How do R love thee, OOP? Let me count the ways

Richie Morrisroe

November 15, 2016

Introduction

What is OOP?

Java/C++ Way

R Way

- S3 classes and methods
- S4 classes and methods

What is OOP?

- A way to manage state
- This is done by hiding the state in objects
- Which then communicate in specific ways (message passing)
- Alice sends Bob a poke message, which he then interprets
- Which stops random changes to one thing invading Poland

Java/C++ OOP

- Objects communicate through methods
- Objects have functions and data attached
- Only defined methods can update the internal state
- Internals of an object are hidden
- I know very little about this :(

The R Approach

- R uses generic functions rather than methods
- Each generic has methods for some subset of objects
- R has multiple, overlapping object systems for such
- Generic functions are at the heart of the language, such as print, plot, ggplot and summary

What is a generic function?

- Essentially everything that works throughout the language

```
length(methods("print"))
length(methods("plot"))
length(methods("["))
length(methods("$"))
length(methods("as.data.frame"))
```

```
[1] 237
[1] 33
[1] 40
[1] 11
[1] 38
```

S3 simply

- In your function, add a class attribute
- Write a generic
- There is no Step 3

```
foobar_func <- function(df) {
    class(df) <- c("data.frame", "foobar")
    df
}
```

S3 Generic

```
print.foobar <- function(data, ...) {
    print("foobar!")
  }
```

Testing our Generic

```
testdf <- data.frame(
    first=sample(letters,
                 size=26,
                 replace=TRUE),
    second=sample(1:1000, size=26, replace=TRUE))
foobar_df <- foobar_func(testdf)
print(foobar_df) %>% head()
```

w	443
q	400
p	48
p	415
y	943
x	599

- D'oh! This didn't work, because S3 dispatches on the first argument of the class vector.

Fixing our Generic

```
  foobar_func <- function(df) {
      class(df) <- c("foobar", "data.frame")
      df
  }
print(foobar_df)
```

- That's almost the entirety of S3

Creating a new generic

```
baz <- function (x, ...) {
   UseMethod("baz", x)
 }
baz.foobar <- function(x, ...) {
    print ("Worst method ever")
}
baz.default <- function(x, ...) {
    print("God, this is a boring example")
}
baz(foobar_df)
baz(testdf)
```

```
[1] "Worst method ever"
[1] "God, this is a boring example"
```

- A call to `UseMethod` is then made for the generic
- It first looks for `foobar`, then `data.frame` and then a method called `default`

S3 Advantages/Disadvantages

Advantages

- Simple
- Flexible
- Quick for simple methods (plot, print, summary etc)

Disadvantages

- No validation
- Limited extensibility (no multiple inheritance)
- S4 was introduced to rectify some of these problems

S4: The Sequel

- S4 operates similarly to S3, but has a much more structured way to create objects.
- Objects must satisfy certain predicates, or the create object functions fail
- This can essentially implement invariants across your R code
- With generic functions (pre-specified or new), simple DSL's can be created

A Digression: Stockfighter

- A (now defunct) start-up which focused on programming games
- The first game involved trading stocks on a fictional exchange
- You were given API client docs and a browser interface (that was pretty crap)
- I wrote a lot of code against this API
- I built a simple client (GitHub)
- And split all my level code and object system into another package (here)
- I'll be using my work on this as an example throughout

Overall Structure

- Stockfighter had `levels` which were associated with a number of things
- Each level had a `venue` or an exchange, and a set of `stocks/tickers` that might trade on them
- Some actions were to `make_order` for a stock or request a `quote` or the status of the `orderbook` or of an outstanding `order`.
- Orders could also be `cancelled`.
- First I built a basic API based on the docs (using `httr`)
- I then created a root object `trades`
- Which I used to create a set of generics useful for many other functions

Trades object

```
setClass(Class = "trades",
         slots = list(ok = "logical",
                      account = "character",
                      venues = "character",
                      tickers = "character",
                      timestamp="data.frame"),
         prototype = list(ok = NA,
                          account = NA_character_,
                          venues = NA_character_,
                          tickers = NA_character_,
                          timestamp = data.frame(
                              start = NA, end = NA)))
```

- This creates an object which all of the other objects inherit from

S4 Classes

- Must be created with a call to setClass
- Must specify a prototype object defining what the allowed values are
- These are ridiculously specific, such that `NA` is only acceptable for Boolean fields
- Slots: what the elements of the class are, and what type they take (ANY can be used to ensure that the class slot can hold anything)
- prototype: default values for the object
- validity: a function that returns TRUE if the object is a instance of the class
- contains: what other class the class inherits from (VIRTUAL creates a virtual class)

Defining some generics

```
account <- function(object) {
    object@account

}
setMethod("account", signature("trades"),
          def = account)

setGeneric("account", function(object) {
  standardGeneric("account")
})
```

- First we define the account function
- Then we set it to work with objects of class trades
- Then we register it as a generic function

More generics

```
venue <- function(object) {
    object@venues
}
ticker <- function(object) {
    object@tickers
}
```

- These are simple access-or functions, but they work on all relevant objects
- They help to clarify the code, rather than losing it in a sea of object@something$list
- Reduce the number of bugs caused from incorrectly grabbing the wrong part of the list

Inheritance

- Pretty easy

```
setClass("quote",
         slots=list(bid="integer",
                    ask="integer",
                    bidSize="integer",
                    askSize="integer",
                    bidDepth="integer",
                    askDepth="integer",
                    last="integer",
                    lastSize="integer",
                    lastTrade="character",
                    quoteTime="character"),
         contains="trades")
```

