t c l / t k Pacote em R

TUTORIAL (Versão 1.0)

Ben Dêivide de Oliveira Batista

15 de Maio de 2020

Conteúdo

1 Considerações iniciais

Estudar progamação de Interface Gráfica ao Usuário (IGU) em *R*, parece um tanto complicado quando buscamos referências ou materiais em português. A criação de pacotes são de extrema importância para desenvolvedores em *R* que pensam em usuários mais amplos, isto é, pesquisadores, estudantes, dentre outros, que não são familiarizados com a linguagem *R*, mas que necessitam de suas rotinas.

Em vista disso, procuramos encontrar um pacote que fosse bem versátil, e encontrei o pacote tcltk, desenvolvido por Peter Dalgaard, e hoje incluído na base do *R* desde a versão 1.1.0. Este pacote permite a criação de interfaces gráficas no *R* usando a linguagem *Tcl* e sua extensão *Tk*, desenvolvida por John Ousterhout em 1990, linguagem essa que interage muito bem também com o *Phyton*, *Perl*, *Ruby*, entre outros. O pacote mais conhecido no *R* com interface gráfica feita pelo pacote tcltk é o pacote Rcmdr, desenvolvido por John Fox. Existem alguns outros pacotes que desenvolvem IGU no *R*, como o gWidgets, GTk2, entre outros. O problema é que a linguagem por traz desses pacotes são externas ao *R* e precisam ser instaladas, senão houver no seu sistema operacional. Daí, isso se torna um tremendo problema quando os pacotes que nós desenvolvermos, forem submetidos ao CRAN. Uma vantagem de desenvolver uma IGU no *R* é que o pacote tcltk é da base da linguagem e assim, seu pacote com IGU desenvolvido pelo tcltk, não precisará de dependência externa como por exemplo o gWidgets. Isso facilita e tanto a submissão de nossos pacotes.

Vendo a importância desse pacote, observei que o próprio *help* do pacote é muito pobre em detalhar as suas funções. Por isso da finalidade desse tutorial. Tentar apresentar, de forma simples como desenvolver interfaces gráficas ao usuários de pacotes em R usando o tcltk.

Ao longo do texto, queremos deixar como sinônimos os termos componentes, para *widget* (elementos gráficos) ou janelas, bem como argumento para opção. Alguns componentes podem ser armazenados em objetos no *R*. Logo, em alguns momentos poderemos chamar componentes de objetos. Usaremos um parêntese no final de cada função com uma fonte diferenciada (`\texttt{}`), `tklabel()`, para diferenciar de um argumento da função, por exemplo `text`, com fonte diferenciada (`\texttt{}`). Um pacote será denotado por uma fonte diferente (`\textsf{}`). O nome das linguagens sempre em itálico, e como é comum, alguns termos quando tiver explicando alguma ideia sobre uma função ou sobre alguma linguagem, é comum usarmos termos em inglês. Estes também estarão em itálico.

Por fim, é importante que o leitor tenha familiaridade com a linguagem *R*, pois poderemos ao longo do texto apresentar alguns códigos com funções que nos auxiliam a diminuir o número de linhas destes, sintetizando as rotinas.

Algumas referências para estudos mais aprofundados, Franca (2005), Lawrence e Verzani (2004).

2 Entendimento para desenvolver uma IGU

Basicamente, para desenvolvermos uma IGU precisamos entender a hierarquia de seus componentes (janelas, textos, quadros, botões, etc.). Pela Figura 1, percebemos que a janela principal chamada *Imprimir* tem maior hierarquia dentre os componentes da IGU e chamamos de componente *pai*. Os demais componentes são chamados de *filho*. E nada impede que outros componentes dependam desses filhos.

Na Figura 2, percebemos que o componente *pai* tem maior hierarquia e todos dependem dele. Ao passo que o *filho 2* é dependente do *filho 1*, e os *filhos 3* dependem do *filho 2*. Nessa ideia, o componente filho 1 é o componente *pai* do *filho 2*, e assim por diante. Veremos mais a frente que esses componentes são criados por funções no pacote `tcltk`, e que por exemplo, na Figura 2 o componente *pai* será criado pela função `ttktoplevel()` , o *filho 1* pela função `ttkpanedwindow()` , o *filho 2* pela função `ttklabelframe()` e o *filho 3* veremos mais a frente.

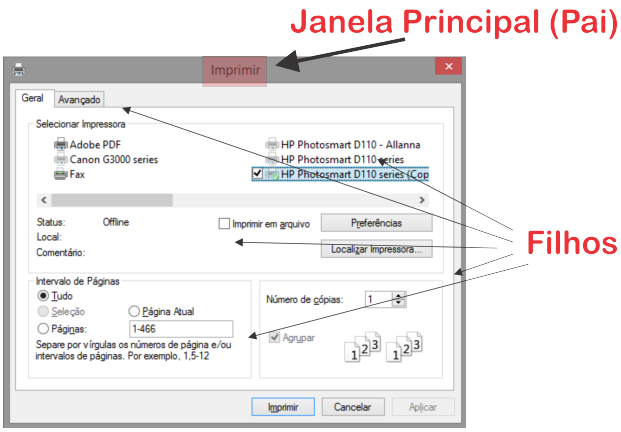

Figura 1: Hierarquia dos componentes da IGU.

Basicamente, o primeiro argumento das funções representam o objeto para o qual se deseja configurar ou atribuir algum comando, isto é, o componente *pai*. Os argumentos seguintes são usados para especificar essas configurações ou atribuições, isto é,

`função(parent, opção...)`.

Esse `parent` pode ser qualquer objeto, uma janela, um *label*, etc.. Usando a Figura 2 como exemplo, apresento o Código

Código R 2.1

Script:

```
1 # Nao execute
2 #Imprimir <- ttktoplevel()
3 #Filho1 <- ttkpanedgroup(Imprimir, ...)
4 #Filho2 <- ttklabelframe(Filho1, ...)
```

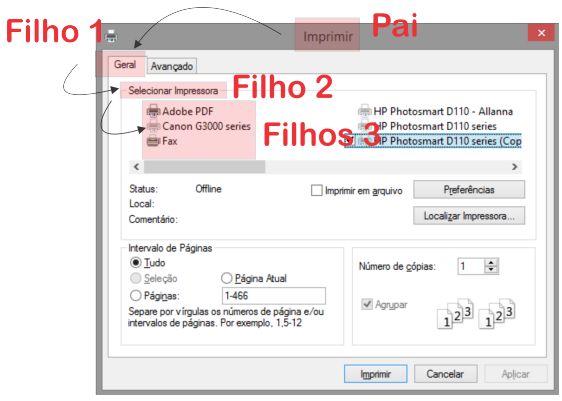

Figura 2: Componente *Pai* e seus *filhos*.

3 Instalação

Para instalar o pacote `tcltk`, veja o Código 3.1, uma vez que esse pacote é da base do R, não precisaremos usar a função `install.packages()`.

