Software Engineering Standards in the R Community

by Oliver Keyes, Jennifer Bryan, David Robinson

Abstract R flourishes because of its wide range of user-submitted packages, providing generalised solutions to real-world problems. Crucial to these packages being useful, adopted and trusted by useRs is that they follow reasonable software engineering expectations, from unit tests to user-friendly documentation. We performed a quantitative and qualitative analysis of CRAN-hosted packages, examining their use of unit tests, internal consistency and documentation standards, along with many other variables. We report on this analysis and suggest best practises for writing new R packages, along with proposals to improve the standards of existing, widely-used packages.

Introduction

Why Good Software is Important to R

Best Practices in Software Engineering

There is no single list of "best practices" for writing high quality software, but there are some general traits that such software possesses. McConnell points to many of them in *Code Complete* McConnell (2004) and divides them into "external" traits, that face the user, and "internal" traits, that face the developer.

From the perspective of the user, software should be accurate, fast-running and easy to use. From the perspective of the developer, the internal code should be maintainable, portable and lend itself to being tested. We can point to specific conventions or expectations that are built on these traits.

Traits of Good Software

1. **Unit and integration tests**: code that tests whether code run by the user is fit for use, by performing operations and checking that the results are as expected. This touches on both "accuracy" (it prevents the wrong result being provided to the user) and "maintainability" (it gives the developer a way of conveniently checking that modifications have not broken functionality before releasing software). In R package development, unit tests can be created in an *ad-hoc* fashion or using a pre-existing testing framework, such as *RUnit* Burger et al. (2015) or *testthat* Wickham (2011).
2. **Documentation**, demonstrating the use of software and illustrating particular pitfalls or components - this speaks directly to the ease of use that the software has. In R package development, such documentation usually consists of an entry for each exported object, and sometimes also includes long-form vignettes that describe the package as a whole. Both elements can be completed just using R itself, R Core Team (2012) but it has become increasingly popular to use package-provided functionality, aimed at reducing the barrier to documentation. For per-object documentation, this is usually the *roxygen2* package Wickham et al. (2015); for long-form vignettes, *knitr* Xie (2015).
3. **User-facing preditability**, allowing a user to (after a certain amount of experience with a piece of software) intuitively understand what unexplored functionality is likely to do if triggered, and understand upon the upgrade of a piece of software how much they will have to relearn. This is harder to automatically test for or build, but there are heuristics (explored below) for identifying whether software follows this standard.

We would also add one final trait not covered by McConnell, and perhaps specific to the R community's focus on a community of developers rather than any individual developer, and that is the ability of people to transition from being external users to being internal developers. In other words, whether the software is designed and built in such a way as to create a very low barrier to upstream bug reports and patches from individual software users.

Testing for Best Practices

Now that we have identified these "best practices" and "traits", how do we test for them in R packages?

Unit testing For unit testing, we can take the content of a package, and its "metadata" (the DESCRIPTION) file and use our knowledge of how the different frameworks (testthat and RUnit) make themselves known. In the case of testthat, a "tests" directory is created in the package source code, with a "testthat" folder underneath it. In the case of RUnit, tests can appear in a dedicated directory, or scattered throughout the package code, but must ultimately include either an explicit call to load the RUnit package, or an implicit call by using :: to refer to exported objects from RUnit's namespace. In both cases, the packages *may* be mentioned in the DESCRIPTION file, but this is not certain.

In the case of tests that do not use a package framework, there is no concrete way of automatically identifying if tests exist, but a general convention is to create a "tests" directory. Accordingly we adopted the following heuristics to identify the presence of tests, and what framework (if any) they followed:

1. If testthat is suggested in the DESCRIPTION file, the result is "testthat";
2. Otherwise, if RUnit is suggested in the DESCRIPTION file, the result is "RUnit";
3. Otherwise, if there is a tests directory, and a testthat directory underneath it, the result is "testthat";
4. Otherwise, if there is a call somewhere in the package's R code to RUnit, the result is "RUnit";
5. Otherwise, if there is a "tests" directory, the result is "Other";
6. Otherwise, the result is "None".

Documentation David, I'm going to let you take this section because you understand knitr-versus-sweave and all of that malarkey much better than muggins here.

Predictability The predictability of a package is, as said, not the easiest thing to evaluate, but we can tease out some information by looking at several characteristics. One obvious heuristic for user-facing predictability is to look for the presence of *semantic versioning*, a versioning system that distinguishes backwards-compatible bugfixes, backwards-compatible new features, and "breaking changes" that create an incompatability between versions. The presence of this versioning system, or something analogous, allows the user to trivially identify, on an update, whether modifications between versions necessitate actions or changes on their end, and whether what the package will do has changed.

