Padrões de programação para a linguagem R e o pacote Shiny

Gabriel Alves Castro

1. Introdução

A linguagem de programação R é uma ferramenta poderosa para a prática da ciência de dados, podendo estar envolvida em grandes projetos de programação. Devido à crescente complexidade dos projetos que envolvem tal linguagem, torna-se imperativa a necessidade da criação de uma padrão de programação para os projetos, desde os trâmites do próprio código, até a maneira com a qual os scripts são estruturados em arquivos. Devido a parca literatura relativa a este processo de padronização, este documento busca estabelecer um padrão para gerar boas práticas de programação entre a equipe envolvida neste projeto. Desse modo, estabelece-se como objetivo final, o auxílo à criação de códigos legíveis, modularizados e eficientes(buscando sempre a complexidade polinomial).

2. Objetivos

Os objetivos deste padrão de programação são estabelecidos como:

-Isolamento conceitual das funcionalidades(relativos à estrutura de funções da linguagem estrutural) dos códigos em diferentes arquivos chave.

-Criação de legibilidade e clareza em torno do código.

-Manutenção da eficiência das funções, referente a ponderação relativa à capacidade de resposta às entradas possíveis (a complexidade de uma operação sempre deve ser a menor possível).

-Reaproveitamento de código.

-Adaptabilidade, e integração entre funções puras do R e programas Shiny.

-Criação de aplicativos Shiny bem modularizados, e de fácil manutenção.

-Criação de aplicativos Shiny de baixa exigência operacional.

2. Padrões gerais para a linguagem R

2.1 Código R e funções

-Nenhum comentário poderá possuir caracteres especiais.

-As funcoes que serao utilizadas pelos usuários finais podem conter até 5 palavras separadas por "_" em sua constituicao. O nome deve representar claramente a funcionalidade da função. Desse modo, entre as 5 palavras, ao menos duas devem ser verbos.

```r
#Esta eh uma funcao caule
gera_grafico_cadastrados_curso_qualquer <- function() {
  #O codigo da funcao
}
```

-toda variavel inicia com letra minuscula e é separada por "_" caso seja um frase, e por uma letra maiúscula caso possua duas palavras. O nome das variaveis e das funcoes na raiz(explicado mais abaixo) nao pode conter mais de 3 palavras.

```
quantidadeCadastrados <- 5000
cadastrados_dia_21 <- 2000
```

-Toda funcao deve indicar explicitamente as variaveis de entrada, o que faz e o formato das variaveis de entrada.

-Toda funcao deve conter um comentario explicando sua funcionalidade.

-Todo passo relevante, loop e funcao devem apresentar um comentario indicando qual a sua complexidade.

```
#-Esta funcao recebe uma coluna referente a alguma variavel de uma tabela
#-Retorna uma variavel modificada com todos o valores em branco substituidos por NA
#-Variavel deve ser a coluna de um data frame ou um vetor

valores_em_branco_para_NA <- function(variavel) {

  variavel[variavel == ""] <- NA #complexidade 2n
  return(variavel)

} #complexidade n(chao de 2n)
```

-Todo comentário referente ao código que não se relaciona ao funcionamento lógico de alguma parte integrante deve conter um comentário anterior no seguinte formato: " #*** “.

```
#***
#Lembrar que o arquivo mdl aqui utilizado foi do dia 21/08
```

-Todo passo nao claro deve conter um comentario explicando a sua funcionalidade.

```
#aqui estou excluindo um dos possíveis valores da coluna valores,
#pois segundo a bibliografia o mesmo nao eh util para nossos estudos
dataFrame <- data.frame(valores = NULL)
dataFrame$valores <- dataFrame$valores[dataFrame$valores != "valor1"]
```

-Toda funcao deve apresentar um comentário com a variável matemática para a complexidade de cada parâmetro.

```
funcao <- function(parametro1, parametro2) { #parametro1 = m, parametro2 = n


} #complexidade O(m, n)
```

-Todo comentário representando uma interface, ou um caminho deve possuir uma legenda em uppercase.

```
#***
#INTERFACE SHINY APP
```

3. Estruturação em diretórios e interfaces

Com o objetivo de isolar logicamente as partes do programa, de modo a facilitar a sua manutenção e permitir a qualquer momento a substituição de uma parte do código por outra, ou mesmo a retirada de uma parte do código, de modo que todo o restante continue funcionando, será adotado um padrão para a organização dos scripts em diretórios. Para entender melhor qual será o padrão empregado, podemos fazer uma analogia entre a estrutura de diretórios e a estrutura da raíz de uma árvore, até a mesma alcançar o seu caule.

3.1 Funções

Desse modo podemos definir as funções em três níveis distintos:

3.1.1 Função base:

São as funções que estarão localizadas na pasta mais profunda em uma determinada sequência de diretórios. Desse modo, funções presentes em diretórios superiores, deverão ser as que fazem uso das funções base de um determinado ramo de diretórios.