Código R 3.1

Script:

```
1 # Carregando o pacote tcltk
2 library(tcltk)
```

4 Gerenciador de janelas

O gerenciador de janelas é a parte das funções do pacote tcltk que é responsável pra criação e configuração das janelas. Chamaremos de janela principal, o componente pai em que toda a IGU e desenhada sobre ela. E janelas filhas, as que tem hierarquia inferior a janela principal, isto é, são desenhadas como dependentes à janela principal. Uma janela será sempre criada pela função `tktoplevel()` e suas configurações, por funções cujo prefixo é iniciado por `tkwm.`, que pode ser visto na Tabela 1.

4.1 Criação da janela principal e seu título

No pacote `tcltk` os comandos sempre iniciarão pelo prefixo `tk`. O outro prefixo variante `ttk` corresponde ao novo tema variante da ferramenta. Por exemplo, o comando `tklabel` e `ttklabel` são equivalentes. Para iniciarmos, vamos criar um pacote chamado `MCP`. Uma sequência de comandos será apresentado a seguir para a criação da janela principal do pacote, ver o **Código R** 4.1.

Código R 4.1

Script:

```
1 # Janela principal
2 mcpprincipal <- tktoplevel()
3 # Nome do Pacote
4 tkwm.title(mcpprincipal, "MCP") # Primeira opcao
5 # tktitle(mcpprincipal) <- "MCP"  # Segunda opcao
```

4.2 Ocultar/Exibir janelas e/ou componentes

Sempre quando iniciamos uma IGU (Interface Gráfica ao Usuário), devemos entender a visualização dos recursos disponíveis, como menu, botões, etc, são construídos de forma gradativa, linha a linha. Assim, para que o usuário veja apenas o resultado final, após carregar o pacote, é interessante utilizar a função `tclServiceMode()` . Vejamos os comandos do **Código R** 4.2.

Tabela 1: Funções que gerenciam a criação e configuração de janelas.

Função	Finalidade
`tktoplevel()`	Função para criar uma janela.
`tkwm.aspect()`	Função que informa o aspecto desejado para a janela dado por uma proporção entre a largura/altura.
`tkwm.client()`	
`tkwm.colormapwindows()`	
`tkwm.command()`	
`tkwm.deiconify()`	Função que retorna uma janela minimizada ou oculta ao seu estado original.
`tkwm.focusmodel()`	
`tkwm.frame()`	
`tkwm.geometry()`	Função que indica no console a geometria da janela. A saída é um resultado `largura x altura + x + y`.
`tkwm.grid()`	
`tkwm.group()`	
`tkwm.iconbitmap()`	
`tkwm.iconify()`	Função que minimiza uma janela.
`tkwm.iconmask()`	
`tkwm.iconname()`	
`tkwm.iconposition()`	
`tkwm.iconwindow()`	
`tkwm.maxsize()`	Função que especifica o valor máximo para as dimensões largura e altura de uma janela.
`tkwm.minsize()`	Função que especifica o valor mínimo para as dimensões largura e altura de uma janela.
`tkwm.overrideredirect()`	
`tkwm.positionfrom()`	
`tkwm.protocol()`	
`tkwm.resizable()`	Função que permite a janela ser redimensionada ou não, ou ainda escolher qual a dimensão pode ser redimensionada.
`tkwm.state()`	Função que permite alterar o estado da janela: oculta, visível, minimizada ou maximizada.
`tkwm.title()`	Função que determina o título da janela.
`tkwm.transient()`	Função que permite tornar uma janela filha de outra janela. Ainda mais, faz com que somente um ícone seja apresentado para toda IGU, ao passo que minimizando a janela principal, a janela filha automaticamente minimiza.
`tkwm.withdraw()`	Função que oculta a janela.

Código R 4.2

Script:

```
1  # Ocutar a IGU
2  tclServiceMode(FALSE)
3  # Habilitar a IGU
4  tclServiceMode(TRUE)
5  # Default
6  tclServiceMode(NULL)
```

O comando `tclServiceMode(FALSE)` é usado para ocultar algum widget que não deseja visualizar no momento. Por exemplo, eu irei criar uma janela principal no **Código R** 4.3 e ocultá-la. Para isso, será realizado a função `tclServiceMode(FALSE)` antes da função `tktoplevel()`. Ao final de toda a criação da interface, para exibir a janela principal basta executar `tclServiceMode(TRUE)` , ver **Código R** 4.3.

Código R 4.3

Script:

```
1  # Ocutar a IGU
2  tclServiceMode(FALSE)
3  # Criacao da janela principal
4  mcpprincipal <- tktoplevel()
5  # Demais widgets
6  # ...
7  # Exibir os widgets, bem como a janela principal
8  tclServiceMode(TRUE)
```

Um outro exemplo, para verificar que esse comando é utilizado para qualquer componente (elemento gráfico, *widget*), veja o **Código R** 4.4.

Código R 4.4

Script:

```
1  # Janela principal
2  mcpprincipal  <- tktoplevel()
3  # Ocultar os widgets seguintes (4)
4  tclServiceMode(FALSE)
5  # Criando um label
6  lbl <- tklabel(mcpprincipal, text = ''Oi, mundo!'')
7  # Carregando lbl
8  tkpack(lbl)
9  # Exibindo os widgets a partir de (4)
10 tclServiceMode(TRUE)
```

Pelo Exemplo dado no **Código R** 4.4, percebam que o widget ocultado foi o objeto `lbl`, e não a janela principal (objeto `mcpprincipal`). Para perceber a diferença, execute o **Código R** 4.4 sem as **linhas 5 e 14**.

Contudo, se for do interesse ocultar uma janela principal já criada (`tktoplevel()`) , pode ser usado a função `tkwm.withdraw()` e reverter usando `tkwm.deiconify()`, veja o **Código R** 4.5.

Código R 4.5

Script:

```
# Criacao da janela principal
mcpprincipal <- tktoplevel()
# Ocutar a IGU
tkwm.withdraw(mcpprincipal)
# Demais widgets
# ...
# Exibir a janela principal
tkwm.deiconify(mcpprincipal)
```

Vejam que os **Códigos R** 4.3 e 4.5 são equivalentes. Como sugestão para desenvolvimento de pacotes em *R*, sugiro utilizar o **Código R** 4.3. De modo similar ao **Código R** 4.5, podemos utilizar a função `tkwm.state()`, **Código R** 4.6.

Código R 4.6

Script:

```
# Criacao da janela principal
mcpprincipal <- tktoplevel()
# Ocutar a IGU
tkwm.state(mcpprincipal, "withdraw")
# Demais widgets
# ...
# Exibir a janela principal
tkwm.state(mcpprincipal, "normal")
```

No **Código R** 4.6, para verificar o estado da janela `mcpprincipal`, use `tkwm.state(mcpprincipal)`, e será retornado `"normal"`, caso esteja visível, ou `"withdraw"`, caso esteja oculto.

As outras opções são `"zoomed"` e `"iconic"` . O primeiro minimiza a janela e o segunda maximiza a janela.

4.3 Tamanho da janela principal

Para dimensionar a janela principal, usamos os argumentos `width` e `height` em `tktoplevel()`, ver Código 4.7.

