUE)
conf.frame <- rk.XML.frame(conf.level, check.conf, rk.XML.stretch(), label ↩
    ="Confidence Interval")
```

Agora, tudo o que precisamos fazer é juntar os objetos em um tabbook e colocá-lo em uma seção de diálogo:

```
full.dialog <- rk.XML.dialog(
        label="Teste t com duas variáveis",
        rk.XML.tabbook(tabs=list("Configurações básicas"=basic.settings, " ↩
            Opções"=list(check.eqvar, conf.frame)))
)
```

Também podemos criar a seção do assistente com suas duas páginas usando os mesmos objetos, portanto, seus IDs serão extraídos para as tags **<copy>**:

```
full.wizard <- rk.XML.wizard(
                label="Teste t com duas variáveis",
                rk.XML.page(
                        rk.XML.text("Como primeiro passo, selecione duas  ↩
                            variáveis que você deseja comparar.
                                E especifique qual você teoriza que é maior ↩
                                    . Selecione dois lados,
                                se sua teoria não te informa qual variável  ↩
                                    é maior."),
                        rk.XML.copy(basic.settings)),
                rk.XML.page(
                        rk.XML.text("Abaixo estão algumas opções avançadas. ↩
                             É geralmente seguro não considerar que as
                                variáveis tenha variância igual. Uma  ↩
                                    correção apropriada será aplicada então.
                                Escolher \"assumir variâncias iguais\" pode ↩
                                     aumentar a força do teste, no entanto ↩
                                    ."),
                        rk.XML.copy(check.eqvar),
                        rk.XML.text("Algumas vezes é útil obter uma  ↩
                            estimativa do intervalo de confiança da
                                diferença em significado. Abaixo você pode  ↩
                                    especificar se um deve ser exibido, e
                                qual nível de confiança deve ser aplicado  ↩
                                    (95% corresponde a 5% de nível de
                                significância)."),
                        rk.XML.copy(conf.frame)))
```

Isso é tudo para a GUI. O documento global será combinado no final por `rk.plugin.skeleton()`.

15.2.2 Código JavaScript

Até agora, usar o pacote rkwarddev pode não parecer ter ajudado muito. Isso vai mudar agora.

Primeiramente, assim como não precisamos nos preocupar com os IDs dos elementos ao definir o layout da GUI, também não precisamos nos preocupar com os nomes das variáveis JavaScript na próxima etapa. Se você quiser mais controle, pode escrever código JavaScript puro e colá-lo no arquivo gerado. Mas provavelmente é muito mais eficiente fazer isso da maneira do rkwarddev.

O mais notável é que você não precisa definir nenhuma variável manualmente, pois `rk.plugin.skeleton()` pode analisar seu código XML e definir automaticamente todas as variáveis que você provavelmente precisará -- por exemplo, você não se preocuparia em incluir uma caixa de seleção se não usar seu valor ou estado posteriormente. Assim, podemos começar a escrever o código JS que gera o código R imediatamente.

> DICA
> A função `rk.JS.scan()` também pode analisar arquivos XML em busca de variáveis.

O pacote possui algumas funções para construções de código JS que são comumente usadas em plugins RKWard, como a função `echo()` ou as condições `if() {...} else {...}`. Existem algumas diferenças entre JS e R, por exemplo, para `paste()` em RKWard você usa a vírgula para concatenar strings de caracteres, enquanto para `echo()` em JS você usa '+', e as linhas devem terminar com um ponto e vírgula. Ao usar as funções R, você pode praticamente esquecer essas diferenças e continuar escrevendo código R.

Essas funções podem receber diferentes classes de objetos de entrada: texto simples, objetos R com código XML como o acima ou, por sua vez, resultados de outras funções JS do pacote. No final, você sempre chamará `rk.paste.JS()`, que se comporta de forma semelhante a `paste()`, mas, dependendo dos objetos de entrada, irá substituí-los por seus IDs XML, nomes de variáveis JavaScript ou até mesmo blocos de código JavaScript completos.

Para o exemplo do teste t, precisamos de dois objetos JS: um para calcular os resultados e outro para imprimi-los na função `printout()`.

```
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)
```

Como você pode ver, rkwarddev também fornece uma implementação de R da função `echo()`. Ela retorna exatamente uma string de um caractere com uma versão JS válida de si mesma. Você também pode notar que todos os objetos R aqui são aqueles que criamos anteriormente. Eles serão automaticamente substituídos por seus nomes de variáveis, então isso deve ser bastante intuitivo. Sempre que você precisar apenas dessa substituição, a função `id()` pode ser usada, que também retornará exatamente uma string de um caractere de todos os objetos que lhe foram fornecidos (você poderia dizer que ela se comporta como `paste()` com uma substituição de objeto muito específica).

A função `js()` é um wrapper que permite usar as condições **if(){...} else {...}** do R como você já está acostumado. Elas serão traduzidas diretamente para código JS. Também preserva alguns operadores como **<**, **>=** ou **||**, para que você possa comparar logicamente seus objetos R sem a necessidade de aspas na maioria das vezes. Vejamos o objeto 'JS.calc' resultante, que agora contém uma string de caracteres com o seguinte conteúdo:

```
echo("res <- t.test (x=" + vrslCompare + ", y=" + vrslAgainst + ",  ↩
    hypothesis=\"" + radUsngtsth + "\"");
        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");
```

> NOTA
> Alternativamente, para condições `if()` aninhadas em `js()`, você pode usar a função `ite()`, que se comporta de forma semelhante à função `ifelse()` do pacote R. No entanto, instruções condicionais construídas usando `ite()` geralmente são mais difíceis de ler e devem ser substituídas por `js()` sempre que possível.

15.2.3 Mapa de plugin

Esta seção é muito curta: Não precisamos escrever um `.pluginmap` pois ele pode ser gerado automaticamente por `rk.plugin.skeleton()`. A hierarquia do menu pode ser especificada através da opção *`pluginmap`*:

```
[...]
        pluginmap=list(
                name="Teste t com duas variáveis",
                hierarchy=list("análise", "significado", "Teste t"))
        [...]
```

15.2.4 Página de ajuda

Esta seção também é muito curta: `rk.plugin.skeleton()` não consegue escrever uma página de ajuda completa com as informações que possui. Mas ela consegue analisar o documento XML em busca de elementos que provavelmente merecem uma entrada na página de ajuda e criar automaticamente um modelo de página de ajuda para o nosso plugin. Tudo o que precisamos fazer depois é escrever algumas linhas para cada seção listada.

> DICA
> A função `rk.rkh.scan()` também pode analisar arquivos XML para criar um esqueleto de arquivo de ajuda.

15.2.5 Gerar arquivos do plugin

Agora vem a etapa final, na qual entregaremos todos os objetos gerados para `rk.plugin.skeleton()`:

```
plugin.dir <- rk.plugin.skeleton("Teste t",
        xml=list(
                dialog=full.dialog,
                wizard=full.wizard),
        js=list(
                results.header="Teste t com duas variáveis",
                calculate=JS.calc,
                printout=JS.print),
        pluginmap=list(
                name="Two Variable t-Test",
```

```
                hierarchy=list("análise", "significados", "Teste t")),
        load=TRUE,
        edit=TRUE,
        show=TRUE)
```

Os arquivos serão criados em um diretório temporário por padrão. As três últimas opções não são necessárias, mas são muito úteis: *load=TRUE* adicionará automaticamente o novo plugin à configuração do RKWard (já que ele está em um diretório temporário e, portanto, deixará de existir quando o RKWard for fechado, ele será removido automaticamente pelo RKWard na próxima inicialização), *edit=TRUE* abrirá todos os arquivos criados para edição nas abas do editor do RKWard e *show=TRUE* tentará iniciar o plugin diretamente, para que você possa ver como ele é sem precisar clicar. Você pode considerar adicionar *overwrite=TRUE* se você for executar seu script repetidamente (por exemplo, após alterações no código), pois, por padrão, nenhum arquivo será sobrescrito.

O objeto de resultado 'plugin.dir' contém o caminho para o diretório no qual o plugin foi criado. Isso pode ser útil em combinação com a função `rk.build.package()`, para construir um pacote Rreal para compartilhar seu plugin com outras pessoas -- por exemplo, enviando-o para a equipe de desenvolvimento da RKWard para ser adicionado ao nosso repositório de plugins.

15.2.6 O script completo

Para recapitular tudo o que foi dito acima, aqui está o script completo para criar o exemplo de teste t funcional. Além do código já explicado, ele também carrega o pacote, se necessário, e usa o ambiente `local()`, portanto, todos os objetos criados não ficarão no seu espaço de trabalho atual (exceto o arquivo 'plugin.dir'):

```
require(rkwarddev)

local({
        variables <- rk.XML.varselector(id.name="vars")
        var.x <- rk.XML.varslot("comparar", source=variables, types="number ↩
            ", required=TRUE)
        var.y <- rk.XML.varslot("contra", source=variables, types="number", ↩
             required=TRUE)
        test.hypothesis <- rk.XML.radio("usar hipótese de teste",
                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("Amostra pareadas", 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("assumir variâncias iguais", value="1",  ↩
            un.value="0")
        conf.level <- rk.XML.spinbox("nível de confiança", min=0, max=1,  ↩
            initial=0.95)
        check.conf <- rk.XML.cbox("imprimir intervalo de confiança", val ↩
            ="1", chk=TRUE)
        conf.frame <- rk.XML.frame(conf.level, check.conf, rk.XML.stretch() ↩
            , label="Intervalo de confiança")

        full.dialog <- rk.XML.dialog(
                label="Teste t com duas variáveis",
```

```
        rk.XML.tabbook(tabs=list("Configurações básicas"=basic. ↩
            settings, "Opções"=list(check.eqvar, conf.frame)))
)

full.wizard <- rk.XML.wizard(
        label="Teste t com duas variáveis",
        rk.XML.page(
                rk.XML.text("Como primeiro passo, selecione duas  ↩
                    variáveis que você deseja comparar.
                        E especifique qual você teoriza que é maior ↩
                            . Selecione dois lados,
                        se sua teoria não te informa qual variável  ↩
                            é maior."),
                        rk.XML.copy(basic.settings)),
                rk.XML.page(
                rk.XML.text("Abaixo estão algumas opções avançadas. ↩
                     É geralmente seguro não considerar que as
                        variáveis tenha variância igual. Uma  ↩
                            correção apropriada será aplicada então.
                        Escolher \"assumir variâncias iguais\" pode ↩
                             aumentar a força do teste, no entanto ↩
                            ."),
                        rk.XML.copy(check.eqvar),
                rk.XML.text("Algumas vezes é útil obter uma  ↩
                    estimativa do intervalo de confiança da
                        diferença em significado. Abaixo você pode  ↩
                            especificar se um deve ser exibido, e
                        qual nível de confiança deve ser aplicado  ↩
                            (95% corresponde a 5% de nível de
                        significância)."),
                        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="Teste t com duas variáveis",
                calculate=JS.calc,
```

