**Introduction**

Months ago, I asked a question to the community: [how should I organize my R research projects?](https://ntguardian.wordpress.com/2018/08/02/how-should-i-organize-my-r-research-projects/) After writing that post, doing some reading, then putting a plan in practice, I now have my own answer.

First, some background. In the early months of 2016 I began a research project with my current Ph.D. advisor that involved extensive coding and spanned over at least two years. My code was poorly organized and thus problematic, as managing the chaos and extending the code became difficult. Meanwhile, I was reading articles by programmers and researchers about ways to organize R code so that research results are reproducible, distributable, and extensible. I identified two different approaches to organizing a project to meet these goals: one centered around makefiles, and another around package development. Given these competing approaches and their differing advantages, I was unsure what to do.

Since writing that post, I did more reading. First, I read two of Hadley Wickham’s (excellent) books: [*R Packages*](https://ntguardian.wordpress.com/2019/02/04/organizing-r-research-projects-cpat-case-study/r-pkgs.had.co.nz) and [*Advanced R*](https://ntguardian.wordpress.com/2019/02/04/organizing-r-research-projects-cpat-case-study/adv-r.had.co.nz). (I loved *R Packages* so much I [bought a physical copy](https://www.amazon.com/Packages-Organize-Test-Document-Share-ebook/dp/B00VAYCHL0).) I also read a book I picked up in a [Humble Bundle](https://www.humblebundle.com/) book sale called [*Code Craft; The Practice of Writing Excellent Code*](https://www.amazon.com/Code-Craft-Practice-Writing-Excellent-ebook/dp/B002MZAR6S/ref=sr_1_1?s=digital-text&ie=UTF8&qid=1548630399&sr=1-1&keywords=code+craft) by Pete Goodliffe for learning about good coding practices. Finally, I read a good portion of the [GNU make manual](https://www.gnu.org/software/make/manual/).

I also spent *months* restructuring the project to comply with what I learned. Many, *many* hours were spent just fixing the mess I had made by not doing things right in the first place.

The result is [**CPAT**](https://github.com/ntguardian/CPAT), an R package implementing some change point analysis statistical tests. What **CPAT** does will be the subject of a future post (it will be published when the accompanying paper is made available online); what I want to focus on in this article is how I learned to organize an R research project, and how that culminated in CPAT.

**Executable vs. Package: Why Not Both?**

In the earlier article I presented two approaches that I suggested were “competing” approaches to organizing a research project: [the *project as executable* approach of Jon Zelner](http://www.jonzelner.net/statistics/make/docker/reproducibility/2016/05/31/reproducibility-pt-1/) and [the *project as package* approach of Csefalvay](https://chrisvoncsefalvay.com/defensive-programming-r/) and [Flight](https://rmflight.github.io/posts/2014/07/analyses_as_packages.html). Both approaches, in my view, possessed unique advantages, but seemed to be at odds.

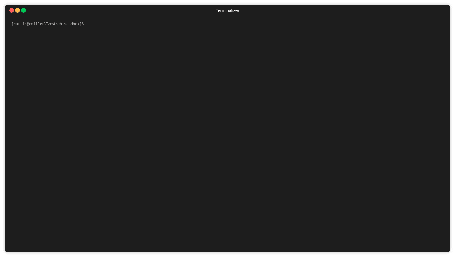
They are not at odds. **CPAT** demonstrates that it is possible to view an R project as both an executable and as a package. That said, the package development approach becomes dominant; making the package executable (from the command line) is an additional feature that makes the project even more portable and extensible.

If one is going to adopt the package development approach, one must use the hierarchy R packages needs. So that means:

* R code that defines the package (which are mostly just functions) is placed in the R/ directory.
* Documentation is placed in the man/ directory (if you’re using [**roxygen2**](https://cran.r-project.org/web/packages/roxygen2/vignettes/roxygen2.html) and [**devtools**](https://cran.r-project.org/web/packages/devtools/readme/README.html) like a sane human being, though, this is something you won’t do yourself, though).
* Project data goes in the data/ directory.
* Compiled code from other languages (such as C++ when using [**Rcpp**](http://www.rcpp.org/)) goes in the src/ directory.
* Code tests—*which are not optional and must be written!*—go in the tests/ directory (but if you’re using [**testthat**](http://testthat.r-lib.org/) for your testing then the tests you actually wrote go in tests/testthat/).
* Long-form documentation goes in vignettes/. This could be the paper itself, if written in the form of a vignette.
* Other important files should be placed in a reasonably-organized inst/ directory, to be installed with the package, along with other files that should be installed into the base directory (such as Makefile). For example, I put all my plots in inst/plots/, and this would also be a good directory to put the paper that accompanies the project.
* Put executable scripts, including R scripts, in exec/.