- Note that you need the tedious `NA` drill (in the prototype) from before if you want to allow for any missing values in any instance of the object

Simplifying Code

- There was a lot of setup and checks to perform for Stockfighter
- Monitoring the levels was useful (graphs later)
- S4 helped me to simplify a lot of code and avoid repetition

```
level <- start_level("sell_side")
while(isTRUE(levok)) {
    current_state <- state_of_market(level,
                                     apikey)
    level_stat <- get_level_status(current_state)
    status <- status(stat)
    levok <- ok(stat)
    if(status!="open") {
        break
    }
}
```

State of Market

```
state_of_market <- function(level, apikey) {
    account <- account(level)
    venue <- venue(level)
    stock <- ticker(level)
    quote <- as_quote(venue, stock)
    ord <- as_orderbook(venue, stock)
    myorders <- (as_orderlist(level, apikey))
    status <- (level_status(level, apikey=apikey))
    res <- list(orderbook=ord,
                quote=quote,
                myorders=myorders,
                status=status)
    res
}
```

- I promised myself here that I wouldn't get distracted by futures

as_orderbook

```
as_orderbook <- function(venue, stock) {
    res <- stockfighterr::get_orderbook(venue, stock)
    resp <- stockfighterr::parse_response(res)
    respo <- orderbook(resp)
}
```

Orderbook

```
orderbook <- function(order) {
        tsparsed <- lubridate::ymd_hms(order$ts)
        orderbook <- with(order,
                          new("orderbook",
                              ok=ok,
                              venues=venue,
                              tickers=symbol,
                              ymdhms=tsparsed,
                              bids=bids,
                              asks=asks,
                              timestamp=timestamp))
        orderbook
}
```

Layers and Layers, oh My!

- The flow goes as follows
- we get a http response from `get_orderbook`
- This gets parsed to a list
- Then converted to a orderbook object
- We can wrap the whole thing into `as_orderbook`
- Which then gets called in a loop to update our understanding of the market

Multiple Inheritance

- S3 always dispatches on the first element of the class attribute
- S4 can dispatch based on multiple different types

An Example: Timing of functions

- I realised soon that I couldn't rely upon the server timestamp
- I really didn't want to rewrite my code
- So I wrote a function to wrap my current functions

```
timed <- function(f, ...) {
    function(...) {
        start <- lubridate::now(tzone="UTC")
        res <- f(...)
        end <- lubridate::now(tzone="UTC")
        res <- list(time=data.frame(start=start, end=end),
                    res=res)
    }
}
```

the Problem

- Before I wrote this function, I had nice S4 objects
- Afterwards, I had horrible lists and indexing code again
- A solution?
- Multiple Inheritance!

Timed class

```
setClass(Class="Timed",
         slots=list(timestamp="data.frame",
                    res="trades"))
```

time, revised

```
timed <- function(f, ...) {
    function(...) {
        start <- lubridate::now(tzone="UTC")
        res <- f(...)
        end <- lubridate::now(tzone="UTC")
        timed <- new("Timed", timestamp=data.frame(start=sta
    }
}
```

Writing new generics

```
ticker.trades <- function(object) {
    object@tickers
}
setMethod("ticker", signature("trades"),
          def = ticker.trades)
ticker.Timed <- function(object) {
    obj <- object@res
    ticker(obj)
}
setMethod("ticker", signature("Timed"),
          def = ticker.Timed)
```

- This was surprisingly difficult to figure out
- Mind you, I had to recreate data in order to test (because the service has shut down)

Useful Generics

- You must write a data.frame method
- Otherwise you will spend all of your time converting to/from data.frame

```
method.skeleton("as.data.frame", signature="orderbook")
setMethod("as.data.frame",
    signature(x = "orderbook"),
    function (x, row.names = NULL, optional = FALSE, ...)
    {

    }
)
```

Data frame methods, continued

```
as.data.frame.orderbook <- function (x, row.names = NULL,
                                     optional = FALSE, ...)
{
    ordbids <- get_bids(x)
    ordasks <- get_asks(x)
    time <- x@ymdhms
    names <- c("time", names(ordbids))
    bidask <- rbind(ordbids, ordasks)
    times <- rep(time, nrow(bidask))
    bidasktime <- cbind(times, bidask)
    dfs <- as.data.frame(bidasktime)
    dfs
}
```

Comparison Methods

- You can define methods for addition, subtraction, etc with what is known as group generics
- I'm not covering them because they are horribly complex
- Common ones include `Arith`, `Compare` and `Ops`

```
setMethod("==",
    signature(e1 = "quote", e2 = "quote"),
    function (e1, e2)
    {
        ifelse(e1@bid == e2@bid &
                   e1@bidSize==e2@bidSize &
                   e1@askSize==e2@askSize &
                   e1@bidDepth==e2@bidDepth &
                   e1@askDepth==e2@askDepth &
                   e1@last==e2@last &
                   e1@lastSize==e2@lastSize &
                   e1@lastTrade==e2@lastTrade, TRUE, FALSE)
    }
)
```

R5

- Don't know much about this
- Appear to be implemented as a combination of an S4 class and an enviroment
- Have side effects (call by reference) semantics
- Accessed via list notation
- Similar to Java/C++ objects

`refclass$state`

Conclusions

- R has a rich heritage of OOP
- These are somewhat contradictory in nature (with confusingly named functions)
- S3 are simple but limited
- S4 are complicated and powerful (and much, much stricter)