```
                        printout=JS.print),
                pluginmap=list(
                        name="Two Variable t-Test",
                        hierarchy=list("análise", "significados", "Teste t ↩
                            ")),
                load=TRUE,
                edit=TRUE,
                show=TRUE,
                overwrite=TRUE)
})
```

15.3 Adicionar páginas de ajuda

Se você deseja escrever uma página de ajuda para o seu plugin, a maneira mais direta de fazer isso é adicionar as instruções específicas diretamente às definições dos elementos XML aos quais elas pertencem:

```
variables <- rk.XML.varselector(
  id.name="vars",
  help="Selecione o objeto de dados que gostaria de analisar.",
  component="Data"
)
```

O texto fornecido ao parâmetro *help* pode então ser obtido por `rk.rkh.scan()` e escrito na página de ajuda deste componente de plugin. Para que isso funcione tecnicamente, no entanto, `rk.rkh.scan()` deve saber quais objetos R pertencem a um componente de plugin. É por isso que você também deve fornecer o parâmetro *component* e garantir que ele seja idêntico para todos os objetos pertencentes ao mesmo componente.

Como você geralmente combinará muitos objetos em um único diálogo e também poderá querer reutilizar objetos como o **<varslot>** em vários componentes de seus plugins, é possível definir globalmente um componente com a função `rk.set.comp()`. Se definida, presume-se que todos os objetos subsequentes usados em seu script pertencem a esse componente específico, até que `rk.set.comp()` seja chamada novamente com um nome de componente diferente. Você pode então omitir o parâmetro *component*:

```
rk.set.comp("Data")
variables <- rk.XML.varselector(
  id.name="vars",
  help="Selecione o objeto de dados que gostaria de analisar."
)
```

Para adicionar seções globais como **<summary>** ou **<usage>** à página de ajuda, você usa funções como `rk.rkh.summary()` ou `rk.rkh.usage()` conforme necessário. Os resultados dessas funções são então usados para definir os elementos da lista, como *summary* ou *usage*, no *rkh* parâmetro de `rk.plugin.component()`/`rk.plugin.skeleton()`.

15.4 Traduzir plugins

O pacote rkwarddev é capaz de produzir plugins externos com suporte completo a internacionalização (i18n). Por exemplo, todas as funções relevantes que geram objetos XML oferecem um parâmetro opcional para especificar *i18n_context* ou *noi18n_label*:

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

Os exemplos acima produzem resultados como este:

```
# varComment
<!-- i18n: Seletor de variável principal -->
  <varselector id="vars" />

# varContext
<varselector id="vars" i18n_context="Seletor de variável principal" />

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

Há também suporte para código JS traduzível. Na verdade, o pacote tenta adicionar chamadas `i18n()` por padrão em locais onde isso é geralmente útil. A função `rk.JS.header()` é um bom exemplo:

```
jsHeader <- rk.JS.header("Testar resultados")
```

Isso gera o seguinte código JS:

```
new Header(i18n("Testar resultados")).print();
```

Mas você também pode marcar manualmente strings em seu código JS como traduzíveis, usando a função `i18n()` da mesma forma que faria se escrevesse o arquivo JS diretamente.

Apêndice A

Referência

A.1 Tipos de propriedades/Modificadores

Em alguns trechos desta introdução, mencionamos as 'propriedades' de elementos da GUI ou outros. Na verdade, existem vários tipos diferentes de propriedades. Normalmente, você não precisa se preocupar com isso, pois pode usar o bom senso para conectar qualquer propriedade a qualquer outra. No entanto, internamente, existem diferentes tipos de propriedades. Isso é importante para a recuperação de valores especiais no modelo JS. Nas instruções getString(″id″)/getBoolean(″id″)/getList(″id″), você também pode especificar alguns dos chamados 'modificadores', como este: `getString(˜id.modificador˜)`. Esse modificador afetará a forma como o valor é impresso. Continue lendo para ver a lista de propriedades e os modificadores que cada uma delas disponibiliza:

Propriedades de string

O tipo de propriedade mais simples, usado simplesmente para armazenar um trecho de texto. Modificadores:

Sem modificador (″″)
A string conforme definida/configurada.

quoted
A string em formato de aspas (adequada para ser passada para R como caractere).

Propriedades boleanas

Propriedades que podem ser ativadas ou desativadas, verdadeiras ou falsas. Por exemplo, as propriedades criadas por tags <convert>, bem como a propriedade que acompanha uma <checkbox> (veja abaixo). Os seguintes valores serão retornados de acordo com o modificador fornecido:

Sem modificador (″″)
Por padrão, a propriedade retornará 1 se for verdadeira e 0 caso contrário. A maneira recomendada de obter valores booleanos é usando `getBoolean()`. Observe que, para `getString()`, a string ″0″ será retornada quando a propriedade for falsa. Essa string seria avaliada como verdadeira, e não como falsa, em JavaScript.

″labeled″
Retorna a string ″true″ quando verdadeiro, ″false″ quando falso, ou quaisquer strings personalizadas que tenham sido especificadas (normalmente em uma <checkbox>).

″true″
Retorna a string como se a propriedade fosse verdadeira, mesmo que seja falsa.

"false"
Retorna a string como se a propriedade fosse falsa, mesmo que seja verdadeira.

"not"
Isso na verdade retorna outra propriedade booleana, que é o inverso da atual (isto é, falso se verdadeiro, verdadeiro se falso)

"numeric"
Obsoleto, fornecido para compatibilidade com versões anteriores. Equivalente a nenhum modificador "". Retorna "1" se a propriedade for verdadeira ou "0" se for falsa.

Propriedades de inteiro

Uma propriedade projetada para armazenar um valor inteiro (mas, é claro, ainda retorna uma string de caracteres numéricos para o modelo JS). Não aceita nenhum modificador. Usada em <spinbox>es (veja abaixo)

Propriedades de número real

Uma propriedade projetada para armazenar um valor numérico real (mas, é claro, ainda retorna uma string de caracteres numéricos para o modelo JS). Usada em <spinbox>es (veja abaixo)

Sem modificador ("")
Para `getValue() / getString()`, isso retorna o mesmo que "formatted". Em versões futuras, será possível obter uma representação numérica.

"formatted"
Retorna o número formatado (como uma string).

Propriedades RObject

Uma propriedade projetada para selecionar um ou mais objetos R. Usada principalmente em varselectors e varslots. Os seguintes valores serão retornados de acordo com o modificador fornecido:

Sem modificador ("")
Por padrão, a propriedade retornará o nome completo do objeto selecionado. Se mais de um objeto for selecionado, os nomes dos objetos serão separados por quebras de linha ("\n").

"shortname"
Semelhante ao anterior, mas retorna apenas o(s) nome(s) abreviado(s) do(s) objeto(s). Por exemplo: um objeto dentro de uma lista receberia apenas o nome que possui dentro da lista, sem o nome da lista.

"label"
Semelhante ao anterior, mas retorna o(s) rótulo(s) do(s) objeto(s) RKWard (se nenhum rótulo estiver disponível, isso é o mesmo que nome curto)

Propriedades de lista de strings

Esta propriedade contém uma lista de strings.

Sem modificador ("")
Para `getValue()/getString()`, isso retorna todas as strings separadas por "\n". Quaisquer caracteres "\n" em cada item são escapados como literalmente "\n". No entanto, o uso recomendado é buscar o valor com `getList()`, que retornará uma matriz de strings.

"joined"
Retorna a lista como uma única string, com os itens unidos por "\n". Em contraste com a ausência de modificador (""), as strings individuais _não_ são escapadas.

Propriedades de código

Uma propriedade mantida por plugins que geraram código. Isso é importante para incorporar plugins, a fim de incorporar o código gerado pelo plugin incorporado no código gerado pelo plugin incorporador (de nível superior). Os os seguintes valores serão retornados de acordo com o modificador fornecido:

Sem modificador ("")
Retorna o código completo, ou seja, as seções ″preprocess″, ″calculate″, ″printout″ e (mas não ″preview″) concatenadas em uma única string.

″preprocess″
Retorna apenas a seção de pré-processamento do código

″calculate″
Retorna apenas a seção de cálculo do código

″printout″
Retorna apenas a seção de impressão do código

″preview″
Retorna a seção de pré-visualização do código

A.2 Elementos de uso geral para serem utilizados em qualquer arquivo XML (`.xml`, `.rkh`, `.pluginmap`)

<snippets>
Permitido como filho direto do nó <document> e somente lá. Deve ser colocado próximo ao início do arquivo. Consulte a seção sobre como usar snippets. Apenas um elemento <snippets> pode estar presente. Opcional, sem atributos.

<snippet>
Define um único trecho. Permitido apenas como filho direto do elemento <snippets>. Atributos:

<id>
Uma string identificadora para o trecho de código. Obrigatória.

<insert>
Insere o conteúdo de um <snippet>. Permitido em qualquer lugar. Atributos:

<snippet>
A string de identificação do trecho de código a ser inserido. Obrigatório.

<include>
Inclui o conteúdo de outro arquivo XML (tudo dentro do elemento <document> desse arquivo). Permitido em qualquer lugar. Atributos:

<file>
O nome do arquivo, relativo ao diretório onde o arquivo atual se encontra. Obrigatório.

A.3 Elementos a serem usados na descrição XML do plugin

As propriedades mantidas pelos elementos são listadas em uma seção separada.

A.3.1 Elementos gerais

<document>

Precisa estar presente em cada arquivo description.xml como nó raiz. Sem função especial. Sem atributos.

<about>

Informações sobre este plugin (autor, licença, etc..). Este elemento é permitido tanto no arquivo `.xml` de um plugin individual quanto nos arquivos `.pluginmap`. Consulte a referência do arquivo `.pluginmap` para obter detalhes de referência e o o capítulo sobre informações do 'about' para uma introdução.

<code>

Define onde procurar o modelo JS para o plugin. Use apenas uma vez por arquivo, como filho direto da tag document. Atributos:

file

Nome do arquivo do modelo JS, relativo ao diretório onde o arquivo xml do plugin está localizado.

<help>

Define onde procurar o arquivo de ajuda do plugin. Use apenas uma vez por arquivo, como um filho direto da tag document. Atributos:

file

Nome do arquivo de ajuda, relativo ao diretório onde o arquivo xml do plugin está localizado.