Código R 4.7

Script:

```
# Dimensionando a janela principal
mcpprincipal <- tktoplevel(width = 500, height = 500)
# Verificando a dimensao de mcpprincipal
tkwm.geometry(mcpprincipal)
```

Console:

```
> <Tcl> 500x500+130+130
```

Para verificar a dimensão da janela principal, use `tkwm.geometry()` , **Código R** 4.7. Observe que nesse mesmo código foi apresentado o console do comando `tkwm.geometry(mcpprincipal)`. Os dois primeiros números, `500x500` representam a largura e altura da janela principal. Os dois últimos números representam a posição da janela na tela do computador.

Dependendo da quantidade de componentes (*widgets*) criado na sua interface, pode ser necessário restringir o dimensionamento mínimo, máximo ou nenhuma alteração, ou ainda escolher uma razão de dimensionamento entre largura e altura. Para isso veja o **Código R** 4.8.

Código R 4.8

Script:

```
# Bloquear o redimensionamento da janela
tkwm.resizable(mcpprincipal, TRUE, FALSE) # Desbloqueio nas duas dimensoes
tkwm.resizable(mcpprincipal, TRUE, FALSE) # Bloqueio na altura
tkwm.resizable(mcpprincipal, FALSE, TRUE) # Bloqueio na largura
tkwm.resizable(mcpprincipal, TRUE, TRUE) # Bloqueio na largura e altura
# Valores minimos para o redimensionamento (em pixels)
tkwm.minsize(mcpprincipal, 100, 100) # Nesse exemplo ficou:
                                     # largura 100 pixels e
                                     # altura 100 pixels
# Valores maximos para o redimensionamento (em pixels)
tkwm.maxsize(mcpprincipal, 500, 500)
# Redimensionamento de acordo com uma razao entre largura/altura
tkwm.aspect(mcpprincipal, 4, 4, 4, 4) #tkwm_aspect(janela, min_numerador,
    min_denominador, max_numerador, max_numerador)
# Voltar o redimensionamento de acordo com uma razao entre largura/altura
tkwm.aspect(mcpprincipal, "", "", "", "")
```

Segundo Franca (2005), a função `tkwm.aspect()` é aparentemente inútil, uma vez que você pode redimensionar uma janela com o mouse da forma como bem entender.

Muito embora fazendo testes sobre a função `tkwm.aspect()`, no sistema operacional Windows, não conseguimos perceber a real utilidade desta função.

4.4 Hierarquia de janelas

Em algum momento na IGU, podemos estar interessados em construir uma janela filha da janela principal. Para isso, vejamos o Código R XX

`tktoplevel(parent =)` e `tkwm.transient(child, parent)`.

5 Componentes de uma janela

Descrever os tipos de componentes de uma janela como frames, labels, botões, notebooks e panegroups, menus, entre outros.

5.1 Organizador de componentes visíveis de IGU

Essa seção é destinada aos *widgets* do tipo *frames* (significado, quadros), cuja finalidade é organizar os componentes de sua IGU, como por exemplo, botões, entradas de textos, *labels*, etc.. A função para esse tipo de *widget* é chamado `ttkframe` . As opções do `ttkframe` são apresentadas na Tabela 2. Lembrando que a função `tkframe` é equivalente a `ttkframe`.

Tabela 2: Opções da função `ttkframe`.

Opções	Finalidade
`width` e `height`	Largura e altura do quadro (*frame*) desenhado, respectivamente.
`padding`	Destinado ao espaço entre as bordas do quadro e outros componentes na janela.
`borderwidth`	especifica a largura da borda do quadro.
`relief`	Estilo da borda do quadro com efeito 3D. Opções: "flat" (*default*), "groove", "raised", "ridge", "solid" e "sunken".

Os dois primeiros argumentos dentre as opções da Tabela 2 para a função `ttkframe` serão `width` e `height` . Essas duas opções são responsáveis pela largura e altura do quadro a ser desenhado.

Código R 5.1

Script:

```
# Janela principal:
mcpprincipal <- tktoplevel()
# Primeiro quadro (ttkframe)
frame1 <- ttkframe(mcpprincipal,
                   width = 200, # largura em pixel
                   height = 50, # comprimento em pixel
                   relief = "raised");tkpack(frame1)
# Segundo quadro (ttkframe)
frame2 <- ttkframe(mcpprincipal,
                   width = 500,
                   height = 200,
                   relief = "raised");tkpack(frame2)
```

Para que as opções fossem visíveis acrescentamos mais uma opção, o argumento `relief` já apresentado na Tabela 2. Contudo, percebemos que esse argumento não funciona bem com a função `tkframe`. Já o argumento `borderwidth` funciona melhor na função `tkframe` do que usá-lo no `ttkframe`. Observe o Código 5.4.

Código R 5.2

Script:

```
# Janela principal:
mcpprincipal <- tktoplevel()
# Terceiro quadro (tkframe)
frame3 <- ttkframe(mcpprincipal,
                   width = 200, # largura em pixel
                   height = 50, # comprimento em pixel
                   relief = "raised",
                   borderwidth = 5);tkpack(frame3)
# Quarto quadro (tkframe)
frame4 <- tkframe(mcpprincipal,
                   width = 200,
                   height = 50,
                   relief = "raised",
                   borderwidth = 5);tkpack(frame4)
# Quinto quadro (tkframe)
frame5 <- tkframe(mcpprincipal,
                   width = 200,
                   height = 50,
                   relief = "raised",
                   borderwidth = 10);tkpack(frame5)
```

O último argumento apresentado é o `padding`. Como falado na Tabela 2, esse argumento permite a especificação de espaço entre pixels entre o texto do *label* e o limite do *widget*. Esse argumento tem uma entrada de valores concatenados, isto é, `ttkframe(parent, padding = c(3, 3, 12, 12))`, em que estes valores representam as posições à esquerda, superior, à direita e inferior, respectivamente, o espaço em pixels. Vejamos exemplo no Código 5.3.

Código R 5.3

Script:

```
# Janela principal:
mcpprincipal <- tktoplevel(width = 500, height = 500)
# Tamanho minimo da janela principal
tkwm.minsize(mcpprincipal, 500, 500)
# Terceiro quadro (tkframe)
frame <- ttkframe(mcpprincipal,
                  width = 200, # largura em pixel
                  height = 50, # comprimento em pixel
                  relief = "raised",
                  borderwidth = 5,
                  padding = c(0, 0, 0, 0));tkpack(frame)
label <- ttklabel(frame, text = "padding = c(0, 0, 0, 0)")
tkpack(label)
frame <- ttkframe(mcpprincipal,
                  width = 200, # largura em pixel
                  height = 50, # comprimento em pixel
                  relief = "raised",
                  borderwidth = 5,
                  padding = c(10, 10, 10, 10));tkpack(frame)
label <- ttklabel(frame, text = "padding = c(10, 10, 10, 10)")
tkpack(label)
frame <- ttkframe(mcpprincipal,
                  width = 200, # largura em pixel
                  height = 50, # comprimento em pixel
                  relief = "raised",
                  borderwidth = 5,
                  padding = c(100, 100, 100, 100));tkpack(frame)
label <- ttklabel(frame, text = "padding = c(100, 100, 100, 100)")
tkpack(label)
```

Uma outra função interessante é a `ttklabelframe`. Nessa função pode ser inserido um título para o quadro específico pelo argumento `text`. O posicionamento do título pode ser escolhido pelo argumento `labelanchor` com possíveis opções "n" (superior), "e" (direito), "w" (esquedo) e "s" (inferior). O *default* é "nw" (superior esquerdo).