3.1.2 Função raíz:

São as funções que estão entre o topo diretório e a base. Elas são organizadas de maneira hierarquica, sendo que, se uma função raiz utiliza outra função raiz em sua construção, essa deverá estar ao menos um diretório acima, no ramo de diretórios dessas funções.

3.1.3 função caule:

Refere-se a função que será utilizada pelo usuário final de uma biblioteca. Por exemplo: Uma função que será usada para gerar um determinado gráfico que será exposto em uma aplicação shiny.

3.2 Interfaces

Para integrar e gerenciar essa estrutura de diretórios, será padronizada uma estrutura de interfaces, que também se subdividem em algumas clasificações. Primeiro, vamos definir uma interface: O que se chamará de interface é um conjunto entre uma pasta e um script (ambos de mesmo nome). A pasta conterá os scripts que estão abaixo nas ramificações de diretórios (ou ramificações da raíz). O script conterá as definições de entrada e saída comuns de uma interface, seguida dos "sources" que irão realizar a integração entre os scripts na floresta (o diretório de trabalho).

obs: No R, o caminho indicado em qualquer source, deve ser realizado de acordo com o diretório de trabalho. Desse modo, as pastas caule (como definidas abaixo), devem ser integradas ao diretório de trabalho para que possam ser utilizadas.

3.2.1 Caules e interfaces caule

Caule refere-se a todo diretório que será integrado no diretório floresta, o diretório caule possui todas as funções caule.

obs: O diretório floresta é o diretório de trabalho, que deverá ser utilizado para rodar o programa. Por exemplo: No shiny, o diretório floresta é o mesmo diretório que contém o script app.R

Acompanhando o diretório caule, haverá o script caule, que contém todos os sources e definições de interface necessárias. Para utilizar as funções do diretório caule, bastará realizar um "source" no script caule.

obs:Um diretório caule não precisa conter todas as funções caule. A criação de uma nova hierarquia pode criar um diretório raiz que não contenha nenhuma função, mas separe logicamente determinadas funções de outras presentes em sua hierarquia. Desse modo, uma função caule pode estar presente em qualquer parte da hierarquia, desde que ela não seja utilizada por nenhuma outra função acima dela nas ramificações.

Exemplo: Criação de uma biblioteca para a avaliação de sistema epidemiológicos (códigos do autor).

diretório caule: biblioteca script caule: biblioteca.R (logo abaixo)

```
#***
#Esta eh uma bibilioteca para automatizar a avaliacao quantitativa dos sistemas
#de informacoes que geram dados acerca das arboviroses

#***
#Padrao de programacao:

#-Aqui estariam as metricas principais do padrao de programacao, ou a indicacao
#do documento que contem os padroes de programacao utilizados no codigo

#Aqui podem ser incluidas as bibliotecas utilizadas no projeto

#***
#Um exemplo de uma definição simples de interface:
#Interface:
#-Entradas: data.frames e/ou colunas de data.frames
#-Retornos: data.frames ou inteiros com métricas de avaliação de sistemas de acordo com a entrada

#***
#FUNCIONAIS
#source("biblioteca/funcionais.R")
#observe que FUNCIONAIS eh um diretorio raiz, e funcionais.R eh um script raiz

#***
#QUALIDADE
#source("biblioteca/qualidade.R")
#observe que QUALIDADE eh um diretorio raiz, e qualidade.R eh um script raiz

#e assim por diante:

#***
#OPORTUNIDADE
#source("biblioteca/oportunidade.R")
#***
#REPRESENTATIVIDADE
#source("biblioteca/representatividade.R")

#***
#SENSIBILIDADE
#source("biblioteca/sensibilidade.R")
```

obs: Todos os sources estão comentados, pois os diretórios não foram realmente criados, sendo somente exemplos.

3.2.2 Raízes e interfaces raíz

Raiz é definida como o diretório raíz e o seu respectivo script raízl.

Os diretórios Raíz referem-se aos diretórios que fazem parte da hierarquia de um diretório caule. De acordo com os padrões definidos logo acima. Representam uma divisão lógica, aonde toda função relacionada pertencente ao mesmo diretório, deverá executar tarefas semelhantes. Por exemplo: O diretório “leitura” deverá conter somente funções e subdiretórios raízes que realizem tarefas de leitura de arquivos.

Os scripts raízes acompanham o diretório raíz, e são responsáveis por organizar as interfaces do diretório raíz,

como também fazer um chamado a todas as funções de hierarquia inferior que poderão ser utilizadas pelas funções presentes nesse diretório raíz, ou acima dele na hierarquia.

obs: O último diretório raíz presente em uma hierarquia guarda todas as funções base de sua respectiva hierarquia.