<copy>

Pode ser usado como filho (direto ou indireto) dos elementos de layout principais, isto é, <dialog> e <wizard>. Isso é usado para copiar um bloco inteiro de elementos XML 1:1. Atributos:

id

O ID a ser procurado. A tag <copy> procurará um elemento XML anterior que tenha recebido o mesmo ID e o copiará, incluindo todos os elementos descendentes.

copy_element_tag_name

Em alguns casos, você desejará uma cópia quase literal, mas alterará o nome da tag do elemento a ser copiado. O exemplo mais importante disso é quando você deseja copiar uma <tab> inteira de uma interface de diálogo para a <page> de uma interface de assistente. Nesse caso, você definiria copy_element_tag_name=″page″ para fazer essa conversão automaticamente.

A.3.2 Definições da interface

<dialog>

Define uma interface do tipo diálogo. Coloque a definição GUI dentro desta tag. Use apenas uma vez por arquivo, como filho direto da tag document. Pelo menos uma das tags ″dialog″ ou ″wizard″ é necessária para um plugin. Atributos:

label

Legenda para o diálogo

recommended

O diálogo deve ser usado como a interface ″recomendada″ (isto é, a interface que será exibida por padrão, a menos que o usuário tenha configurado o RKWard para usar uma interface específica por padrão)? Este atributo não tem efeito atualmente, pois é implicitamente ″verdadeiro″, a menos que o assistente seja recomendado.

<wizard>

Define uma interface do tipo assistente. Coloque a definição da GUI dentro desta tag. Use apenas uma vez por arquivo, como filho direto da tag document. Pelo menos uma das tags ″dialog″ ou ″wizard″ é necessária para um plugin. Aceita apenas tags <page> ou <embed> como filhos diretos. Atributos:

label

Legenda para o assistente

recommended

O assistente deve ser usado como a interface ″recomendada″ (isto é, a interface que será exibida por padrão, a menos que o usuário tenha configurado o RKWard para usar uma interface específica por padrão)? Opcional, o padrão é ″false″.

A.3.3 Elementos do layout

Todos os elementos desta seção aceitam um atributo id=″identifierstring″. Este atributo é opcional para todos os elementos. Ele pode ser usado, por exemplo, para ocultar/desativar todo o elemento de layout e todos os elementos contidos nele (consulte o capítulo lógica da GUI). A string de identificação não pode conter ″.″ (ponto) ou ″;″ (ponto e vírgula) e deve geralmente ser limitada a caracteres alfanuméricos e ao sublinhado (″_″). Apenas os atributos adicionais estão listados.

<page>

Define uma nova página dentro de um assistente. Só é permitido como filho direto de um elemento <wizard>.

<row>

Todos os filhos diretos de uma tag ″row″ serão colocados da esquerda para a direita.

<column>

Todos os filhos diretos de uma tag ″column″ serão colocados de cima para baixo.

<stretch>

Por padrão, os elementos da GUI ocupam todo o espaço disponível. Por exemplo, se você tiver duas colunas lado a lado, a da esquerda está repleta de elementos, mas a da direita contém apenas um botão de opção (**<radio>**), o controle **<radio>** se expandirá verticalmente, mesmo que não precise do espaço disponível, e ficará com uma aparência ruim. Nesse caso, você realmente deseja adicionar um espaço em branco abaixo do **<radio>**. Para isso, use o elemento <stretch>. Ele simplesmente ocupará um pouco de espaço. Não abuse desse elemento, geralmente é uma boa prática que os elementos da GUI ocupem todo o espaço disponível, apenas ocasionalmente o layout ficará espaçado. O elemento <stretch> não aceita nenhum argumento, nem mesmo um ″id″. Além disso, você não pode colocar nenhum filho dentro do elemento <stretch> (em outras palavras, você o usará apenas como ″<stretch/>″).

<frame>

Desenha uma moldura/caixa ao redor de seus filhos diretos. Pode ser usado para agrupar visualmente opções relacionadas. O layout dentro de uma moldura é de cima para baixo, a menos que você coloque uma <row> dentro dela. Atributos:

label

Legenda para a moldura (opcional)

checkable

Os frames podem ser marcados. Nesse caso, todos os elementos contidos neles serão desativados quando o frame estiver desmarcado e ativados quando estiver marcado. (opcional, o padrão é ″false″)

checked
Somente para frames marcáveis: Se o frame deve ser marcado por padrão? O padrão é ″true″. Não é interpretado para frames não marcáveis.

<tabbook>
Organiza elementos em um livro de abas. Aceita apenas tags <tab> como filhos diretos.

<tab>
Você define a página em um livro de abas. Coloque a definição da GUI para a aba dentro desta tag. Pode ser usada apenas como filha direta de uma tag <tabbook>. Em um <tabbook>, deve haver pelo menos duas abas definidas. Atributos:

label
Legenda para a página da aba (obrigatória)

<text>
Exibe o texto contido nesta tag na GUI. Algumas marcações simples em estilo HTML são suportadas (principalmente <b>, <i>, <p> e
). No entanto, mantenha a formatação ao mínimo. Inserir uma linha completamente vazia adiciona uma quebra de linha. Atributos:

type
Tipo de texto. Um dos seguintes: ″normal″, ″aviso″ ou ″erro″. Isso influencia a aparência do texto (opcional, o padrão é normal)

A.3.4 Elementos ativos

Todos os elementos desta seção aceitam um atributo id=″identifierstring″. Este atributo é obrigatório para todos os elementos. Somente os atributos adicionais estão listados. A string de identificação não pode conter ″.″ (pontos).

<varselector>
Fornece uma lista de objetos disponíveis a partir da qual o usuário pode selecionar um ou mais. Requer um ou mais <varslots> como contrapartida para ser útil. Atributos:

label
Legenda para o seletor de variáveis (opcional, o padrão é ″Selecionar variável(eis)″)

<varslot>
Usado em conjunto com um ″seletor de variáveis″ para permitir que o usuário selecione uma ou mais variáveis. Atributos:

label
Legenda para o varslot (recomendado, o padrão é ″Variável:″)

source
O seletor de variáveis para obter a seleção (obrigatório, a menos que você se conecte manualmente ou usando source_property)

source_property
Uma propriedade arbitrária da qual copiar valores quando o botão de seleção é clicado. Se especificada, esta propriedade substitui o atributo ″source″.

required
Se - para submeter o código - é necessário que este varslot contenha um valor válido. Consulte required-property (opcional, o padrão é falso)

multi
Indica se o varslot contém apenas um objeto (padrão, ″false″) ou vários objetos

allow_duplicates
Se o varslot pode aceitar apenas objetos únicos (padrão, ″false″) ou se o mesmo objeto pode ser adicionado várias vezes.

min_vars
: Só faz sentido se multi="true": Número mínimo de variáveis a serem selecionadas para que a seleção seja considerada válida (opcional, o padrão é "1")

min_vars_if_any
: Só faz sentido se multi="true": Alguns espaços para variáveis podem ser considerados válidos, se, por exemplo, o espaço para variáveis estiver vazio ou contiver pelo menos dois valores. Isso especifica quantas variáveis devem ser selecionadas, se houver alguma (2 no exemplo). (opcional, o padrão é "1")

max_vars
: Só faz sentido se multi="true": Número máximo de variáveis a selecionar (opcional, o padrão é "0", o que significa que não há máximo)

classes
: Se você especificar um ou mais nomes de classe R (separados por espaços (" ")), aqui, o varslot aceitará apenas objetos pertencentes a essas classes (opcional, *use com muita cautela*, o usuário não deve ser impedido de fazer escolhas válidas, e R possui *muitas* classes diferentes)

types
: Se você especificar um ou mais tipos de variáveis (separados por espaços (" ")), aqui, o varslot aceitará apenas objetos desses tipos. Os tipos válidos são "unknown", "number", "string", "factor", "invalid". (Opcional, *use com muita cautela*, o usuário não deve ser impedido de fazer escolhas válidas, e o RKWard nem sempre sabe o tipo de uma variável)

num_dimensions
: O número de dimensões que um objeto precisa ter. "0" (o padrão) Significa que qualquer número de dimensões é aceitável. (opcional, o padrão é "0")

min_length
: O comprimento mínimo que um objeto precisa ter para ser aceitável. (opcional, o padrão é "0")

max_length
: O comprimento máximo que um objeto precisa ter para ser aceitável. (opcional, o padrão é o maior número inteiro representável no sistema)

<valueselector>
: Fornece uma lista de strings disponíveis (não objetos R) para serem selecionadas em um ou mais <valueslots> acompanhantes. As opções de string podem ser definidas usando tags <option> como filhos diretos (veja abaixo) ou definidas usando propriedades dinâmicas. properties. Atributos:

label
: Legenda para o seletor de valores (opcional, o padrão é nenhuma legenda)

<valueslot>
: Usado em conjunto com um <valueselector> para permitir que o usuário selecione um ou mais itens de string. Este elemento é praticamente idêntico a <varslot>, e compartilha os mesmos atributos, exceto aqueles que se referem às propriedades dos itens aceitáveis (ou seja, classes, tipos, num_dimensions, min_length, max_length).

<radio>
: Define um grupo de botões exclusivos de rádio (apenas um pode ser selecionado por vez). Requer pelo menos duas tags <option> como filhos diretos. Nenhuma outra tag é permitida como filho. Atributos:

label
: Legenda para o controle remoto (recomendado, o padrão é "Selecione uma opção:")

<dropdown>

Define um grupo de opções, das quais apenas uma pode ser selecionada por vez, usando uma lista suspensa. Isso é funcionalmente equivalente a um <radio>, mas tem uma aparência diferente. Requer pelo menos duas tags <option> como filhos diretos. Nenhuma outra tag é permitida como filha. Atributos:

label
Legenda para a lista suspensa (recomendado, o padrão é ″Selecione uma opção:″)

<select>

Fornece uma lista de strings disponíveis a partir das quais o usuário pode selecionar um número arbitrário. As opções de string podem ser definidas usando tags <option> como filhos diretos (veja abaixo) ou definidas usando properties dinâmicas. Atributos:

label
Legenda para o <select> (opcional, o padrão é nenhuma legenda)

single
Se definido como verdadeiro, apenas um único valor poderá ser selecionado, em vez de vários valores simultaneamente. (booleano, padrão é falso)

<option>

Só pode ser usado como filho direto de um elemento <radio>, <dropdown>, <valueselector> ou <select>. Representa uma opção selecionável em um controle de rádio ou lista suspensa. Como os elementos <option> são sempre parte de um dos elementos de seleção, eles normalmente não têm um ″id″ próprio, mas veja abaixo. Atributos:

label
Legenda para a opção (obrigatória)

value
O valor da string que o elemento pai retornará se esta opção estiver marcada/selecionada (obrigatório)

checked
Indica se a opção deve ser marcada/selecionada por padrão ″true″ ou ″false″. Em um <radio> ou <dropdown>, apenas uma opção pode ser definida como `checked="true"`, e se nenhuma opção for definida como marcada, a primeira opção no elemento pai será marcada/selecionada automaticamente. Em um <select>, qualquer número de opções pode ser definido como marcado. (opcional, padrão ″false″)

id
Especificar o ″id″ do parâmetro para os elementos <option> é opcional (e, na verdade, é recomendado não definir um ″id″, a menos que você realmente precise de um). No entanto, especificar um ″id″ permitirá que você habilite/desabilite <option> dinamicamente, conectando-se à propriedade booleana `id_of_radio.id_of_optionX.enabled`. Atualmente, isso funciona apenas para opções dentro de elementos <radio> ou <dropdown>. As opções <valueselector> e <select> não suportam IDs no momento.