Código R 5.4

Script:

```
mcpprincipal <- tktoplevel()
frame <- ttklabelframe(mcpprincipal,
                    text = "Superior␣central",
                    labelanchor = "n",
                    width = 200, # largura em pixel
                    height = 50, # comprimento em pixel
                    relief = "raised",
                    borderwidth = 5);tkpack(frame)
frame <- ttklabelframe(mcpprincipal,
                        text = "Inferior␣centro",
                        labelanchor = "s",
                        width = 200, # largura em pixel
                        height = 50, # comprimento em pixel
                        relief = "raised",
                        borderwidth = 5);tkpack(frame)
frame <- ttklabelframe(mcpprincipal,
                        text = "Direita␣centro",
                        labelanchor = "e",
                        width = 200, # largura em pixel
                        height = 50, # comprimento em pixel
                        relief = "raised",
                        borderwidth = 5);tkpack(frame)
frame <- ttklabelframe(mcpprincipal,
                        text = "Esquerda␣centro",
                        labelanchor = "w",
                        width = 200, # largura em pixel
                        height = 50, # comprimento em pixel
                        relief = "raised",
                        borderwidth = 5);tkpack(frame)
frame <- ttklabelframe(mcpprincipal,
                        text = "Superior␣esquerda",
                        labelanchor = "nw",
                        width = 200, # largura em pixel
                        height = 50, # comprimento em pixel
                        relief = "raised",
                        borderwidth = 5);tkpack(frame)
frame <- ttklabelframe(mcpprincipal,
                        text = "Inferior␣esquerda",
                        labelanchor = "sw",
                        width = 200, # largura em pixel
                        height = 50, # comprimento em pixel
                        relief = "raised",
                        borderwidth = 5);tkpack(frame)
```

Caso seja necessário inserir um separador entre os *widgets*, a função é `ttkseparator` . Esse separador pode ser colocado na horizontal ou vertical. Para isso, use o argumento `orient` que assume `"horizontal"` (*default*) ou `"vertical"`, Código 5.5

Código R 5.5

Script:

```
# Janela principal:
mcpprincipal <- tktoplevel()
# Primeiro label
label1 <- ttklabel(mcpprincipal, text = "Texto1"); tkpack(label1)
# Separador horizontal
sep <- ttkseparator(mcpprincipal, orient = "horizontal")
tkpack(sep, fill = "both")
# Segundo label
label2 <- ttklabel(mcpprincipal, text = "Texto_2"); tkpack(label2)
```

Script:

```
# Janela principal:
mcpprincipal <- tktoplevel()
# Primeiro label
label1 <- ttklabel(mcpprincipal, text = "Texto1")
tkgrid(label1, row = 0, column = 0)
# Separador vertical
sep <- ttkseparator(mcpprincipal, orient = "vertical")
tkgrid(sep, sticky = "ns", row = 0, column = 1)
# Segundo label
label2 <- ttklabel(mcpprincipal, text = "Texto_2")
tkgrid(label2, row = 0, column = 2)
```

Quando for inserir o separador (`separator()`), usando o `tkpack` ou `tkgrid`, é bom lembrar de inserir o argumento `fill` para a primeira função, e `sticky` para a segunda. Caso não insira, será desenhado um separador de tamanho 1 pixel, praticamente não perceptível na IGU. Para ver esses detalhes veja o Código 5.5.

6 Gerenciador de geometria dos componentes inseridos na IGU

Uma das formas coisa mais importantes após criar um *widget* é inseri-lo na janela principal. Há três funções para inserir um *widget* na janela principal. São as funções `tkpack` , `tkgrid` e `tkplace` . Observando os códigos até aqui, como também daqui para frente, vamos perceber que sempre ao final de cada código aparecerá uma dessas duas funções. Isso porque essas duas funções permitem controlarmos uma melhor posição geométrica entre os componentes da IGU. A diferença entre essas duas funções é a forma de entrar com os componentes, que será visto ao longo do material com os exemplos. Podemos mesclar essas duas funções numa mesma IGU, respeitando a hierarquia dos componentes.

A seguir, vamos apresentar cada uma separadamente.

6.1 Usando a função `tkpack()`

As opções para usar a função `tkpack()` estão apresentadas na Tabela 3. No Código 6.2, apresentamos as duas opções para o `tkpack()`, `after` e `before`.

Tabela 3: Opções para a função `tkpack()`.

Opções	Finalidade
`after`	Insere um componente depois do componente específico. O valor é o nome do objeto que deseja inserir o componente depois dele. Ex.: `ttkpack(child2, after = child)`, isto é, inserir o componente `child2` depois de `child1`.
`before`	Insere um componente criado antes do componente específico. O valor é o nome do objeto que deseja inserir o componente depois dele. Ex.: `ttkpack(child2, before = child)`, isto é, inserir o componente `child2` antes de `child1`.
`anchor`	Especifica a posição dos componentes na direção da bússola. Os valores são `"n"` (norte), `"ne"` (nordeste), `"e"` (leste), `"se"` (sudeste), `"s"` (sul), `"sw"` (sudoeste), `"w"` (oeste), `"nw"` (noroeste) e `"center"` (centro, é o *default*).
`expand`	Permite expandir um componente da IGU. Ele trabalha junto com o argumento `fill`. Os valores possíveis são `TRUE`, `FALSE` ou 0 (*default*).
`fill`	Juntamente com a opção `expand`, determina em que direção no plano cartesiano o componente se estica. Os valores são `"y"` (vertical), `x` (horizontal), `"both"` (as duas direções) ou `"none"` (nenhuma, é o *default*). Para o `fill` ser executado, devemos assumir que `expand = TRUE`.
`ipadx`	Exceto para o quadros de organização (*frames*), determina o espaço interno nos lados esquerdo e direito dos componentes. Assume valores nos reais positivos. Semelhante ao `padding` para os *frames*. O *default* é 0.
`ipady`	Exceto para o quadros de organização (*frames*), determina o espaço interno nos lados superior e inferior dos componentes. Assume valores nos reais positivos. Semelhante ao `padding` para os *frames*. O *default* é 0.
`padx`	Exceto para o quadros de organização (*frames*), determina o espaço ao redor dos lados direito e esquerdo dos componentes. Assume valores nos reais positivos. O *default* é 0.
`pady`	Exceto para o quadros de organização (*frames*), determina o espaço ao redor dos lados superior e inferior dos componentes. Assume valores nos reais positivos. O *default* é 0.
`side`	Determina aonde o componente será inserido. Os valores assumidos são `"left"` (esquerda), `"right"` (direita), `"top"` (superior, é o *default*) e `"bottom"` (inferior).
`in`	Identificação do componente onde deseja que a função `tkpack` insira o componente criado. O valor é um caractere. Esse argumento é opcional, pois quando criamos um componente já identificamos o componente pai. Uma vez que no *R* sempre criamos um objeto para receber esses componentes, esse argumento não será necessário para nossas aplicações.