3.3 Padrões acerca da estrutura de diretórios

-Todo script deve apresentar o caminho para realizar o acesso a ele.

```
#***
#CAMINHO
#biblioteca/funcionais/limpa_valores_em_branco.R
```

-Os diretórios raíz e scripts raíz devem possuir o mesmo nome.

-Os diretórios caule e scripts caule devem possuir o mesmo nome.

-O diretório floresta deve ser bem definido.

-Os scripts caule e raíz devem conter interfaces bem definidas, que representem a sua divisão lógica.

```
#***
#INTERFACE FUNCIONAIS

#funcao limpa_valores_em_branco
#-Recebe uma coluna
#-Retorna a mesma coluna sem os valores em branco
```

obs: Deve ser realizado um estudo bibliográfico para conhecer maneiras de criar interfaces na linguagem R.

4. Padões de programação para o Shiny

Todo aplicativo shiny deverá funcionar por meio de um script do tipo app.R, o qual irá integrar pro meio da função NS, todas as ui.R e server.R.

4.1 Divisão por meio da função NS

Cada tab deverá possuir um script separado, utilizando a função NS, com seus próprios ui e server que será denominado função tab. Dentro de app.R, a ui e o server serão chamados como funções.

Exemplo:

ui <- dashboardBody(tags*head*(*tags*link(rel = “stylesheet”, type = “text/css”, href = “custom.css”)),

```
useShinyjs(),

tabItems(

  tabItem("home",

    homeTabUi("home_ui_server") #funcao NS que chama a ui

  )

)
```

)

server <- function(input, output, session) {

callModule(homeServer, “home_ui_server”) #funcao NS que chama o server

}

4.2 Padrões de codificação

-Toda função tab deverá possuir antes de todo o código um comentário indicando a que tab pertence. -Todo script deverá conter seus respectivos id’s, para facilitar a navegação. -Toda indicação de id, deverá indicar a que tipo de entrada ou saída refere-se.

```
#***
# Entradas:
# -home: tabItem- mae da ui;
# -usuarios_solicitados: textOutput- quantitativo de usuarios solicitados;
# -usuarios_matriculados: textOutput- quantitativo de usuarios matriculados;
# -usuarios_inscritos: textOutput- quantitativo de usuarios inscritos;
# -usuarios_qualificados: textOutput- quantitativo de usuarios qualificados;
```

4.3 Arquitetura de caules no shiny app

A arquitetura de caules pode ser facilmente seguida no shiny app. Sendo que o diretório floresta seria o mesmo diretório em que o script.R estará localizado. Para facilitar a integração entre as funções tab e outras funções caule, pode-se criar um diretório caule com as funções tab, e em sua hierarquia seriam integradas outros diretórios caule (que nesse caso se tornariam diretórios raíz), os quais poderão ter suas funções utilizadas dentro das funções tab. Por exemplo: Podemos adicionar um diretório caule pronto que possui funções capazes de gerar gráficos que uma função tab precisa.

Assim dentro do diretório floresta, podem ser criados diversos diretórios caule, com diferentes funções tabs, dados e outras funções. Assim, com as interfaces definidas, seria fácil dar manutenção ao aplicativo, como também trocar as suas partes integrantes.

4.3.1 Padrões para as funções utilizadas na lógica de reatividade do shiny app

-Toda etapa que possa ser definida como uma função, ou que possa se repetida durante o código, deverá possuir uma função caule para executá-la.

-Todo valor retornado em um output deve provir de uma função caule.

-Todo tratamento realizado no código deve provir de uma função caule.

-Quando necessário, deverão haver funções caule responsáveis por receber reactives que retornem um output específico, e fazer o tratamento necessário.

5. Fraquezas da arquitetura de caules, e futuros enriquecimentos

Podemos observar que a complexidade de gerenciar os sources é a maior dificuldade. Sendo que qualquer pequeno erro poderia danificar o funcionamento das funções caule. Também é necessário seguir uma sequência rigorosa de sources, de modo que tudo funcione adequadamente. Desse modo, a medida que utilizemos essa arquitetura, deveremos pensar em padrões de documentação dos caules e raízes, de modo a manter a fácil manutenção e modularidade do código, que é o objetivo final desta arquitetura.

Acerca dos padrões aqui estabelecidos aqui, devemos sempre reprensá-los a medida que avançamos com os nossos códigos.

6. Padrão para o armazenamento e registro das limpezas em arquivos

Esta etapa será escrita futuramente, a medida que a equipe avança nos conhecimentos acerca das tabelas presentes na base de dado. A ideia será registrar de maneira clara e padronizada quais as limpezas realizadas, e quais a variáveis utilizadas para realizar uma determinada medida estatística, ou filtro.