<checkbox>

Define uma caixa de seleção, isto é, uma única opção que pode ser ativada ou desativada. Atributos:

label
Legenda para a caixa de seleção (obrigatória)

value
O valor que a caixa de seleção retornará se estiver marcada (obrigatório)

value_unchecked
O valor que será retornado se a caixa de seleção não estiver marcada (opcional, o padrão é ″″, isto é, uma string vazia)

checked
Indica se a opção deve ser marcada por padrão como ″verdadeiro″ ou ″falso″ (opcional, padrão ″falso″)

<frame>

O elemento frame é geralmente usado como um elemento de layout puro e está listado na seção sobre elementos de layout. No entanto, ele também pode ser marcado, funcionando assim como uma caixa de seleção simples. Ao mesmo tempo,

<input>

Define um campo de entrada de texto livre. Atributos:

label
Legenda para o campo de entrada (obrigatória)

initial
Texto inicial do campo de texto (opcional, o padrão é ″″, ou seja, uma string vazia)

size
Uma das opções ″small″ (pequeno), ″medium″ (médio) ou ″large″ (grande). ″Grande″ define um campo de entrada com várias linhas, ″pequeno″ e ″médio″ são campos de linha única (opcional, o padrão é ″medium″)

required
Se - para submeter o código - é necessário que esta entrada não esteja vazia. Consulte required-property (opcional, o padrão é falso)

<matrix>

Uma tabela para inserir dados matriciais (ou vetores) na GUI.

> NOTA
> Este elemento de entrada *não* está otimizado para inserir/editar grandes quantidades de dados. Embora não haja um limite estrito para o tamanho de uma <matrix>, em geral, ela não deve exceder cerca de dez linhas/colunas. Se você espera dados maiores, permita que os usuários os selecionem como um objeto R (o que pode ser uma boa ideia como opção alternativa, em quase *todos* casos em que você usa um elemento de matriz).

Atributos:

label
Legenda para a tabela (obrigatória)

mode
Um dos seguintes valores: ″integer″ (inteiro), ″real″ ou ″string″. O tipo de dados que será aceito na tabela (obrigatório)

min
Valor mínimo aceitável (para matrizes do tipo ″inteiro″ ou ″real″) (opcional, o padrão é o menor valor representável)

max
Valor máximo aceitável (para matrizes do tipo ″inteiro″ ou ″real″) (opcional, o padrão é o maior valor representável)

allow_missings
Indica se valores ausentes (vazios) são permitidos na matriz. Isso está implícito para matrizes ou modo ″string″ (opcional, o padrão é falso).

allow_user_resize_columns
Quando definido como verdadeiro, o usuário pode adicionar colunas digitando nas células mais à direita (inativas) (opcional, o padrão é verdadeiro).

allow_user_resize_rows
Quando definido como verdadeiro, o usuário pode adicionar linhas digitando nas células mais inferiores (inativas) (opcional, o padrão é verdadeiro).

rows
Número de linhas na matriz. Não tem efeito para allow_user_resize_rows=″true″. ″

> NOTA
> Isso também pode ser controlado definindo a propriedade ″rows″.

(opcional, o padrão é 2).

columns
Número de colunas na matriz. Não tem efeito para allow_user_resize_columns=″true″.

> NOTA
> Isso também pode ser controlado definindo a propriedade ″columns″.

(opcional, o padrão é 2).

min_rows
Número mínimo de linhas na matriz. A matriz não será reduzida abaixo desse tamanho. (opcional, o padrão é 0; veja também: `allow_missings`.).

min_columns
Número mínimo de colunas na matriz. A matriz não será reduzida abaixo desse tamanho. (opcional, o padrão é 0; veja também: `allow_missings`.).

fixed_height
Força o elemento da GUI a permanecer em sua altura inicial. Não use em combinação com matrizes, onde o número de linhas pode mudar de alguma forma. Útil, especialmente ao criar um elemento de entrada vetorial (columns=″1″). Com esta opção definida como verdadeira, nenhuma barra de rolagem horizontal será exibida, mesmo que a matriz exceda a largura disponível (pois isso afetaria a altura). (opcional, o padrão é falso).

fixed_width
Ligeiramente mal nomeado: Assume que a contagem de colunas não mudará. A última (ou normalmente a única) coluna será esticada para ocupar a largura disponível. Não use em combinação com matrizes, onde o número de colunas pode mudar de alguma forma. Útil, especialmente ao criar um elemento de entrada vetorial (linhas=″1″). (opcional, o padrão é falso).

horiz_headers
Textos a serem usados para o cabeçalho horizontal, separados por ″;″. O cabeçalho ficará oculto se definido como ″″. (opcional, o padrão é o número da coluna).

vert_headers
Strings a serem usadas para o cabeçalho vertical, separadas por ″;″. O cabeçalho ficará oculto se definido como ″″. (opcional, o padrão é o número da linha).

<optionset>

Uma interface de usuário para repetir um conjunto de opções para um número arbitrário de itens (introdução a conjuntos de opções). Atributos:

min_rows
Se especificado, o conjunto será marcado como inválido, a menos que tenha pelo menos este número de linhas (opcional, inteiro).

min_rows_if_any
Semelhante a min_rows, mas só será testado se houver pelo menos uma linha. (opcional, inteiro).

max_rows
Se especificado, o conjunto será marcado como inválido, a menos que tenha no máximo este número de linhas (opcional, inteiro).

keycolumn
ID da coluna que atuará como coluna-chave. Um conjunto de opções com uma coluna-chave (válida) atenderá como um conjunto de opções ″controlado″. Um conjunto de opções sem coluna-chave permitirá a inserção/remoção manual de itens. A coluna-chave deve ser marcada como externa. (opcional, o padrão é nenhuma coluna-chave).

Elementos-filho:

<optioncolumn>
Declara uma coluna de opções do conjunto. Para cada valor que você deseja obter do conjunto de opções, você deve declarar um <optioncolumn> separado. Atributos:

id
O ID da coluna de opções (obrigatório, string).

external
Defina como verdadeiro se a coluna de opções for controlada externamente ao conjunto de opções. (opcional, booleano, o padrão é falso).

label
Se fornecida, a coluna de opções será exibida em uma coluna com esse rótulo. (opcional, texto, o padrão é não exibir).

connect
A propriedade à qual esta coluna de opções será conectada, fornecida como id dentro da área de <content>. Para <optioncolumn> externas, o valor correspondente será definido como o valor definido externamente. Para <optioncolumn> regulares (não externas), a linha correspondente da propriedade da <optioncolumn> será definida quando a propriedade for alterada dentro da área de conteúdo. (opcional, string, o padrão é não conectado).

default
Apenas para colunas externas: O valor a ser assumido para esta coluna, caso nenhum valor seja conhecido para uma entrada. Raramente útil. (Opcional, o padrão é uma string vazia)

<content>
Declare o conteúdo/interface do usuário do conjunto. Sem atributos. Todos os elementos ativos, passivos e de layout usuais são permitidos como elementos childname. Além disso, em versões anteriores do RKWard (até a 0.6.3), o elemento filho especial **<optiondisplay>** era permitido. Isso está obsoleto na versão 0.6.4 do RKWard e deve ser simplesmente removido dos plugins existentes.

<logic>
Especificação opcional da lógica da interface do usuário para aplicar *dentro* da região de conteúdo do conjunto de opções. Consulte a referência sobre <logic>

<browser>
Um elemento projetado para selecionar um único nome de arquivo (ou nome de diretório). Observe que este campo aceita qualquer string, embora seja destinado ao uso exclusivo de arquivos.

label
Legenda para o navegador (opcional, o padrão é ″Digite o nome do arquivo″)

initial
Texto inicial do navegador (opcional, o padrão é ″″, ou isto é, uma string vazia)

type
Uma das opções ″file″, ″dir″ ou ″savefile″. Para selecionar um arquivo existente, um diretório existente ou um arquivo inexistente, respectivamente (opcional, o padrão é ″file″)

allow_urls
Se URLs (não locais) podem ser selecionadas (opcional, o padrão é ″false″)

filter
: Filtro de tipo de arquivo, por exemplo (″*.txt *.csv″ para arquivos .txt e .csv). Uma entrada separada para ″Todos os arquivos″ é adicionada automaticamente (opcional, o padrão é ″″, ou seja, Todos os arquivos)

required
: Se - para submeter o código - é necessário que o campo não esteja vazio. Observe que isso não significa necessariamente que o nome do arquivo selecionado seja válido. Consulte required-property (opcional, o padrão é verdadeiro)

<saveobject>

Um elemento projetado para selecionar o nome de um objeto R para salvar (isto é, geralmente não existente, ao contrário de um varslot):

label
: Legenda para o campo de entrada (opcional, o padrão é ″Salvar em:″)

initial
: Texto inicial da entrada (opcional, o padrão é ″my.data″)

required
: Para submeter o código, é necessário que o campo contenha um nome de objeto permitido? Consulte a propriedade required-property (opcional, o padrão é verdadeiro)

checkable
: Em muitos casos de uso, salvar em um objeto R é opcional. Nesses casos, uma caixa de seleção pode ser integrada ao elemento saveobject usando este atributo. Quando definido como verdadeiro, o objeto salvo será ativado/desativado pela caixa de seleção. Consulte a active-property do elemento saveobject (opcional, o padrão é falso)

checked
: Somente para elementos saveobject verificáveis: Se o controle está marcado/habilitado por padrão (opcional, o padrão é falso)