Código R 6.1

Script:

```
1  # Janela principal
2  mcpprincipal <- tktoplevel()
3  # Primeiro componente
4  child1 <- tklabel(mcpprincipal, text = "child1");tkpack(child1)
5  # Segundo componente: tkpack -> before
6  child2 <- tklabel(mcpprincipal, text = "child2")
7  tkpack(child2, before = child1)
8  # Terceiro componente: tkpack -> after
9  child3 <- tklabel(mcpprincipal, text = "child3")
10 tkpack(child3, after = child1)
11 child4 <- tklabel(mcpprincipal, text = "child4")
12 ID <- child3$ID
13 tkpack(child4, after = ID)
```

Observe na linha 12 do Código 6.2, que no argumento `before` ou `after` podemos utilizar o nome do objeto R específico ou a identificação (ID) que esse componente recebeu. Todo objeto (componente) R feito por alguma função do pacote tcltk recebe uma identificação (ID) para melhor controle do desenho da interface.

A próxima opção é `anchor` . Esse argumento especifica a direção onde os componentes da IGU serão desenhados. Os valores para esse argumento é baseado na direção da bússola, que são `"n"` (norte), `"ne"` nordeste, `"e"` leste, `"se"` sudoeste, `"s"` sul, `"sw"` sudoeste, `"w"` oeste, `"nw"` noroeste e `"center"` centro.

Código R 6.2

Script:

```
1  # Janela principa
2  mcpprincipal <- tktoplevel()
3  # Usando a opcao (argumento) anchor em tkpack
4  n  <- tklabel(mcpprincipal, text = "n")
5  tkpack(n, anchor = "n")
6  ne <- tklabel(mcpprincipal, text = "ne")
7  tkpack(ne, anchor = "ne")
8  e  <- tklabel(mcpprincipal, text = "e")
9  tkpack(e, anchor = "e")
10 se <- tklabel(mcpprincipal, text = "se")
11 tkpack(se, anchor = "se")
12 s  <- tklabel(mcpprincipal, text = "s")
13 tkpack(s, anchor = "s")
14 sw <- tklabel(mcpprincipal, text = "sw")
15 tkpack(sw, anchor = "sw")
16 w  <- tklabel(mcpprincipal, text = "w")
17 tkpack(w, anchor = "w")
18 c  <- tklabel(mcpprincipal, text = "center")
19 tkpack(c, anchor = "center")
```

As opções seguintes são `expand` e `fill`. Vamos apresentá-las no Código 6.3.

Código R 6.3

Script:

```
1  # Janela principal
2  mcpprincipal <- tktoplevel()
3  #-----------------------------------------------------------------
4  # Primeiro label
5  frame <- ttkframe(mcpprincipal);tkpack(frame, expand = TRUE, fill = "both")
6  label <- ttklabel(frame, text = "expand=TRUE,fill='both'")
7  tkpack(label, side = "left")
8  # Primeira entrada
9  entrada <- ttkentry(frame)
10 tkpack(entrada, side = "left", expand = TRUE, fill = "both")
11 #-----------------------------------------------------------------
12 # Segundo label
13 frame <- ttkframe(mcpprincipal);tkpack(frame, expand = TRUE, fill = "x")
14 label <- ttklabel(frame, text = "expand=TRUE,fill='x'")
15 tkpack(label, side = "left")
16 # Primeira entrada
17 entrada <- ttkentry(frame)
18 tkpack(entrada, side = "left", expand = TRUE, fill = "x")
19 #-----------------------------------------------------------------
20 # Segundo label
21 frame <- ttkframe(mcpprincipal);tkpack(frame, expand = TRUE, fill = "y")
22 label <- ttklabel(frame, text = "expand=TRUE,fill='y'")
23 tkpack(label, side = "left")
24 # Primeira entrada
25 entrada <- ttkentry(frame)
26 tkpack(entrada, side = "left", expand = TRUE, fill = "y")
```

Algo bem interessante no Código 6.3, é a inserção dos componentes da IGU usando o `tkpack()` . O padrão (*default*) dessa função é inserir os componentes na IGU com o argumento `side = "top"` . Isso significa que eles serão empilhados, um abaixo do outro, à medida que vão sendo inseridos pela função `tkpack()`. Entretanto, o interesse no Código 6.3 é apresentar três linhas de componentes na IGU, sendo que em cada linha teremos dois componentes sequenciados, um *label* e um quadro de entrada. Uma saída seria alterar o argumento `side = "left"` . Porém, isso iria desenhar todos os componentes em apenas uma única linha, e mais uma vez, não atingiríamos o objetivo. Assim, a saída nesse caso foi criarmos componentes hierarquizados para resolver o problema. Primeiro, criamos quadros de organização (*frame*), **linhas 5, 13 e 21** do **Código R** 6.3. Ao serem inseridos, usamos as configurações de `expand` e `fill` desejadas, e lembrando que por *default*, o argumento `side="top"`. No passo seguinte, inserimos os componentes desejados (`label` e `entrada`). Observe que ao inserir esses componentes em cada um dos `frame` específico, o argumento `side` em `tkpack()` foi alterado para `side="left"` . Isso significa que eles foram inseridos sequenciados na mesma linha pela esquerda. Dessa forma, atingimos o objetivo e pode ser verificado ao executar o código.

Agora, retornando a explicação dos argumentos `expand` e `fill` , o usuário ao executar o **Código R** 6.3, poderá redimensionar a janela principal com o próprio cursor, aumentando e diminuindo, e verá o resultado de cada um dos valores assumidos para esses argumentos.

É bom lembrar que os argumentos argumentos `expand` e `fill` em `tkpack()` tiveram o objetivo de configurar os objetos `entrada`. Contudo, se sua hierarquia superior, no caso os `frame`, não tivessem sido inseridos na IGU com essa configuração, os objetos `entrada` não apresentariam as configurações

desejadas.

Uma outra forma de entrada desses componentes, seria usando a função `tkgrid()` . Veremos na Subseção 6.2 que não precisaremos dos componentes do tipo *frame* (quadros de organização).

Para que fique claro o argumento `expand`, independente do argumento `fill` na função `tkpack`, vejamos o **Código R** 6.4.

Código R 6.4

Observe por esse código que ao criar uma janela principal com 200 pixels de largura e altura, usando o argumento `expand = TRUE`, os três botões criados se expandiram a essa dimensão.