The approach championed by Zelner doesn’t require a particular organizational style but simply that there be a coherent organization to the project. R package development not only has a coherent structure but even *enforces* it. If that structure doesn’t quite work, then one can add other files and directories as needed and note them in the .Rbuildignore file, so they’re ignored when the package is built.

When writing an R package, the relevant R tools basically enforce some essential points of style such as documenting objects. Also, the developer-researcher starts to think of important functionality of the project in terms of reusable functions that should be added to the package to be called by the scripts that actually execute the analysis—with documentation and everything else. Having well-documented functions, even if they serve a minor purpose, helps greatly in making the project more easily understood and written not only by others but by the original author as well. In my case, since I wrote **CPAT** almost exclusively with [vim](https://www.vim.org/), I wrote a [UltiSnips](https://github.com/SirVer/ultisnips) snippet creating a function skeleton that not only defines the function but automatically adds the framework of the documentation, as seen below.



While package development does place (helpful) constraints, it does not specify everything. In other words, there is room for style. I essentially define *style* to mean any aspect of programming in which a choice is made that was not determined by the programming language or software. Examples of style include naming conventions, indentation, etc. Consistent style makes for understandable code; having consistent style is arguably more important than the stylistic decisions made. So I decided to codify my own stylistic preferences in a [style guide](https://github.com/ntguardian/CPAT/blob/master/STYLE.pdf), and when I did my code rewrite I made the new code comply with my style guide, even if that aded more time to editing. Whenever I encountered a new “decision point” (such as, say, dataset naming conventions), I committed my decision to the style guide.

As I mentioned above, the package development approach turns out not to be mutually exclusive with the project-as-executable approach. While it seems like documentation on R package development (including Dr. Wickham’s book) mentions the exec/ directory of a package only in passing, I found it to be a good place to place executable R scripts. Similarly, make can still be used to automate analysis tasks; R packages allow for including make files.

So in addition to the files that essentially defined the package, I also wrote stand-alone, command line executable R scripts and placed them in the exec/ directory (which causes them to be flagged as “executable” when the package is installed). I wrote a Vim template file for R scripts that provides a skeleton for making the package executable from the command line. That template is listed below:

#!/usr/bin/Rscript

################################################################################

# MyFile.R

################################################################################

# 2018-12-31 (last modified date)

# John Doe (author)

################################################################################

# This is a one-line description of the file.

################################################################################

# optparse: A package for handling command line arguments

if (!suppressPackageStartupMessages(require("optparse"))) {

install.packages("optparse")

require("optparse")

}

################################################################################

# MAIN FUNCTION DEFINITION

################################################################################

main <- function(foo, bar, help = FALSE) {

# This function will be executed when the script is called from the command

# line; the help parameter does nothing, but is needed for do.call() to work

quit()

}

################################################################################

# INTERFACE SETUP

################################################################################

if (sys.nframe() == 0) {

cl\_args <- parse\_args(OptionParser(

description = "This is a template for executable R scripts.",

option\_list = list(

make\_option(c("--foo", "-f"), type = "integer", default = 0,

help = "A command-line argument"),

make\_option(c("--bar", "-b"), type = "character",

help = "Another command-line argument")

)

))

do.call(main, cl\_args)

}

Converting my scripts into modularized, executable programs was, not surprisingly, very time consuming, and the transition was not perfect; some scripts just could not be modularized well. Nevertheless, the end result was likely worth it, and I could then write a [Makefile](https://github.com/ntguardian/CPAT/blob/master/Makefile) defining how the pieces fit together. This tamed the complexity of the project and made it more reproducible; someone looking to repeat my analysis should only have to type make in a Linux terminal[1](https://ntguardian.wordpress.com/2019/02/04/organizing-r-research-projects-cpat-case-study/#fn-3527-1) to see the results themselves.

While I did make my project modular and executable, though, I did not try to make it stand-alone with, say, [**packrat**](https://rstudio.github.io/packrat/) or [Docker](https://www.docker.com/). I did try to use **packrat**, even setting it up to work with my package. But I ran into severe problems when I tried to work with my package at the University of Utah Mathematics Department, since the computer system’s R installation is almost four years old as of this writing and highly tempermental due to how the system administrator set it up. **packrat** made complications working with the department computers even worse, and I disabled it in a huff one day and never looked back. As for Docker or [GitLab](https://about.gitlab.com/), I did not want my project tied up with proprietary or web-based services, and I felt that the end result Zelner was seeking when using these services is overkill; when you’ve added **packrat** (which I didn’t because of complications, but still) and defined how the project pieces fit together with make, you’ve mostly conquered the reproducibility problem, in my view. So I never missed these services.

The end result of this work can be seen in [the paper branch of CPAT](https://github.com/ntguardian/CPAT/tree/paper), also permanently available [in this tarball](http://www.math.utah.edu/~cmiller/software/CPAT/CPAT_0.0.0.9000_paper.tar.gz). The directory tree is also informative.