<spinbox>

Uma caixa de seleção numérica na qual o usuário pode escolher um valor numérico, usando entrada direta pelo teclado ou pequenas setas para cima/para baixo. Atributos:

label
: Legenda para a caixa de seleção (recomendado, o padrão é ″Digite o valor:″)

min
: O menor valor que o usuário pode inserir na caixa de seleção (opcional, o padrão é o menor valor tecnicamente representável na caixa de seleção)

max
: O maior valor que o usuário pode inserir na caixa de seleção (opcional, o padrão é o maior valor tecnicamente representável na caixa de seleção)

initial
: O valor inicial exibido na caixa de seleção (opcional, o padrão é ″0″)

type
: Uma das opções ″real″ ou ″integer″ (inteiro). Indica se a caixa de seleção aceitará números reais ou apenas números inteiros (opcional, o padrão é ″real″)

default_precision
: Só faz sentido se a caixa de seleção for do tipo ″real″. Especifica o número padrão de casas decimais exibidas na caixa de seleção (apenas essa quantidade de zeros à direita será exibida). Quando o usuário pressionar as setas para cima/para baixo, esse número de casas decimais será alterado. O usuário ainda poderá inserir valores com uma precisão maior, no entanto (veja abaixo) (opcional, o padrão é ″2″)

max_precision
: O número máximo de dígitos que podem ser representados de forma significativa (opcional, o padrão é ″8″)

<formula>

Este elemento avançado permite ao usuário selecionar uma fórmula/conjunto de interações a partir de variáveis selecionadas. Por exemplo, para um GLM, este elemento pode ser usado para permitir que o usuário especifique os termos de interação no modelo. Atributos:

fixed_factors

O ID do varslot que contém os fatores fixos selecionados (obrigatório)

dependent

O ID do varslot que contém a variável dependente selecionada (obrigatório)

<embed>

Incorpora um plugin diferente no atual (consulte o capítulo sobre incorporação). Atributos:

component

O nome registrado do componente a ser incorporado (consulte o capítulo sobre registro de componentes) (obrigatório)

as_button

Se definido como ″true″, apenas um botão será colocado na GUI incorporada. A GUI incorporada só será exibida (em uma janela separada) quando o botão for pressionado (opcional, o padrão é ″false″)

label

Só faz sentido se as_button=″true″: A legenda do botão (recomendado, o padrão é ″Opções″)

<preview>

Caixa de seleção para ativar/desativar a funcionalidade de pré-visualização. Observe que, a partir da versão 0.6.5 do RKWard os elementos **<preview>** (pré-visualização) são tratados de forma especial em diálogos de plugins (não em assistentes): Eles serão colocados na coluna de botões, independentemente de onde estejam definidos na interface do usuário. Ainda é uma boa prática defini-los em um local adequado no layout, para manter a compatibilidade com versões anteriores. Atributos:

label

Legenda da caixa (opcional, o padrão é ″Pré-visualização″)

mode

Tipo de pré-visualização. Os tipos suportados são ″plot″ (consulte o capítulo sobre pré-visualizações de gráficos), ″output″ (consulte o capítulo sobre pré-visualizações de saída (HTML)), ″data″ (consulte o pré-visualizações de dados) e ″custom″ (consulte o pré-visualizações personalizadas). (opcional, o padrão é ″plot″)

placement

Posicionamento da pré-visualização: ″attached″ (à área de trabalho principal), ″detached″ (janela independente), ″docked″ (anexada à caixa de diálogo do plugin) e ″default″ (atualmente, é igual a ″docked″, mas poderá ser configurável pelo usuário em algum momento). Em geral, recomenda-se manter esta configuração como padrão para melhor consistência da interface do usuário (opcional, o padrão é ″default″).

active

Indica se a pré-visualização está ativa por padrão. Em geral, apenas as pré-visualizações ancoradas (docked) devem ser ativadas por padrão e, mesmo para estas, há um motivo para que o padrão seja pré-visualizações inativas (opcional, o padrão é ″false″)

A.3.5 Seção de lógica

<logic>

O elemento que contém a seção de lógica. Todos os elementos abaixo são permitidos somente dentro do elemento <logic>. O elemento <logic> é permitido somente como filho direto do elemento <document> (no máximo uma vez por documento) ou de elementos <optionset> (no máximo uma vez por optionset). A seção de lógica do documento se aplica tanto a GUIs <dialog> quanto a <wizard> da mesma forma.

<external>

Cria uma nova propriedade (string) que deve ser conectada a uma propriedade externa caso o plugin seja incorporado. Consulte a seção sobre plugins ″incompletos″. Atributos:

id
O ID da nova propriedade (obrigatório)

default
O valor de string padrão da nova propriedade, isto é, o valor usado se a propriedade não estiver conectada a uma propriedade externa (opcional, o padrão é uma string vazia)

<i18n>

Cria uma nova propriedade (string) que deve fornecer uma legenda internacionalizado (i18n). Atributos:

id
O ID da nova propriedade (obrigatório)

label
A legenda. Isto será traduzido. (obrigatório)

<set>

Define uma propriedade com um valor fixo (é claro que, se você conectar a propriedade a alguma outra propriedade, o valor não permanecerá fixo). Por exemplo, se você incorporar um plugin, mas quiser ocultar alguns de seus elementos, você pode definir a propriedade de visibilidade desses elementos como falsa. Útil especialmente para plugins incorporados. Nota: Se houver vários elementos <set> para um único `id`, o último a ser definido terá precedência. Isso às vezes será útil ao usar partes <include>. Atributos:

id
O ID da propriedade a ser definida (obrigatório)

to
O valor da string para definir a propriedade (obrigatório). Observação: Para propriedades booleanas, como visibilidade e habilitação, você normalmente definirá o atributo como ″true″ ou ″false″.

<convert>

Cria uma nova propriedade booleana que depende do estado de uma ou mais propriedades diferentes. Atributos:

id
O ID da nova propriedade (obrigatório)

sources
Os IDs das propriedades das quais esta propriedade dependerá. Uma ou mais propriedades podem ser especificadas, separadas por ″;″ (obrigatório)

mode
O modo de conversão/operação. Um dos seguintes: ″equals″ (igual a), ″notequals″ (diferente de), ″range″ (intervalo), ″and″ (E lógico) ou ″or″ (OU lógico). Se o modo for ″igual a″, a propriedade será verdadeira somente se: o valor de todas as suas fontes for igual ao padrão do atributo (veja abaixo). Se o modo for ″diferente de″, a propriedade será verdadeira somente se o valor de todas assuas fontes for diferente do padrão do atributo (veja abaixo). Se o modo for ″intervalo″, as fontes devem ser numéricas (inteiras ou reais). A propriedade será verdadeira somente se todas as fontes estiverem no intervalo especificado pelos atributos mínimo e máximo (veja

abaixo). Se o modo for ″E″, as fontes devem ser propriedades booleanas. A propriedade será verdadeira somente se todas as fontes forem verdadeiras simultaneamente. Se o modo for ″OU″, as fontes devem ser propriedades booleanas. A propriedade será verdadeira somente se pelo menos uma das fontes for verdadeira. (obrigatório)

standard
Só faz sentido nos modos equals ou notequals: o valor da string a ser comparado contra (obrigatório se estiver em um desses modos)

min
Somente significativo no modo range: o valor mínimo para comparação (opcional, o padrão é o menor número de ponto flutuante representável na máquina)

max
Somente significativo no modo range: o valor máximo para comparação (opcional, o padrão é o maior número de ponto flutuante representável na máquina)

require_true
Se definida como ″true″, a propriedade se tornará obrigatória e só será considerada válida se seu estado for verdadeiro/ativado. Portanto, se a propriedade for falsa, ela bloqueará o botão **Enviar** (opcional, o padrão é ″false″).

> CUIDADO
> Se você usar isso, certifique-se de que o usuário possa detectar facilmente o que está errado, por exemplo, mostrando um <text> explicativo.

<switch>
Cria uma nova propriedade que será repassada para diferentes propriedades de destino (ou strings fixas) com base no valor de uma propriedade de condição. Isso permite criar uma lógica semelhante às construções `if()` ou `switch()`. Atributos:

id
O ID da nova propriedade (obrigatório)

condition
O ID da propriedade de condição (obrigatório)

Elementos-filho:

<true>
Se a propriedade de condição for booleana, você pode especificar os dois elementos filhos: <true> e <false> (e somente estes). (Obrigatório, se <false> também for fornecido)

<false>
Se a propriedade de condição for booleana, você pode especificar os dois elementos filhos: <true> e <false> (e somente estes). (Obrigatório, se <true> também for fornecido)

<case>
Se a propriedade de condição não for booleana, você pode fornecer um número arbitrário de elementos <case>, um para cada valor da propriedade de condição que você deseja corresponder (pelo menos um elemento desse tipo é necessário se a propriedade de condição não for booleana).

<default>
Se a propriedade de condição não for booleana, o elemento opcional <default> permite especificar o comportamento caso nenhum elemento <case> corresponda ao valor da propriedade de condição (opcional, permitido apenas uma vez, em combinação com um ou mais elementos <case>).

Os elementos filhos <true>, <false>, <case> e <default> assumem os seguintes atributos:

standard
Apenas para elementos <case>: O valor para comparar a propriedade de condição contra (obrigatório, string).

fixed_value
: Uma string fixa que deve ser fornecida como o valor da propriedade <switch> se a condição atual for atendida (obrigatório, se dynamic_value não for fornecido).

dynamic_value
: O `id` da propriedade de destino que deve ser fornecido como o valor da propriedade <switch> se a condição atual corresponder (obrigatório, se fixed_value não for fornecido).

<connect>
: Conecta duas propriedades. A propriedade do cliente será alterada sempre que a propriedade do governador for alterada (mas não o contrário). Atributos:

client
: O ID da propriedade do cliente, ou seja, a propriedade que será ajustada (obrigatório)

governor
: O ID da propriedade do governador, isto é, a propriedade que ajustará a propriedade do cliente. Isso pode incluir um modificador (obrigatório)

reconcile
: Se ″verdadeiro″, a propriedade do cliente ajustará a propriedade do governador na conexão de forma que a propriedade do governador aceite apenas valores que também sejam aceitáveis pelo cliente (por exemplo, suponha que o governador seja uma propriedade numérica com valor mínimo ″0″ e o cliente seja uma propriedade numérica com valor mínimo ″100″. O mínimo de ambas as propriedades será ajustado para 100, se reconcile=″true″). Geralmente funciona apenas para propriedades do mesmo tipo básico (opcional, padrão ″falso″).

<dependency_check>
: Cria uma propriedade booleana que é verdadeira se as dependências especificadas forem atendidas, e falsa caso contrário. A sintaxe XML do elemento é a mesma do elemento **<dependencies>**, descrito na referência de `.pluginmap`. A partir da versão 0.6.1, apenas as especificações de versão do RKWard e R são consideradas, e não as dependências de pacotes ou pluginmaps.

