Programowanie w języku R. Model obiektowy w R

Aleksander Denisiuk
Uniwersytet Warmińsko-Mazurski
Olsztyn, ul. Słoneczna 54
denisjuk@matman.uwm.edu.pl

3 kwietnia 2023

Model obiektowy w R

Najnowsza wersja tego dokumentu dostępna jest pod adresem

`http://wmii.uwm.edu.pl/~denisjuk/uwm`

Model obiektowy

Obiekty w R

- S3
  - funkcje generyczne
- S4
  - formalna definicja klas i dziedziczenia
  - przeciążenie funkcji generycznych
- Klasy referencyjne (reference classes, RC)
  - zbliżone do „zwykłego” programowania obiektowego
  - metody należą do klas
  - modyfikowane dane nie są kopiowane
- Typy podstawowe (base types)
  - podstawa innych danych
  - zazwyczaj opracowywane w C

Typy podstawowe

Typy podstawowe

- podstawą każgego obiektu R jest struktura danych C (`struct`)
  - określa jak jest umieszczany w pamięci
  - zawiera informację o *typie* obiektu
- Tylko *R core team* może tworzyć nowe typy podstawowe
  - ostatnio nowe typy były dodawane w latach 2011, 2005

Funkcja `typeof()`

- Typ podstawowy obiektu `typeof()`

```
v <- vector()
typeof(v)
m <- matrix()
typeof(m)
l <- list()
typeof(l)
d <- data.frame()
typeof(d)
f <- function() {}
typeof(f)
typeof(sum)
```

`typeof()` vs `is....()`


```
is.logical(v)
is.vector(v)
is.matrix(m)
is.list(l)
is.list(d)
is.data.frame(d)
is.function(f)
is.function(sum)
is.primitive(sum)
```

- `is.object(x)`
  - ✦ **TRUE** dla obiektów S3, S4, RC

S3

Wstęp

- Najprostszy możliwy model obiektowy na świecie
- Pierwszy zaimplementowany w R
- Jedyny używany w pakietach `base` oraz `stats`
- Najczęściej używany w pakietach z CRAN

Obiekty S3

- Czy `x` jest obiektem S3?

```
is.object(x) & !isS4(x)
```

- Biblioteka `pryr`

```
library(pryr)

df <- data.frame(x = 1:10, y = letters[1:10])
otype(df)
otype(df$x)
otype(df$y)
```

Funkcje generyczne

- Metody S3 są własnościami funkcji *generycznych*
  - nie klas czy obiektów
- Wybór metody na podstawie klasy
  - w kodzie R w jawny sposób `UseMethod()`

    `mean`
  - biblioteka `pryr`

    ```
    library(pryr)
    ftype(mean)
    ```

Wewnętrzne funkcje generyczne

- Zaimplementowane w C
  - ✦ brak wywołania `UseMethod()`

```
?"internal generic"
ftype(sum)
ftype(cbind)
```

Wybór odpowiedniej metody

- Funkcja `generic.class()`
- Prykładowo:
  - `mean.Date()`
  - `print.factor()`
- problemy:
  - `print.data.frame()`
  - `t.test()`
    - `ftype()`
    - unikać kropek w nazwach

Wyliczanie

- Wszystkie metody funkcji generycznej: `methods()`

```
methods("mean")
methods("t.test")
methods("axis")
```

- Wszystkie metody klasy: `methods(class=...)`

```
methods(class="ts")
```

Deklaracja klas i obiektów

- Jeden krok

```
foo <- structure(list(), class = "foo")
```

  - najczęściej obiekty S3 są tworzone na podstawie listy, wektora bądź funkcji
  - nazwa: małe litery, podkreślnik, unikać kropki

- Dwa kroki

```
foo <- list()
class(foo) <- "foo"
```

- Klasa obiektu: `class(object)`

```
class(foo)
```

  - może być wektor (od bardziej do mniej szczególnej)

```
mod <- glm(mpg ~ disp, data = mtcars)
class(mod)
```

Konstruktor

- Można skontrolować dane początkowe

```
foo <- function(x) {
 if (!is.numeric(x)) stop("X must be numeric")
 structure(list(x), class = "foo")
}
foo(1:100)
foo("tekst")
```