**In Practice**

In some sense the end goal is to have an R package that could be distributed to others via, say, [CRAN](https://cran.r-project.org/), so they can *use* the methods you employed and developed, not just reproduce your research; at least, that’s the case for me, a mathematical statistician interested in analyzing and developing statistical tests and procedures. When a package is written to contain research and not just for software distribution, it comes with a lot of files that aren’t needed for the package to function; just look at the dirctory tree!

The solution is to just delete the files that can be recreated—perhaps with make clean if you set it up right—and consider adding other files to .Rbuildignore when you want to distribute the package for others to use. So this isn’t actually a big problem.

Another issue that I encountered and am still unsure about are functions that are useful to the project but not useful outside of it. If you look through the [paper branch manual](https://github.com/ntguardian/CPAT/blob/paper/MANUAL.pdf) or even the [public version manual](https://github.com/ntguardian/CPAT/blob/master/MANUAL.pdf) you will find functions that were useful only for the project, perhaps for converting data structures created by scripts or making particular plots that make sense only for the paper. They’re all private functions that need to be accessed via the ::: operator, yet they’re still in the manual.

I’m undecided whether this is good style. On the one hand, it’s nice that when others read your code there’s manual entries even for functions that are local to the project to further document what was done and how the code works. Even when distributing the software, having every function documented, even ones that are “private” to the package, seems to be in concordance with the spirit of open source software, making the source code easier understood by users who need and want to know how your software works. It also could serve as a good way to modularize documentation; a statistical formula is kept with the function that computes it rather than the interface to that function (which likely links to that underlying function). Having examples for those internal functions also should provide an additional layer of testing and helps when others want to extend the package.

On the other hand… most of the pages of the manual are devoted to functions the user isn’t supposed to be calling directly in their work. Of all those functions, maybe five are functions the user is expected to use. Should all that documentation space be devoted to something the user doesn’t use?

While I’m not set in my opinion, I lean to having more documentation rather than less, even if most of it is for private functions. After all, it’s useful to me when I’m developing the project and package.

**Conclusion**

I feel like spending those months to make my project logical and reporducible was time well spent. Not only did I learn a lot in the process, I had a useful end product that is [now available on CRAN](https://cran.r-project.org/web/packages/CPAT/index.html). Additionally, this project is not over; my advisor and I are continuing to work on extending the results that lead to the creation of this package in the first place, which will call for more simulation experiments. Now that I’ve organized my work I now have a good base for continuing that work.

I hope that this article inspired others on how to organize their R research projects. Gauging from reactions to my previous article, I think this is an underappreciated topic, unfortunately. Having a plan for managing package complexity and organization goes a long way to keeping your work under control and helps others appreciate what you’ve done. It also can lead to your work having a greater impact since others can use it as well.

I got a lot of good feedback from my previous article. I look forward to hearing what the community has to say now. I’m always open to suggestion.

[Packt Publishing](https://www.packtpub.com/) published a book for me entitled [*Hands-On Data Analysis with NumPy and Pandas*](https://ntguardian.wordpress.com/books/hands-on-data-analysis-with-numpy-and-pandas/), a book based on my video course [*Unpacking NumPy and Pandas*](https://ntguardian.wordpress.com/video-courses/unpacking-numpy-and-pandas/). This book covers the basics of setting up a Python environment for data analysis with Anaconda, using Jupyter notebooks, and using NumPy and pandas. If you are starting out using Python for data analysis or know someone who is, please consider [buying my book](https://www.packtpub.com/big-data-and-business-intelligence/hands-data-analysis-numpy-and-pandas) or at least spreading the word about it. You can buy the book directly or purchase a [subscription to Mapt](https://mapt.packtpub.com/?gclid=Cj0KCQjwuuHdBRCvARIsAELQRQGrXDoR4ox216bUlSD6mTQYibN24CA5bYF7QikiQQ6j3cGm63HRXsEaAjJjEALw_wcB) and read it there.

If you like my blog and would like to support it, spread the word (if not get a copy yourself)!

1. Sadly my project is tied closely to the Unix/Linux setup; I have no idea how well it would work on Windows and I’m not very interested in making the project easy for Windows use (despite having Windows 10 installed on my primary laptop). What this means is that the goal of full reproducibility isn’t met for most Windows users, a large market of users. That said… if you’re a Windows user, just download [VirtualBox](https://www.virtualbox.org/) for free, download and install [Ubuntu](https://www.ubuntu.com/) or some other Linux distribution you like, install R and the needed packages, and now you can reproduce my work. You may even discover why I profer to work in a Linux environment yourself. [](https://ntguardian.wordpress.com/2019/02/04/organizing-r-research-projects-cpat-case-study/#fnref-3527-1)