<script>
: Define o código do script para controlar a lógica da interface do usuário. Consulte a seção sobre lógica GUI com script para obter detalhes. O código do script a ser executado pode ser fornecido usando o atributo `″file″`, ou como um texto (comentado) do elemento. O elemento **<script>** não é permitido na seção **<logic>** de um conjunto de opções. Atributos:

file
: Nome do arquivo de script. (obrigatório)

A.4 Propriedades dos elementos do plugin

Todos os elementos de layout e todos os elementos ativos possuem as seguintes propriedades, acessíveis através de ″id_of_element.name_of_property″:

visible
: Indica se o elemento GUI está visível ou não (booleano)

enabled
: Indica se o elemento GUI está habilitado ou não (booleano)

required
: Indica se o elemento GUI é obrigatório (para conter uma configuração válida) ou não. Observe que qualquer elemento que esteja desativado ou oculto também é implicitamente não obrigatório (booleano).

Além disso, alguns elementos possuem propriedades adicionais às quais você pode se conectar. A maioria dos elementos ativos também possui uma propriedade ″padrão″ cujo valor será retornado nas chamadas para `getBoolean/getString/getList (˜...˜)`, caso nenhuma propriedade específica tenha sido nomeada, conforme descrito abaixo.

<text>

A propriedade padrão é texto.

text
O texto exibido (texto)

<varselector>

Nenhuma propriedade padrão

selected
Os objetos atualmente selecionados. Provavelmente você não deseja usar isso. Usado internamente (RObject)

root
O objeto raiz/pai dos objetos oferecidos para seleção (RObject)

<varslot>

A propriedade padrão é ″available″

available
Todos os objetos armazenados no varslot (RObject)

selected
Dos objetos contidos no varslot, aqueles que estão atualmente selecionados. Você provavelmente não deseja usar isso. Usado internamente (RObject)

source
Uma cópia dos objetos selecionados no seletor de variáveis correspondente. Você provavelmente não deseja usar isso. Usado internamente (RObject)

<valueselector>

A propriedade padrão é ″selected″

selected
As strings atualmente selecionadas. Modificador ″labeled″ para recuperar os rótulos correspondentes. Em um <valueselector>, você provavelmente não deseja usar isso diretamente (somente em um <select>). (leitura/gravação de StringList)

available
Lista de valores de string para seleção. (leitura/gravação de StringList)

labels
Legendas a serem exibidas para os valores de string. (lista de strings de leitura/gravação)

<valueslot>

Semelhante a <varslot>, mas as propriedades são listas de strings, em vez de RObjects.

<radio>

A propriedade padrão é ″string″

string
O valor da opção atualmente selecionada (string)

number
O número da opção atualmente selecionada (as opções são numeradas de cima para baixo, começando em 0) (número inteiro)

<dropdown>

O mesmo que <radio>

<select>

O mesmo que <valueselector>

<option>

Não há propriedade padrão. ″enabled″ é a *única* propriedade e não está disponível atualmente para opções dentro de um <select> ou <valueselector>. <option> não possui as propriedades ″visible″ ou ″required″.

enabled

Indica se esta única opção deve ser ativada ou desativada. Na maioria dos casos, você ativará/desativará todo o <radio< ou <dropdown<. Mas isso pode ser usado para definir dinamicamente a ativação de uma única opção dentro de um <radio< ou <dropdown< (booleano)

<checkbox>

A propriedade padrão é ″state.labeled″, o que significa que os valores especificados pelos atributos `value` e `value_unchecked` são retornados, *não* a legenda exibida da caixa de seleção.

state

Estado da caixa de seleção (ligada ou desligada). Observe que modificadores úteis desta propriedade (como de todas as propriedades booleanas) são ″not″ e ″labeled″ (consulte tipos de propriedades). No entanto, muitas vezes é mais útil conectar-se à propriedade sem modificador, ou seja, ″*checkbox_id*.state″, que retornará o estado da caixa de seleção em um formato adequado para uso em uma instrução if (0 ou 1). (booleano)

<frame>

A propriedade padrão é ″checked″ (marcada), se - e somente se - o frame for marcável. Para frames não marcáveis, não há propriedade padrão.

checked

Disponível apenas para quadros selecionáveis: estado da caixa de seleção (ativada ou desativada). Observe que modificadores úteis para esta propriedade (como para todas as propriedades booleanas) são ″not″ e ″numeric″ (consulte tipos de propriedades). (booleano)

<input>

A propriedade padrão é ″texto″

text

Texto atual no campo de entrada (string)

<matrix>

A propriedade padrão é ″cbind″.

rows

Número de linhas na matriz (inteiro). Se a matriz permitir que o usuário adicione/remova linhas, esta propriedade deve ser tratada como somente leitura. Caso contrário, alterá-la modificará o tamanho da matriz.

columns

Número de colunas na matriz (inteiro). Se a matriz permitir que o usuário adicione/remova colunas, esta propriedade deve ser tratada como somente leitura. Caso contrário, alterá-la modificará o tamanho da matriz.

tsv
Os dados na matriz estão em formato TSV (string; leitura e gravação). Observe que, em comparação com o layout TSV usual, as *colunas*, e não as linhas, são separadas por caracteres de nova linha, e as células dentro de uma coluna são separadas por caracteres de tabulação.

0,1,2...
Os dados de uma única coluna (0 para a coluna mais à esquerda). `getValue()`/`getString()` retorna isso como uma única string, separada por ″\n″. No entanto, a maneira recomendada de obter isso é usando `getList()`, que retorna esta coluna como uma matriz de strings.

row.0,row.1,row.2...
Os dados de uma única linha (0 para a linha superior). `getValue()`/`getString()` retorna isso como uma única string, separada por ″\n″. No entanto, a maneira recomendada de obter isso é usando `getList()`, que retorna esta linha como uma matriz de strings.

cbind
Dados em um formato adequado para colar em R, envolvidos em uma instrução cbind (string; somente leitura).

<optionset>

Sem propriedade padrão.

row_count
Número de itens no conjunto de opções (número inteiro). Somente leitura.

current_row
Item atualmente ativo no conjunto de opções (número inteiro). -1 para nenhum item ativo. Leitura e escrita.

optioncolumn_ids
Para cada <optioncolumn> que você definir, uma propriedade de lista de strings será criada com o id especificado.

<browser>

A propriedade padrão é ″selection″

selection
Texto atual (nome do arquivo selecionado) no navegador (string)

`overwrite`
Indica se a opção ″overwrite″ (sobrescrever) está marcada (booleano, somente leitura, ou seja, você pode ler o estado da caixa de seleção, mas não alterá-lo programaticamente)

<saveobject>

A propriedade padrão é ″selection″

selection
Nome completo do objeto selecionado (string; somente leitura - para definir isso programaticamente, use ″parent″ e ″objectname″)

parent
O objeto parent (pai) do objeto selecionado. Este é sempre um objeto R existente de um tipo que pode conter outros objetos (por exemplo, uma lista ou um data.frame). Quando definido como uma string vazia ou um objeto inválido, assume-se ″.GlobalEnv″. (RObject)

objectname
O nome base do objeto selecionado, isto é, a string inserida pelo usuário (alterada para um nome R válido, se necessário) (string)

active
Somente para objetos de salvamento verificáveis: Indica se o controle está marcado/habilitado. Sempre verdadeiro para objetos de salvamento não verificáveis (booleano)

<spinbox>

A propriedade padrão é ″int″ ou ″real.formatted″, dependendo do modo do spinbox

int
Valor inteiro armazenado na caixa de seleção, ou o inteiro mais próximo, se estiver no modo real (inteiro)

real
Valor real contido na caixa de seleção (e inteiro, se for um inteiro) (real)

<formula>

A propriedade padrão é ″model″

model
A string do modelo atual (string)

table
O data.frame que contém as variáveis necessárias. Se forem usadas variáveis de apenas um data.frame, o nome desse data.frame será retornado. Caso contrário, um novo data.frame será construído conforme necessário (string).

labels
Se variáveis de múltiplos data.frames estiverem envolvidas, seus nomes podem ficar alterados (por exemplo, se ambos os data.frames contiverem uma variável chamada ″x″). Isso retorna uma lista com os nomes alterados como índices e a legenda descritiva como valor (string).

fixed_factors
Os fatores fixos. Provavelmente você não vai querer usar isso. Usado internamente (RObject)

dependent
A(s) variável(eis) dependente(s). Provavelmente você não vai querer usar isso. Usado internamente (RObject)

<embed>

Nenhuma propriedade padrão

code
O código gerado pelo plugin incorporado (código)

<preview>

A propriedade padrão é ″state″

state
Indica se a caixa de pré-visualização está marcada (não necessariamente se a pré-visualização já foi exibida) (booleano)

<convert>

Este elemento (usado na seção <logic>) é especial, pois tecnicamente *é* uma propriedade, em vez de apenas conter uma ou mais propriedades. É do tipo booleano. Observe que modificadores úteis para esta propriedade (como para todas as propriedades booleanas) são ″not″ e ″numeric″ (consulte tipos de propriedades)

<switch>

Este elemento (usado na seção <logic>) é especial, pois é tecnicamente uma propriedade (string), em vez de apenas armazenar uma ou mais propriedades. Ele permite alternar entre várias propriedades de destino dependendo do valor de uma propriedade de condição ou remapear os valores da propriedade de condição. Quaisquer modificadores fornecidos são passados para as propriedades de destino; portanto, por exemplo, se todas as propriedades de destino forem propriedades RObject, você também poderá usar o modificador "shortname" na chave. No entanto, se as propriedades de destino forem de tipos diferentes, o uso de modificadores pode levar a erros. Para `fixed_value`s, qualquer modificador é descartado silenciosamente. Observe que as propriedades de destino, quando acessadas por meio de uma chave, são sempre somente leitura.