  - zazwyczaj ta sama nazwa, co klasa

- Uwaga: można zmienić klasę istniejącego obiektu
  - nigdy tego nie robić

Nowe funkcje generyczne

- Nowa funkcja generyczna wywołuje `UseMethod()`

```
f <- function(x){
    UseMethod("f")
}
```

- Metoda `f()` dla klasy `a`

```
f.a <- function(x){
    return("Class a")
}
a <- structure(list(), class = "a")
class(a)
f(a)
```

- Metoda `mean()` dla klasy `a`

```
mean.a <- function(x){
    return("a")
}
mean(a)
```

Wybór metody

- Funkcja `UseMethod()` twory wektor nazw funkcji `paste0("generic", ".", c(class(x), "default"))` i przegląda wektor w takiej kolejności

```
f <- function(x) UseMethod("f")
f.a <- function(x) "Class a"
f.default <- function(x) "Unknown class"

f(structure(list(), class = "a"))
f(structure(list(), class = c("b", "a")))
f(structure(list(), class = "c"))
```

- `f.a(structure(list(), class = "c"))` nie jest zalecane

Na typach podstawowych

- Dla danych typów podstawowych wykorzystuje się *klasa niejawna*

```
iclass <- function(x) {
  if (is.object(x)) {
    stop("x is not a primitive", call. = FALSE)
  }
  c(
    if (is.matrix(x)) "matrix",
    if (is.array(x) && !is.matrix(x)) "array",
    if (is.double(x)) "double",
    if (is.integer(x)) "integer",
    mode(x)
  )
}
iclass(matrix(1:5))
#> [1] "matrix"  "integer" "numeric"
```

S4

Wstęp

- Formalna definicja klas
  - opisanie danych i dziedziczenia
- Wybór metody na podstawie kilka argumentów
- Operator `@` dostępu do pól obiektów
- Paczka `methods`
  - może nie być dostępna w trybie wsadowym
    - `library(methods)`

Obiekty S4

- Czy `x` jest obiektem (funkcją generyczną) S4?
  - `str(x)`
  - `isS4(x)`
  - `pryr::otype(x)`
- Brak obiektów S4 w podstawowych bibliotekach

Deklaracja klas

- Funkcja `setClass()`
  - nazwa klasy (`UpperCamelCase`)
  - lista slotów (właściwości)
  - nazwa klasy rodzicielskiej
  - opcjonalnie: konstruktor

```
setClass("Person",
  slots = list(name = "character",
               age = "numeric"))
setClass("Employee",
  slots = list(boss = "Person"),
  contains = "Person")
```

Tworzenie obiektów

- Funkcja `new()`
  - jeżeli klasa posiada konstruktor, wykorzystać zamiast `new()`

```
alice <- new("Person", name = "Alice", age = 40)

john <- new("Employee", name = "John",
            age = 20, boss = alice)
```

Dostęp do slotów

- Operator `@` albo funkcja `slot()`
  - podobno do `$` oraz `[[` dla list

```
alice@age

slot(john, "boss")
```

Dziedziczenie po typach podstawowych

- Specjalny slot `.Data`

```
setClass("RangedNumeric",
 contains = "numeric",
 slots = list(min = "numeric", max = "numeric"))
rn <- new("RangedNumeric", 1:10,
 min = 1, max = 10)
rn@min
rn@.Data
```

- Po modyfikacji klasy trzeba ponownie stworzyć wszystkie obiekty tej klasy

Nowe funkcje generyczne

- `setGeneric(function)` tworzy nową funkcję generyczną
  - konwersja istniejącej funkcji `function` na generyczną
  - dla nowych funkcji koniecznie wywołać `standardGeneric()`

```
setGeneric("myGeneric", function(x) {
  standardGeneric("myGeneric")
})
```

    - odpowiednik `useMethod()`

Nowe metody

- `setMethod(function, classes, code)` rejestruje funkcję generyczną, dla argumentów odpowiadającym klasom z listy

```
setGeneric("union")
setMethod("union",
  c(x = "data.frame", y = "data.frame"),
  function(x, y) {
    unique(rbind(x, y))
  }
)
```