Ease of transition

Practices in CRAN

Using the above heuristics, we retrieved (distinctly) the metadata and source code of each package on CRAN as of 03:00:01 on 27 April 2015. This came to 6,551 packages in total. Metadata was retrieved using the meta-cran service, and the source code by downloading each package's source from CRAN. Each package was then checked using each of the heuristics described in the section above (*Testing for Best Practices*): the resulting dataset can be found in the R package that accompanies this paper.

Some figures and analyses, like Figure 1.

Bibliography

M. Burger, K. Juenemann, and T. Koenig. *RUnit: R Unit Test Framework*, 2015. URL http://CRAN.R-project.org/package=RUnit. R package version 0.4.28. [p1]

S. McConnell. *Code Complete*. Microsoft Press, 2nd edition, 2004. ISBN 0735619670. [p1]

R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2012. URL http://www.R-project.org/. ISBN 3-900051-07-0. [p1]

H. Wickham. testthat: Get started with testing. *The R Journal*, 3:5–10, 2011. URL http://journal.r-project.org/archive/2011-1/RJournal_2011-1_Wickham.pdf. [p1]

H. Wickham, P. Danenberg, and M. Eugster. *roxygen2: In-Source Documentation for R*, 2015. URL http://CRAN.R-project.org/package=roxygen2. R package version 4.1.1. [p1]

Y. Xie. *knitr: A General-Purpose Package for Dynamic Report Generation in R*, 2015. URL http://CRAN.R-project.org/package=knitr. R package version 1.10.5. [p1]

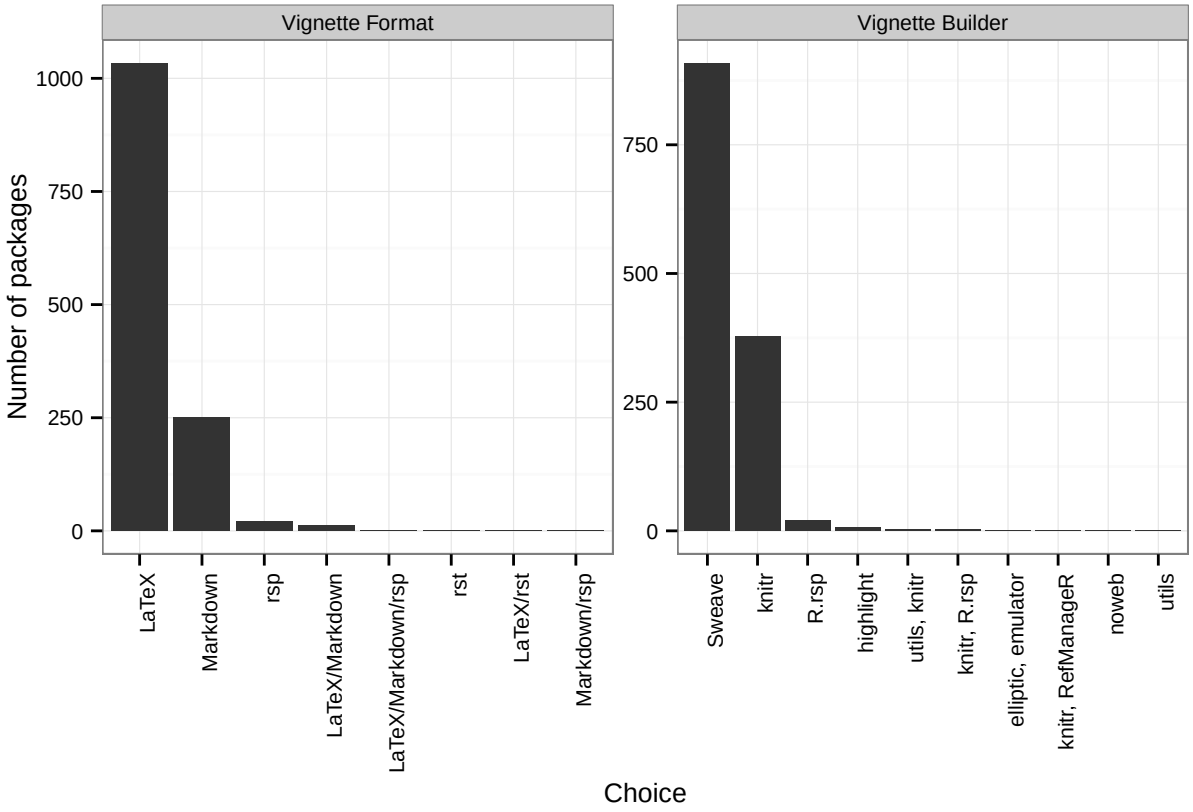

Figure 1: Distribution of the choice of vignette builder and format, among the 20.2% of CRAN packages that use vignettes.

Oliver Keyes
Wikimedia Foundation

ironholds@gmail.com

Jennifer Bryan
University of British Columbia

jenny@stat.ubc.ca

David Robinson
Princeton University

admiral.david@gmail.com