A.5 Plugins incorporáveis incluídos na versão oficial RKWard

Vários plugins incorporáveis são fornecidos com o RKWard e podem ser usados em seus próprios plugins. A documentação detalhada está disponível atualmente apenas nos arquivos de código-fonte ou de ajuda desses plugins. No entanto, aqui está uma lista para lhe dar uma visão geral rápida do que está disponível:

ID	Pluginmap	Descrição	Exemplo de uso
rkward::plot_options	embedded.pluginmap	Oferece uma ampla gama de opções para gráficos. A maioria dos plugins de plotagem utiliza isto.	Gráficos->Gráfico de barras, a maioria dos outros plugins de plotagem
rkward::color_chooser	embedded.pluginmap	Plugin muito simples para especificar uma cor. A implementação atual fornece uma lista de nomes de cores. Implementações futuras poderão fornecer opções de seleção de cores mais elaboradas.	Gráficos->Histograma
rkward::plot_stepfun_options	embedded.pluginmap	Opções do gráfico da função Passo	Gráficos->Gráfico ECDF
rkward::histogram_options	embedded.pluginmap	Opções (gráficas) do histograma	Gráficos->Histograma
rkward::barplot_embed	embedded.pluginmap	Opções da barra de gráficos	Gráficos->Barra de gráficos
rkward::one_var_tabulation	embedded.pluginmap	Fornece tabelas com base em uma única variável.	Gráficos->Barra de gráficos

rkward::limit_vecto-r_length	embedded.pluginm-ap	Limitar o comprimento de um vetor (aos n maiores ou menores elementos).	Gráficos->Barra de gráficos
rkward::level_select	embedded.pluginm-ap	Fornece um <valueselector> preenchido com os níveis (ou valores únicos) de um vetor.	Dados->Recodificar dados categóricos
rkward::multi_input	embedded.pluginm-ap	Combina controles de spinbox, entrada e rádio para fornecer entrada de dados de caracteres, numéricos e lógicos.	Dados->Recodificar dados categóricos

Tabela A.1: Plugins padrão incorporáveis

A.6 Elementos para uso em arquivos `.pluginmap`

<document>

Precisa estar presente em cada arquivo `.pluginmap` como nó raiz (exatamente uma vez). Atributos:

base_prefix

Os nomes de arquivo especificados no arquivo `.pluginmap` são considerados relativos ao diretório do arquivo `.pluginmap` + o prefixo especificado aqui. Útil, especialmente se todos os seus componentes estiverem localizados em um único subdiretório.

namespace

Um espaço de nomes para os IDs dos componentes. Ao procurar componentes para incorporação, os componentes poderão ser recuperados por meio da string ″namespace::component_id″. Definido como ″rkward″ por enquanto.

id

Uma string identificadora opcional para este `.pluginmap`. Especificar isso permite que autores terceirizados se refiram ao seu `.pluginmap` e o carreguem a partir do deles (consulte o capítulo sobre como lidar com dependências).

priority

Uma das opções *`"hidden"`*, *`"low"`*, *`"medium"`* ou *`"high"`*. Os `.pluginmap` com prioridade ″medium″ ou ″high″ são ativados automaticamente quando o RKWard os encontra pela primeira vez. Use *`priority="hidden"`* para `.pluginmap`s que não devem ser ativados, diretórios (apenas para inclusão). Na implementação atual, isso não oculta o `.pluginmap`. (Opcional, o padrão é ″medium″).

<dependencies>

Este elemento, que especifica dependências, é permitido como filho direto do elemento <document> (uma vez) e como filho de elementos <component> (uma vez para cada elemento <component>). Especifica as dependências que devem ser atendidas para usar o(s) plugin(s). Consulte o capítulo sobre dependências para uma visão geral. Atributos:

rkward_min_version, rkward_max_version
Versão mínima e máxima permitida do RKWard. As especificações de versão podem incluir sufixos não numéricos, como ″0.5.7z-devel1″. Se uma dependência especificada não for atendida, o(s) plugin(s) ao(s) qual(is) ela se aplica *serão ignorados*. Mais informações. Opcional; se não for especificado, nenhuma versão mínima/máxima do RKWard será exigida.

R_min_version, R_max_version
Versão mínima e máxima permitida do R. As especificações de versão *não* podem incluir sufixos não numéricos, como ″0.5.7z-devel1″. A dependência da versão do R será exibida nas páginas de ajuda dos plugins, mas não tem efeito direto, a partir da versão 0.6.1 do RKWard. Mais informações. Opcional; se não for especificado, nenhuma versão mínima/máxima do R será exigida.

platforms
Plataformas onde este plugin está disponível. Os valores suportados são `″unix″`, `″windows″`, `″macos″`, `″any″` e combinações separadas por dois pontos (por exemplo, `″unix:macos″`). `″unix″` inclui quaisquer variantes de Linux e BSD, mas *não* MacOS. Se o seu plugin não depender de plataforma, basta omitir este atributo.

Elementos-filho:

<package>
Adiciona uma dependência a um pacote R específico. Atributos:
name
Nome do pacote (obrigatório).
min_version, max_version
Versão mínima/máxima permitida (opcional).
repository
Repositório onde o pacote pode ser encontrado. Opcional, mas altamente recomendado, caso o pacote não esteja disponível no CRAN.

<pluginmap>
Adiciona uma dependência a um específico `.pluginmap` do RKWard. Atributos:
name
String de identificação do `.pluginmap` obrigatório (obrigatório).
min_version, max_version
Versão mínima/máxima permitida (opcional).
url
URL onde o `.pluginmap` pode ser encontrado. Obrigatório.

<about>
Pode estar presente exatamente uma vez como filho direto do elemento <document>. Contém metainformações sobre o `.pluginmap` (ou plugin). Consulte o capítulo sobre informações do 'about' para obter uma visão geral. Atributos:

name
Nome visível do usuário. Opcional. Não precisa ser o mesmo que o ″id″.
version
Número da versão. Opcional. O formato não é restrito, mas para maior segurança, siga esquemas de versionamento comuns, como ″x.y.z″.
releasedate
Especificação da data de lançamento. Opcional, no formato ″YYYY-MM-DD″.
shortinfo
Uma *breve* descrição do plugin / `.pluginmap`. Opcional.
url
URL onde você pode encontrar mais informações. Opcional, mas recomendado.

copyright
: Especificação de direitos autorais, por exemplo: ″2012-2013 por John Doe″. Opcional, mas recomendado.

licence
: Especificação da licença, por exemplo, ″GPL″ ou ″BSD″. Certifique-se de acompanhar seus arquivos com uma cópia completa da licença relevante. Opcional, mas recomendado.

category
: Categoria de plugin(s), por exemplo, ″Teoria de resposta ao item″. A partir da versão 0.6.1, nenhuma categoria está predefinida. Opcional.

Elementos-filho:

<author>
: Adiciona informações sobre um autor. Atributos:

  name, given, family
  : Especifique o nome completo para `name`, ou especifique `given` e `family` separadamente.

  role
  : Descrição da função do autor (opcional).

  email
  : Endereço de e-mail para contato com o autor. Obrigatório. Pode ser definido como a lista de discussão rkward-devel, caso você esteja inscrito e seu plugin seja destinado a ser incluído na versão oficial do RKWard.

  url
  : URL com mais informações sobre o autor, por exemplo página inicial (opcional).

<components>
: Precisa estar presente exatamente uma vez como filho direto do elemento <document>. Contém os elementos <component> individuais descritos abaixo. Sem atributos.

<component>
: Um ou mais elementos <component> devem ser fornecidos como filhos diretos do elemento <components> (e somente lá). Registra um componente/plugin com o rkward. Atributos:

  type
  : Para futuras extensões: O tipo de componente/plugin. Por enquanto, sempre definido como ″padrão″ (o único tipo atualmente suportado).

  id
  : O ID pelo qual este componente pode ser recuperado (para colocá-lo no menu (veja abaixo), ou para incorporá-lo). Veja <document>-namespace acima.

  file
  : Requerido pelo menos para componentes do tipo ″padrão″: O nome do arquivo XML que descreve a GUI.

  label
  : A legenda para este componente, quando colocado na hierarquia do menu.

  optional
  : Só faz sentido para componentes com dependencies definidas: Normalmente, é considerado um erro reportável se um componente não for compatível com esta versão do RKWard. No entanto, se o componente realmente não for necessário no ambiente atual, definir este atributo como `″true″` suprime qualquer aviso (`″false″` por padrão).

<attribute>
: Define um atributo de um componente. Até o momento, só faz sentido para plugins de importação. Permitido apenas como um filho direto de <component>. Atributos:

id
Id do atributo
value
Valor do atributo
labels
Legenda associada ao atributo

<hierarchy>

Precisa estar presente exatamente uma vez como filho direto do elemento <document>. Descreve onde os componentes declarados acima devem ser colocados na hierarquia do menu. Aceita apenas elementos <menu> como filhos diretos. Sem atributos.

<menu>

Um ou mais elementos <menu> devem ser fornecidos como filhos diretos do elemento <hierarchy>. Declara um novo (sub)menu. Se um menu com o ID fornecido (veja abaixo) já existir, os dois menus serão mesclados. O elemento <menu> pode ser um filho direto do elemento <hierarchy> (menu de nível superior) ou um filho direto de qualquer outro elemento <menu> (submenu). Por outro lado, o elemento <menu> aceita outros elementos <menu> ou <entry> como filhos. Atributos:

id
Uma string identificadora do menu. Útil quando as definições de menu são lidas de vários arquivos `.pluginmap` para garantir que os plugins possam ser colocados no(s) mesmo(s) menu(s). Alguns IDs de menu, como ″file″, referem-se a menus predefinidos (neste caso, o menu ″File″). Certifique-se de verificar os arquivos `.pluginmap` existentes para usar IDs consistentes.

label
Uma legenda para o menu.

group
Permite controlar a ordem dos itens do menu. Consulte ordem dos itens do menu. Opcional.

<entry>

Uma entrada de menu, isto é, uma opção de menu para invocar um plugin. Pode ser usado apenas como filho direto de um elemento <menu>, não aceitando elementos filhos. Atributos:

component
O ID do componente que deve ser invocado quando esta entrada de menu for ativada.

group
Permite controlar a ordem dos itens do menu. Consulte ordem dos itens do menu. Opcional.

<group>

Declara um grupo de itens no menu. Consulte ordenação de itens do menu. Atributos:

id
O nome deste grupo.

separated
Opcional. Se definido como ″verdadeiro″, o item neste grupo será visualmente separado dos itens ao redor.

group
Nome do grupo ao qual este grupo será adicionado (opcional).

<context>

Declara as entradas em um contexto. Permitido apenas como filho direto da tag <document>. Aceita apenas tags <menu> como filhos diretos. Atributos:

id

O ID do contexto. Até o momento, apenas dois contextos foram implementados: ″x11″ e ″import″.

<require>

Inclui outro arquivo `.pluginmap`. Este arquivo `.pluginmap` é carregado apenas uma vez, mesmo que seja <require> de vários outros arquivos. O caso de uso mais importante é incluir um arquivo pluginmap, que declara alguns componentes, que são incorporados por componentes declarados neste `.pluginmap`. Elementos <require> são permitidos apenas como filhos diretos do nó <document>. Atributos:

file

O nome do arquivo do `.pluginmap` a ser incluído. Isso é visto em relação ao diretório do arquivo `.pluginmap` atual + o prefixo base (veja acima, elemento <document>). Se você não souber o caminho relativo para o `.pluginmap` a ser incluído, use o atributo *`map`* para referenciá-lo pelo id.

map

Para incluir um arquivo `.pluginmap` de um pacote diferente (ou um `.pluginmap` do RKWard de seu `.pluginmap` externo), você pode se referir a ele pelo seu*`namespacename::id`*, conforme especificado no elemento<document> do `.pluginmap` obrigatório. A inclusão falhará se nenhum `.pluginmap` com esse id for conhecido (por exemplo, não estiver instalado no sistema do usuário). Você deve usar este método apenas para incluir `.pluginmap`s fora do seu pacote. Para mapas dentro do seu pacote, especificar um caminho relativo (atributo *`file`*) é mais rápido e confiável.

