If you use the same code three times, write a function.  
If you write three such related functions, set up a package.  
But if you write three embarrassingly similar functions…  
write code to generate their code for you?  
In this post, we’ll deal with source code generation.  
We’ll differentiate scaffolding from generating code, and we’ll present various strategies observed in the wild.

**Introduction**

**If you can repeat yourself, you’re lucky**

When would you need to generate code? A possible use case is wrapping a web API with many, many endpoints that have a predictable structure (parameters, output format) that’s well documented (“API specs”, “API schema”).

In any case, to be able to generate code, you’ll have some sort of underlying data/ontology.  
Having that data (specs of a web API, of an external tool you’re wrapping, structured list of all your ideas, etc.), and some consistency in the different items, is quite cool, lucky you!  
Some of us deal with less tidy web APIs. 

**Scope of this post**

In this post, we’ll look into *scaffolding* code (when your output is some sort of skeleton that’s still need some human action before being integrated in a package) and *generating* code (you hit a button and end up with more functions and docs in the package for its users to find).

**Scaffolding code**

Even without getting to the dream situation of code being cleanly generated, it can help your workflow to create function skeletons based on data.

* The idea was to have following commits edit functions enough to make them work without, as he said, starting from scratch.
* The experimental scaffolder package by Yuan Tang *“provides a comprehensive set of tools to automate the process of scaffolding interfaces to modules, classes, functions, and documentations written in other programming languages. As initial proof of concept, scaffolding R interfaces to Python packages is supported via reticulate.”*.  
  The scaffold\_py\_function\_wrapper() function takes a Python function as input and generates a R script skeleton (R code, and docs, both of them needing further editing).

In these two cases, what’s generated is a template for both R code and the corresponding roxygen2 docs.

**Generating code**

“odin works using code generation; the nice thing about this approach is that it never gets bored. So if the generated code has lots of tedious repetitive bits, they’re at least likely to be correct

Quite convincing, right? But when and how does one generate code for an R package?

**Generating code once or once in a while**

* For the package whose development prompted him to start the Twitter thread mentioned earlier, Miles McBain used code generation.  
  The package creates wrappers around dplyr functions, that can in particular automatically ungroup() your data.  
  Now say Miles decides to wrap a further dplyr function.

Code generating a function

build\_fn <- function(fn) {

fn\_name <- name(fn)

glue::glue("{fn\_name} <- function(...) {{\n",

" dplyr::ungroup(\n",

" {fn}(...)\n",

" )\n",

"}}\n")

}

Code generating docs

build\_fn\_doco <- function(fn) {

fn\_name <- name(fn)

glue::glue(

"##' Ungrouping wrapper for {fn\_name}\n",

"##'\n",

"##' The {PKGNAME} package provides a wrapper for {fn\_name} that always returns\n",

"##' ungrouped data. This avoids mistakes associated with forgetting to call ungroup().\n",

"##'\n",

"##' For original documentation see [{fn}()].\n",

"##'\n",

"##' Use [{fn\_name}...()] to retain groups as per `{fn}`, whilst\n",

"##' signalling this in your code.\n",

"##'\n",

"##' @title {fn\_name}\n",

"##' @param ... parameters for {fn}\n",

"##' @return an ungrouped dataframe\n",

"##' @author Miles McBain\n",

"##' @export\n",

"##' @seealso {fn}, {fn\_name}..."

)

}

Voilà, there’s an updated R/ folder, and after running devtools::document() an updated man/ folder and NAMESPACE, and it all works.  
You’ll have noticed the use of the glue package, that Alicia Schep also praised in her rstudio::conf talk, and that we’ve seen in many of the examples we’ve collected for this post.

**Code generator in a dedicated package**

All the examples from the previous subsections had some sort of build scripts living in their package repo.  
There’s no convention on what to call them and where to store them.  
Now, R developers like their code packaged in package form.  
Alicia Schep actually stores a package in the build/ folder of vlbuildr, vlmetabuilder, that creates vlbuildr anew from the Vegalite schema!  
That’s meta indeed!  
Fret not, the build/ folder also holds a script called build.R that unleashes the auto-magic.

**When to update the package?**

We haven’t seen any code generating workflow relying on a Makefile or on a hook to an external source, so we assume such packages are updated once in a while when their maintainer amends, or notices an amendment of, the underlying ontology.

**Generating code at install time**

In the previous cases of code generation, the R package source was similar to many R package sources out there.  
Now, we’ve also seen cases where the code is generated when installing the package.  
It means that the code generation has to be perfect, since there isn’t be any human edit between the code generation and the code use.  
Let’s dive into a few examples.

**Generating icon aliases in icon**

In icon, an R package by Mitchell O’Hara-Wild that allows easy insertion of icons from Font Awesome, Academicons and Ionicons into R Markdown, to insert an archive icon one can type icon::fa("archive") or icon::fa\_archive(), i.e. every possible icon has its own alias function which pairs well with autocompletion e.g. in RStudio when starting to type icon::fa\_.  
When typing ?icon::fa\_archive one gets a man page entitled “Font awesome alias”, the same for all aliases.

#' @evalRd paste("\\keyword{internal}", paste0('\\alias{fa\_', gsub('-', '\_', fa\_iconList), '}'), collapse = '\n')

#' @name fa-alias

#' @rdname fa-alias

#' @exportPattern ^fa\_

fa\_constructor <- function(...) fa(name = name, ...)

for (icon in fa\_iconList) {

formals(fa\_constructor)$name <- icon

assign(paste0("fa\_", gsub("-", "\_", icon)), fa\_constructor)

}

rm(fa\_constructor)

When *documenting* the package, the man page “fa-alias” is created.  
The @evalRd tag ensures aliases for all icons from fa\_iconList get an alias{}

The code generation allows an easy update to new Font Awesome versions, with a very compact source code.

**Generate C++ bindings with Rcpp::compileAttributes()**

Rcpp::compileAttributes() generates code (the bindings required to call C++ functions from R) after scanning a package source files. Find more information in the Rcpp vignette about attributes. You could call the function “whenever functions are added, removed, or have their signatures changed.” but the aforementioned vignette also states “if you are using either RStudio or devtoolsto build your package then the compileAttributes function is called automatically whenever your package is built”.

**Generating code on-the-fly**

One step further, one might generate code on-the-fly, i.e. as users run the package.

# Populate methods while the connection is being established.

protocol\_spec <- jsonlite::fromJSON(self$url("/json/protocol"), simplifyVector = FALSE)

self$protocol <- process\_protocol(protocol\_spec, self$.\_\_enclos\_env\_\_)

# self$protocol is a list of domains, each of which is a list of

# methods. Graft the entries from self$protocol onto self

list2env(self$protocol, self)

that are called when creating a chromote object.  
The process\_protocol() function converts the Chrome Devtools Protocol JSON to a list of functions.

**Conclusion**

In this post we explored different aspects of source code scaffolding and generation in R packages.  
We’ve mentioned examples of code scaffolding (gitea, scaffolder), of code generation by a script (wisegroup, eml.build, redux, xaringanthemer) or by a meta package (vlbuildr and vlmetabuildr) before package shipping, of code generation at install time (icon, civis, minicss, Rcpp::compileAttributes()) and of code generation at run time (chromote, stevedore).  
Many of these examples used some form of string manipulation, in base R or with glue, to either generate an R script and its roxygen2 docs **or** code using eval() and parse() (minicss).  
One of them doesn’t use any text representation, and as.function and call/as.call instead (stevedore).  
icon also doesn’t write R files.