Script:

```
1  #---------------------
2  # Usando o expand=TRUE
3  #---------------------
4  # Janela principal
5  tkpack.propagate(mcpprincipal <- tktoplevel(width = 200, height = 200),
       FALSE)
6  # Criando uma funcao para construir e inserir os botoes
7  fbotao <- function(text, expand) tkpack(tkbutton(mcpprincipal, text =
       text), expand = expand)
8  # Titulo dos botoes
9  text <- c("botao1", "botao2", "botao3")
10 # Usando a funcao sapply() para criar
11 # de uma vez tres botoes com expand = TRUE
12 sapply(text, fbotao, expand = TRUE)
```

Já nessa rotina seguinte, usando o argumento `expand = FALSE`, os três botões criados não se expandiram a dimensão da janela principal. Para executá-la, rode inicialmente as linhas 5-9, e posteriormente, rode essa rotina seguinte.

Script:

```
1  #---------------------
2  # Usando o expand=FALSE
3  #---------------------
4  sapply(text, fbotao, expand = FALSE)
```

Observe que os botões não se expandiram ao longo da janela principal.

Uma função interessante no **Código R** 6.4 é `tkpack.propagate()`. Essa função na rotina permitiu com que a janela principal não se ajustasse aos botões criados, isto é, ela preservou suas dimensões determinadas inicialmente. Para mais detalhes, ver os **Códigos R** 6.11 e 6.12. Um último detalhe no **Código R** 6.4 foi que os botões se expandiram na vertical. Isso ocorreu por o padrão (*default*) da função `tkpack()` para o argumento `side` é igual a `"top"`. Caso desejasse que esses botões fosse dispostos na horizontal, começando da esquerda para à direita, bastaria usar `side = "left"` em `tkpack()`, ver **Código R** 6.6.

Dando sequência a mais um conjunto de opções para a função `tkpack`, temos `padx`, `pady`, `ipadx` e `ipady`. Essas opções são similares ao argumento `padding` para as funções `ttkframe` ou `tkframe` . Portanto, as opções do tipo `pad` não se aplicam aos quadros de organização, mas aos seus filhos (componentes), como botões, quadros de entrada, etc. O prefixo `i-` representam as distâncias internas dos componentes aos seus rótulos (*labels*), caso contrário, teremos as distâncias externas. Para o sufixo `-x`, teremos as distâncias dos lados direito e esquerdo e para o sufixo `-y`, teremos as distâncias

da parte superior e inferior. Para mais detalhes, veja o Código 6.5 e a Tabela 3.

Código R 6.5

Script:

```
# Janela principal
mcpprincipal <- tktoplevel()
#-----------------------------------
# Primeiro quadro de organizacao
frame  <- ttkframe(mcpprincipal);tkpack(frame)
# Exemplos para o ipadx
for (i in c(1, 5, 10, 15, 20)) {
  texto <- paste("ipadx=", i, sep ="");botao <- tkbutton(frame, text =
   texto)
  tkpack(botao, side = "left", ipadx = i)
}
```

Script:

```
# Janela principal
mcpprincipal <- tktoplevel()
#-----------------------------------
# Primeiro quadro de organizacao
frame  <- ttkframe(mcpprincipal);tkpack(frame)
# Exemplos para o padx
for (i in c(1, 5, 10, 15, 20)) {
  texto <- paste("padx=", i, sep = "")
  botao <- tkbutton(frame, text = texto)
  tkpack(botao, side = "left", padx = i)
}
```

Script:

```
# Janela principal
mcpprincipal <- tktoplevel()
#-----------------------------------
# Terceiro quadro de organizacao
frame  <- ttkframe(mcpprincipal);tkpack(frame)
# Exemplos para o ipady
for (i in c(1, 5, 10, 15, 20)) {
  texto <- paste("ipady=", i, sep = "")
  botao <- tkbutton(frame, text = texto)
  tkpack(botao, side = "top", ipady = i)
}
```

Script:

```
mcpprincipal <- tktoplevel()
#-----------------------------------
# Quarto quadro de organizacao
frame  <- ttkframe(mcpprincipal);tkpack(frame)
# Exemplos para o pady
for (i in c(1, 5, 10, 15, 20)) {
  texto <- paste("pady=", i, sep = "")
  botao <- tkbutton(frame, text = texto)
  tkpack(botao, side = "top", pady = i)
}
```

Por fim, temos a opção `side` , como já foi introduzido no Código 6.3. Na Tabela 3, apresentamos os valores assumidos para esse argumento, lembrando que `side = "top"` é o padrão (*default*). Esse argumento é o que determina a entrada dos componentes na janela principal. Deixando o padrão, a entrada dos componentes serão um abaixo do outro. Caso queira uma entrada sequenciada de modo horizontal, dependendo da direção. Veja o Código 6.6.

Código R 6.6

O primeiro script é um exemplo para `side = "top"`.

Script:

```
# Janela principal
mcpprincipal <- tktoplevel()
# Inserindo labels (rotulos)
for (i in 1:3) {
  label <- tklabel(mcpprincipal, text = paste("Texto", i, sep = ""))
  tkpack(label, side = "top")
}
```

O próximo exemplo no código é usado para `side = "bottom"`.

Script:

```
# Janela principal
mcpprincipal <- tktoplevel()
# Inserindo labels (rotulos)
for (i in 1:3) {
  label <- tklabel(mcpprincipal, text = paste("Texto", i, sep = ""))
  tkpack(label, side = "bottom")
}
```

O penúltimo código é usado para `side = "left"`.

Script:

```
1 # Janela principal
2 mcpprincipal <- tktoplevel()
3 # Inserindo labels (rotulos)
4 for (i in 1:3) {
5   label <- tklabel(mcpprincipal, text = paste("Texto", i, sep = ""))
6   tkpack(label, side = "left")
7 }
```

Por fim, o último exemplo do código é usado para `side = "right"`.

Script:

```
1 # Janela principal
2 mcpprincipal <- tktoplevel()
3 # Inserindo labels (rotulos)
4 for (i in 1:3) {
5   label <- tklabel(mcpprincipal, text = paste("Texto", i, sep = ""))
6   tkpack(label, side = "right")
7 }
```

Ainda sobre esse tipo de entrada de componentes, apresentamos mais cinco funções que podem ser importantes: `tkpack.info()`, `tkpack.configure()`, `tkpack.forget()`, `tkpack.propagate()` e `tkpack.slaves()`. Essas funções permitem fazermos alterações, verificar as configurações, por exemplo, dos componentes já inseridos na interface.

A primeira função é `tkpack.info()` . Esta função permite sabermos qual a configuração do objeto inserido. Esses objetos excluem os componente do tipo janelas (`ttktoplevel()`), que devem usar a função `tkwinfo()` , Seção 7. Verificando o último **Script** do Código R 6.6, e as configurações do objeto `label`, vejamos a aplicação da função `tkpack.info()` no Código 6.7.

Código R 6.7

Assumimos, que a última rotina do Código 6.6 foi executada.

Script:

```
1 # Informacoes sobre o objeto label
2 tkpack.info(label)
```

Console:

```
> <Tcl> -in .14 -anchor center -expand 0 -fill none -ipadx 0 -ipady 0 -padx 0
  -pady 0 -side right
```

Observe pelo **Código R** 6.7, que única opção inserida para o objeto `label` do **Código R** 6.6, **linha 6**, foi o `side = "right"`. As demais opções são *default*.