A.7 Elementos para uso em arquivos .rkh (ajuda)

<document>

Precisa estar presente em cada arquivo `.xml` como nó raiz (exatamente uma vez). Sem atributos.

<title>

Título da página de ajuda. Isto *não* é interpretado para páginas de ajuda de um plugin (este título é obtido do próprio plugin), apenas para páginas independentes. Sem atributos. O texto contido na tag <title> se tornará a legenda da página de ajuda. Pode ser definido apenas uma vez, como filho direto do nó <document>.

<summary>

Um breve resumo da página de ajuda (ou para que serve este plugin). Este será sempre exibido no topo da página de ajuda. Sem atributos. O texto contido na tag <summary> será exibido. Recomendado, mas não obrigatório. Pode ser definido apenas uma vez, como filho direto do nó <document>.

<usage>

Um resumo um pouco mais detalhado do uso. Este será sempre exibido diretamente após o <summary>. Sem atributos. O texto contido na tag <usage> será exibido. Recomendado para páginas de ajuda de plugins, mas não obrigatório. Pode ser definido apenas uma vez, como filho direto do nó <document>.

<section>

Uma seção de uso geral. Pode ser usada qualquer número de vezes como um filho direto do nó <document>. Essas seções são exibidas na ordem de sua definição, mas todas *após* a seção <usage> e *antes* a seção <settings>. O texto contido na tag <section> será exibido.

id

Um identificador necessário para acessar esta seção a partir da barra de navegação (ou um link). Precisa ser único dentro do arquivo. Obrigatório, sem valor padrão.

title
Título (legenda) desta seção. Obrigatório, sem valor padrão.

short_title
Um título curto adequado para ser exibido na barra de navegação. Opcional, o padrão é o título completo.

<settings>

Define a seção que contém referências às várias configurações da GUI. Somente significativo e usado apenas para páginas de ajuda relacionadas a plugins. Use como um filho direto do <document>. Pode conter apenas elementos <settings> e <caption> como filhos diretos. Sem atributos.

<setting>

Explica uma única configuração na GUI. Permitido apenas como filho direto do elemento <settings>. O texto contido no elemento é exibido.

id
O ID da configuração no plugin `.xml`. Obrigatório, sem valor padrão.

title
Um título opcional para a configuração. Se omitido (a omissão é recomendada na maioria dos casos), o título será obtido do plugin `.xml`.

<caption>

Uma legenda para agrupar visualmente várias configurações. Só pode ser usada como um elemento filho direto do elemento <settings>.

id
O ID do elemento correspondente (normalmente um <frame>, <page> ou <tab>) no plugin `.xml`.

title
Um título opcional para a legenda. Se omitido (a omissão é recomendada na maioria dos casos), o título será retirado do plugin `.xml`.

<related>

Define uma seção contendo links para informações adicionais relacionadas. Será sempre exibida após a seção <settings>. Sem atributos. O texto contido na tag <related> será exibido. Normalmente, isso conterá uma lista no estilo HTML. Recomendado para páginas de ajuda de plugins, mas não obrigatório. Pode ser definido apenas uma vez, como filho direto do nó <document>.

<technical>

Define uma seção contendo informações técnicas irrelevantes para os usuários finais (como a estrutura interna do plugin). Será sempre exibida por último em uma página de ajuda. Sem atributos. O texto contido na tag <related> será exibido. Não é obrigatório e não é recomendado para a maioria das páginas de ajuda de plugins. Pode ser definido apenas uma vez, como filho direto do nó <document>.

<link>

Um link. Pode ser usado em qualquer uma das seções descritas acima.

href
O destino URL. Observe que vários URLs específicos para RKWard estão disponíveis. Consulte a seção sobre como escrever páginas de ajuda para obter detalhes.

<label>

Insere o valor de um rótulo da interface do usuário. Pode ser usado em qualquer uma das seções descritas acima.

id
O ID do elemento no plugin, do qual copiar o atributo `label`.

<várias tags HTML>

A maioria das tags HTML básicas são permitidas dentro das seções. No entanto, por favor, mantenha a formatação manual ao mínimo.

A.8 Funções disponíveis para scripts de lógica de GUI

Classe "Component"

Classe que representa um único componente ou propriedade de componente. A instância mais importante desta classe é a variável "gui", que é predefinida como a propriedade raiz do componente atual. Os seguintes métodos estão disponíveis para instâncias da classe "Component":

absoluteId(base_id)
Retorna o ID absoluto de *base_id*, ou - se base_id for omitido - o identificador do componente.

getValue(id)
Desaconselhado. Use `getString(), getBoolean() ou getList()` em vez disso. Retorna o valor da propriedade filha fornecida. Retorna o valor desta propriedade, se o ID for omitido.

getString(id)
Retorna o valor da propriedade filha fornecida como uma string. Retorna o valor desta propriedade, se o ID for omitido.

getBoolean(id)
Retorna o valor da propriedade filha fornecida como um booleano (se possível). Retorna o valor desta propriedade, caso o ID seja omitido.

getList(id)
Retorna o valor da propriedade filha fornecida como uma matriz de strings (se possível). Retorna o valor desta propriedade, se o ID for omitido.

setValue(id, valor)
Defina o valor da propriedade filha fornecida como *valor*.

getChild(id)
Retorna uma instância da propriedade filha com o *id* fornecido.

addChangeCommand(id, comando)
Execute o comando `comando` sempre que a propriedade filha fornecida por *id* for alterada. `id` pode ser fornecido como uma única string ou como um array de IDs (se a função for chamada para alterações em várias propriedades). `comando` é um valor chamável (geralmente uma função), porém, para compatibilidade com plugins escritos para versões anteriores do RKWard, também pode ser fornecido como uma string para ser avaliada.

A função retorna o parâmetro `comando`, para conveniência (para que você possa, por exemplo, atribuí-lo a uma variável e/ou chamá-lo durante a inicialização).

Classe "RObject"

Classe que representa um único objeto R. Uma instância desta classe pode ser obtida usando o comando **makeRObject(nome_do_objeto)**. Os seguintes métodos estão disponíveis para instâncias da classe "RObject":

> ATENÇÃO
> Se ainda houver comandos pendentes no servidor, as informações fornecidas por esses métodos podem estar desatualizadas no momento da execução do código do plugin. *Não* confie nelas para operações críticas (correndo o risco de perda de dados).

getName()
: Retorna o nome absoluto do objeto.

exists()
: Retorna se o objeto existe. Geralmente, você deve verificar isso antes de usar qualquer um dos métodos listados abaixo.

dimensions()
: Retorna uma matriz de dimensões (semelhante a **dim()** em R).

classes()
: Retorna uma matriz de classes (semelhante a **class()** em R).

isClass(classe)
: Retorna verdadeiro se o objeto for da classe *classe*.

isDataFrame()
: Retorna verdadeiro se o objeto for um data.frame.

isMatrix()
: Retorna verdadeiro se o objeto for uma matriz.

isList()
: Retorna verdadeiro se o objeto for uma lista.

isFunction()
: Retorna verdadeiro se o objeto for uma função.

isEnvironment()
: Retorna verdadeiro se o objeto for um ambiente.

isDataNumeric()
: Retorna verdadeiro se o objeto for um vetor de dados numéricos.

isDataFactor()
: Retorna verdadeiro se o objeto for um vetor de dados fatoriais.

isDataCharacter()
: Retorna verdadeiro se o objeto for um vetor de dados de caracteres.

isDataLogical()
: Retorna verdadeiro se o objeto for um vetor de dados lógicos.

parent()
: Retorna uma instância de ″RObject″ representando o objeto pai deste objeto.

child(nomefilho)
: Retorna uma instância de ″RObject″ representando o filho *nomefilho* deste objeto.

Classe ″RObjectArray″

Uma matriz de instâncias de RObject. Uma instância desta classe pode ser obtida usando o comando **makeRObjectArray(nomesobjetos)**. É particularmente útil ao lidar com 'vars-lots', que permitem selecionar vários objetos.

Função include()

O comando **include(filename)** pode ser usado para incluir um arquivo JS separado.

Função doRCommand()

Obsoleto. Não use em novos plugins: **doRCommand(comando, callback)**. Use **new RCommand()** em vez disso.

Função new RCommand()

new RCommand(comando, id_opcional) pode ser usado para consultar o R para obter informações. Leia a seção sobre consultando o R de dentro de um plugin para obter detalhes e ressalvas.

Apêndice B

Solução de problemas durante o desenvolvimento de plugins

Então você leu toda a documentação, fez tudo certo e ainda assim não consegue fazer o plugin funcionar? Não se preocupe, vamos resolver isso. A primeira coisa a fazer é: ativar a janela **Mensagens de depuração do RKWard** (disponível no menu **Janelas** ou clicando com o botão direito em uma das barras de ferramentas) e, em seguida, iniciar seu plugin novamente. Como regra geral, você não deve ver nenhuma saída na janela de mensagens quando seu plugin for invocado ou em qualquer outro momento. Se houver alguma, provavelmente está relacionada ao seu plugin. Veja se isso ajuda.

Se tudo parecer normal no console, tente aumentar o nível de depuração (na linha de comando, usando **rkward --debug-level 3** ou definindo o nível de depuração para 3 em **Configurações** → **Configurar RKWard** → **Depurar**). Nem todas as mensagens exibidas em níveis de depuração mais altos indicam necessariamente um problema, mas é provável que o seu problema apareça em algum lugar entre as mensagens.

Se você ainda não conseguiu descobrir o que está errado, não se desespere. Sabemos que isso é complicado e, afinal, talvez você também tenha se deparado com um bug no RKWard que precisa ser corrigido. Basta escrever para a lista de discussão de desenvolvimento e nos contar sobre o problema. Ficaremos felizes em ajudar.

Por fim, mesmo que você tenha descoberto como fazer por conta própria, mas achou a documentação pouco útil ou até mesmo incorreta em alguns aspectos, por favor, nos informe também na lista de discussão, para que possamos corrigir/melhorar a documentação.

Apêndice C

Licença

Tradução de Marcus Gama marcus.gama@kde.org

Esta documentação é licenciada sob os termos da Licença de Documentação Livre GNU.