Wybór metody

- Metody należą do funkcji, przeszukiwanie klas zgodnie z hierarchią
- `callSuper()` wywołuje metodę rodzicielską

Klasy referencyjne

Wstęp

- Najnowszy system obiektowy
  - od roku 2010, wersja R 2.12
- Metody należą do obiektów
- Obiekty są modyfikowalne
- Zaimplementowane są w R na bazie obiektów S4

Tworzenie obiektów

- Funkcja `setRefClass(name)` oraz `new()`

```
Account <- setRefClass("Account")
Account$new()
```

Właściwości obiektów

- Argument `fields` funkcji `setRefClass(name)`
  - obiekt klasy `fields`

```
Account <- setRefClass("Account",
  fields = list(balance = "numeric"))

a <- Account$new(balance = 100)
a$balance

a$balance <- 200
a$balance
```

Zmienność obiektów

- RC obiekty są *referencjami*

```
b <- a
b$balance

a$balance <- 0
b$balance
```

- Kopiowanie obiektów

```
c <- a$copy()
c$balance

a$balance <- 100
c$balance
```

Metody obiektów

- należą do klas
  - ✦ mogą zmodyfikować obiekt

```
Account <- setRefClass("Account",
  fields = list(balance = "numeric"),
  methods = list(
    withdraw = function(x) {
      balance <<- balance - x
    },
    deposit = function(x) {
      balance <<- balance + x
    }
  )
)
a <- Account$new(balance = 100)
a$deposit(100)
a$balance
```

Dziedziczenie

- Argument `contains`

```
NoOverdraft <- setRefClass("NoOverdraft",
 contains = "Account",
 methods = list(
  withdraw = function(x) {
   if (balance < x) stop("Not enough money")
   balance <<- balance - x
  }
 )
)
accountJohn <- NoOverdraft$new(balance = 100)
accountJohn$deposit(50)
accountJohn$balance

accountJohn$withdraw(200)
```

Wybór metody

- Po jednym argumencie jak w S3
  - klasa `ANY` pasuje do wszystkich klas
  - `missing` pasuje do pominiętych argumentów
  - `callNextMethod()` wywołuje metodę rodzicielską
- Przy wyborze po wielu argumentach i wielokrotnym dziedziczeniu jest inaczej

Wybór modelu

S3

- Wystarczy w większości przypadków
  - proste obiekty
  - standardowe funkcje generyczne (`print`, `summary`, etc)
  - minimum kodu
  - większość klas to S3

S4

- Skomplikowane obiekty, dziedziczenie
- Dobre przykłady pakietów:
  - `Matrix`
  - Bioconductor

RC

- Podobny do innych języków programowania
- Możliwa zmiana obiektów przy wywołaniu funkcji

```
f(a, b)
```

  - ✦ unikać
    - ■ minimalizować

Case Study

Przykład

- Psycholożka Anna pracuje z anorektykami
- Co tydzień dla każdego pacjenta mierzy BMI
- Dane:
  - ciąg tygodni
  - ciąg pomiarów (trajektoria)
- Metody
  - wyświetlić trajektorie
    - `print()`
    - `plot()`
  - dane pominięte:
    - obliczyć ilość
      - uzupełnić

Deklaracja klasy


```
trajectory <- function(times, traj) {
    structure(list(times = times, traj = traj),
    class = "trajectory")
}
```

print


```
print.trajectory <- function(x){
    cat("* Times ="); print (x$times)
    cat("* Traj = \n"); print (x$traj)
}
```

plot


```
plot.trajectory <- function(x, ...){
   matplot(x$times,
           t(x$traj),
           xaxt="n",
           type="l",
           ylab= "",
           xlab="",
           pch=1, ...)
   axis(1,at=x$times)
 }
```

countmissing


```
countmissing <- function(x)
                  UseMethod("countmissing")

countmissing.trajectory <- function (x) {
    sum(is.na(x$traj))
}
```

Przykład


```
sekretyDiety <- trajectory(
    times=c(1,3,4,5),
    traj=rbind (
        c(15,15.1, 15.2, 15.2),
        c(16,15.9, 16,16.4),
        c(15.2, NA, 15.3, 15.3),
        c(15.7, 15.6, 15.8, 16)
    )
)
```