A segunda função `tkpack.configure()` . Essa função permite reconfigurarmos um componente já criado. Vejamos o **Código R** 6.8.

Código R 6.8

Nesse **Script**, usamos a função `tkpack.info()` para sabermos as características do objeto `botao`.

Script:

```
# Janela principal
mcpprincipal <- tktoplevel()
# Inserindo um botao
botao <- tkbutton(mcpprincipal, text = "botao1"); tkpack(botao)
# Informacoes sobre o objeto botao
tkpack.info(botao)
```

Console:

```
> <Tcl> -in .19 -anchor center -expand 0 -fill none -ipadx 0 -ipady 0 -padx 0
  -pady 0 -side top
```

Aqui, verificamos a finalidade da função `tkpack.configure()`.

Script:

```
# Reconfigurando a opcao fill e expand
tkpack.configure(botao, fill = "both", expand = TRUE)
# Revendo as informacoes do objeto botao
tkpack.info(botao)
```

Console:

```
> <Tcl> -in .19 -anchor center -expand 1 -fill both -ipadx 0 -ipady 0 -padx 0
  -pady 0 -side top
```

Uma rotina apresentada por Lawrence e Verzani (2004) também pode ilustrar bem a função `tkpack.configure`, veja o Código 6.9.

Código R 6.9

Essa rotina foi extraída de Lawrence e Verzani (2004, p. 376). Essa rotina apresenta uma outra importante função que é `tkwinfo()`. Essa função apresenta um padrão um pouco diferente das outras funções do pacote tcltk. O primeiro argumento não é o componente *pai*, como discutido na Seção 2. Temos a Seção 7 específica para esses tipos de funções mais gerais.

Script:

```
window <- tktoplevel()
tkwm.title(window, "tkpack.configure")
frame <- ttkframe(window , padding = c(3 ,3 ,12 ,12))
tkpack(frame, expand = TRUE, fill = "both")
##
pack_btn <- function(txt, ...) {
  tkpack(button <- ttkbutton(frame, text = txt ), ...)
}
##
pack_btn("Top", side = "top", expand = TRUE, fill = "both")
pack_btn("Bottom", side = "bottom" , expand = TRUE, fill = "both")
pack_btn("Left", side = "left", expand = TRUE, fill = "both")
pack_btn("Right", side = "right", expand = TRUE, fill = "both")
##
children <- as.character(tkwinfo("children", frame))
sapply(children , tkpack.configure , fill = "none")
```

A segunda terceira função é `tkpack.forget()` . Essa função remove algum componente inserido com a função `tkpack()`. Veja no **Código R** 6.10. Essa função é similar a função `tkgrid.forget()` para a função `tkgrid` .

Código R 6.10

Script:

```
# Janela principal
mcpprincipal <- tktoplevel()
# Criacao de um botao
botao <- tkbutton(mcpprincipal, text = "botao1");tkpack(botao)
# Removendo o botao
tkpack.forget(botao)
```

A penúltima função é `tkpack.propagate()`. Esta função auxilia no controle das características comprimento e largura das janelas ao criar seus componentes filhos. Por exemplo, veja o **Código R** 6.11. Observe que criamos uma janela principal (`mcpprincipal`) com características específicas, comprimento de 200 pixels e altura de 100 pixels. Contudo, ao criar um botão e inseri-lo (`tkpack`) em `mcpprincipal`, a função `tkpack` ajusta (propaga) a janela principal ao tamanho do botão.

Código R 6.11

Script:

```
# Janela principal
mcpprincipal <- tktoplevel(width = 200, height = 100)
# Criando um botao
tkpack(tkbutton(mcpprincipal, text = "Oi"))
```

Para que isso não ocorra, usamos a função `tkpack.propagate(parent, FALSE)` com a opção

FALSE. Assim, nessa função basta informar a função qual o componente que deseja a não propagação (FALSE). Por *default*, os componentes sempre propagam, isto é, `tkpack.propagate(parent, TRUE)` . Portanto, os valores possíveis para essa função são TRUE ou FALSE. Veja o **Código R** 6.12.

Código R 6.12

Script:

```
1 # Janela principal
2 mcpprincipal <- tktoplevel(width = 200, height = 100)
3 # Desabilitamos a propagacao de mcpprincipal
4 tkpack.propagate(mcpprincipal, FALSE)
5 # Criando um botao
6 tkpack(tkbutton(mcpprincipal, text = "Oi"))
```

Por fim, a última função do gerenciador *pack* é `tkpack.slaves()`. Essa função retorna uma lista de ID com todas os componentes filhos dos componentes pais informado na função. Observe o **Código R** 6.13 e veja as linhas 8-9. Usamos a função `tkpack.slaves()` para sabermos a ID dos componentes filhos dos respectivos componentes pais (`mcpprincipal e frame`). Essa função é similar a função `tkwinfo("children", parent)` , sendo que `parent` representa os componentes pais. O resultado final é o mesmo

Código R 6.13

Script:

```
1 # Janela principal
2 mcpprincipal <- tktoplevel()
3 # Criando um quadro de organizacao
4 tkpack(frame <- tkframe(mcpprincipal))
5 # Criando um botao
6 tkpack(button <- tkbutton(frame, text = "Oi␣mundo!"))
7 # Vericando os filhos
8 tkpack.slaves(frame)
9 tkpack.slaves(mcpprincipal)
```

6.2 Usando a função `tkgrid()`

6.3 Usando a função `tkplace()`

7 Funções gerais do pacote tcltk

Uma função interessante é `tkwinfo()` . Essa função apresenta uma estrutura de função diferente das funções básicas do tcltk, porque essa função tem como primeiro argumento a característica desejada sobre o objeto de interesse. Logicamente, esse objeto deve ser uma função do pacote tcltk. Para exemplificarmos, vejamos a **linha 15** do **Código R** 6.9. O objeto `frame` criado na **linha 3**, é um componente *pai* em relação aos botões (componentes *filhos*) criados posteriormente na rotina. Assim, na **linha 15**, desejamos saber quais são as identificações (Veja o **Código R** 6.2) dos quatro botões criados. Precisamos dessas identificações porque os componentes foram criados sem a criação

Tabela 4: Opções para a função `tktkgrid()`.

Opções	Finalidade
`after`	Destinado a inserir o componente depois do componente específico. O valor é o nome do objeto que deseja inserir o componente depois dele. Ex.: `ttkpack(child2, after = child)`, isto é, inserir o componente `child2` depois de `child1`.
`before`	Destinado a inserir o componente antes do componente específico. O valor é o nome do objeto que deseja inserir o componente depois dele. Ex.: `ttkpack(child2, before = child)`, isto é, inserir o componente `child2` antes de `child1`.
`anchor`	Argumento especificado na IGU na direção da bússola. Os valores são `"n"` (norte), `"ne"` (nordeste), `"e"` (leste), `"se"` (sudeste), `"s"` (sul), `"sw"` (sudoeste), `"w"` (oeste), `"nw"` (noroeste) e `"center"` (centro, é o *default*).
`expand`	Esse argumento permite expandir o componente na IGU. Ele trabalha junto com o argumento `fill`. Os valores possíveis são `TRUE`, `FALSE` ou 0 (*default*).
`fill`	Juntamente com a opção `expand`, determina em que direção no plano cartesiano o componente se expande. Os valores são `"y"` (vertical), `x` (horizontal), `"both"` (as duas direções) ou `"none"` (nenhuma, é o *default*). Para o `fill` ser executado, devemos assumir que `expand = TRUE`.
`ipadx`	Exceto para o quadros de organização (*frames*), espaço interno nos lados esquerdo e direito dos componentes. Assume valores nos reais positivos. Semelhante ao `padding` para os *frames*. O *default* é 0.
`ipady`	Exceto para o quadros de organização (*frames*), espaço interno nos lados superior e inferior dos componentes. Assume valores nos reais positivos. Semelhante ao `padding` para os *frames*. O *default* é 0.
`padx`	Exceto para o quadros de organização (*frames*), espaço ao redor dos lados direito e esquerdo dos componentes. Assume valores nos reais positivos. O *default* é 0.
`pady`	Exceto para o quadros de organização (*frames*), espaço ao redor dos lados superior e inferior dos componentes. Assume valores nos reais positivos. O *default* é 0.
`side`	Argumento que indica aonde o componente será inserido. Os valores assumidos são `"left"` (esquerda), `"right"` (direita), `"top"` (superior, é o *default*) e `"bottom"` (inferior).
`in`	Identificação do componente *pai*. O valor é um caractere.

dos seus respectivos objetos (observe as **linhas 11-13**, **Código R** 6.9, que só temos a criação dos componentes, sem armazená-los em um objeto).

Uma outra rotina que pode exemplificar a aplicação da função `tkwinfo()` é o **Código R** 6.5. Nas **linhas 7-10**, do primeiro **Script**, criamos um *loop* para gerar cinco botões. Nesse caso, os cinco botões são atribuídos ao mesmo no nome de objeto. Isto significa que qualquer informação do tipo `botao$ID`, informará apenas características do último botão criado. Vejamos o **Código R** 7.1. Esses resultados mostram que apenas o último componente é informado sobre suas características.

Código R 7.1

Os resultados dessa rotina foram baseadas no primeiro **Script** do **Código R** 6.6.

Console:

```
> botao$ID
> ".22.1.5"
> ##
> tkpack.info(botao)
> <Tcl> -in .22.1 -anchor center -expand 0 -fill none -ipadx 20 -ipady 0 -padx
    0 -pady 0 -side left
> ##
> tkwinfo("children", frame)
> <Tcl> .22.1.1 .22.1.2 .22.1.3 .22.1.4 .22.1.5
```

Voltando ao **Código R** 6.9, com o auxílio da função `tkwinfo()`, pedimos os componentes *filhos* (`"children"`) do objeto `frame`, e o transformamos como caractere, **linha 15**. Assim, a partir do objeto `children` usamos as identificações dos botões na função `sapply()` para a função `tkpack.configure()` de forma vetorizada a alteração do argumento `fill`. Veja no **Código R** 7.2 o *output* da **linha 15** do **Código R** 6.9, sem armazená-lo no objeto `children`.

Código R 7.2

Esse *output* está baseado na rotina do **Código R** 6.9, **linha 15**.

Console:

```
> as.character(tkwinfo("children", frame))
> ".23.1.1" ".23.1.2" ".23.1.3" ".23.1.4"
```

Uma observação é que ao passo que criamos esse material, vamos criando as interfaces e testando os comandos. Obviamente, a identificação dos componentes não vão sendo as mesmas. Por exemplo, se você executar as rotinas para obter os **Código R** 7.1 e **Código R** 7.2, as identificações do componentes filhos resultarão em números diferentes, dependendo do número de janelas criadas.

Portanto, a estrutura da função `tkwinfo()` é:

```
tkwinfo("característica", parent)
```

A opção *parent* é componente *pai*, e a *característica* desejada a esse componente pode ser não só `"children"`, mas muitas outras. Essas características são apresentadas na Tabela 5. Lembrando que determinadas características são específicas para componentes específicos e devem está entre aspas na função.

Tabela 5: *Características* da função `tkwinfo()`.

cells	children	class	colormapfull	depth
geometry	height	id	ismapped	manager
name	parent	pointerx	pointery	pointerxy
reqheight	reqwidth	rootx	rooty	screen
screencells	screendepth	screenheight	screenwidth	screenmmheight
screenmmwidth	screenvisual	server	toplevel	viewable
visual	visualid	vrootheight	vrootwidth	vrootx
vrooty	width	x	y	atom
atomname	containing	interps	pathname	exists
fpixels	pixels	rgb	visualsavailable	

8 Desenvolvendo pacotes com IGU usando o pacote tcltk

9 Exemplos retirados de Lawrence e Verzani (2004)

9.1 Exemplo 17.1, página

9.2 Exemplo 18.1, página 371

Esse exemplo apresenta a construção de uma janela, verificando se existe uma janela principal ou não. Há um erro no algoritmo no livro, **linha 10**, **Código R** 9.1, e aqui foi corrigido.

Código R 9.1

Script:

```
1  newWindow <- function(title, command, parent, width, height) {
2    window <- tktoplevel()
3    if (!missing(title )) tkwm.title(window , title)
4    if (!missing(command ))
5      tkwm.protocol(window, "WM_DELETE_WINDOW" , function() {
6        if (command()) # command returns logical
7          tkdestroy(window)
8      })
9    if (!missing(parent )) {
10     parent_window <- tkwinfo("toplevel", parent)
11     if (as.logical(tkwinfo("viewable", parent_window))) {
12       tkwm.transient(window , parent)
13     }
14   }
15   if (!missing(width)) tkconfigure(window, width = width)
16   if (!missing(height)) tkconfigure(window, height = height)
17   window_system <- tclvalue(tcl("tk", "windowingsystem"))
18   if (window_system == "aqua") {
19     frame <- ttkframe(window, padding = c(3 ,3 ,12 ,12))
20   } else {
21     int_frame <- ttkframe(window, padding = 0)
22     tkpack(int_frame, expand = TRUE, fill = "both")
23     frame <- ttkframe(int_frame, padding = c(3 ,3 ,12 ,0))
24     sizegrip <- ttksizegrip(int_frame)
25     tkpack(sizegrip, side = "bottom", anchor = "se")
26   }
27   tkpack(frame, expand = TRUE, fill = "both", side = "top")
28   return(frame)
29 }
```

Referências

FRANCA, A. *Tcl/tk*: Programaccão linux. Rio de Janeiro: Brasport, 2005. 373 p.

LAWRENCE, M. F.; VERZANI, J. *Programming Graphical User Interfaces in R*. Boca Raton: CRC Press, 2004. 463 p. (The R Series).

Índice