**Software engineering principles to improve quality and performance of R software**

Seth Russell1, Tellen D. Bennett1,2, Debashis Ghosh1,3

1 University of Colorado Data Science to Patient Value (D2V), Anschutz Medical Campus, Aurora, CO, United States of America

2 Pediatric Critical Care, University of Colorado School of Medicine, Aurora, CO, United States of America

3 Biostatistics and Informatics, Colorado School of Public Health, Aurora, CO, United States of America

Corresponding Author:

Seth Russell

CU Data Science to Patient Value

13199 E. Montview Blvd, Suite 210-15

Aurora, CO 80045 USA

Email address: seth.russell@ucdenver.edu

**ABSTRACT**

Today’s computational researchers are expected to be highly proficient in using software to solve a wide range of problems ranging from processing large datasets to developing personalized treatment strategies from a growing range of treatment options. While these researchers are well versed in their own field, formal training in software engineering principles and appropriate mentorship is often lacking. Two major themes that are not covered in most university coursework nor current literature are software testing and software optimization. Through a survey of all currently available The Comprehensive R Archive Network (CRAN) packages we show that reproducible and replicable software tests are frequently not available, and that many packages do not appear to employ software performance and optimization tools and techniques. Through use of examples from an existing R package, we demonstrate powerful testing and optimization techniques that can improve the quality of any researcher’s software.

**INTRODUCTION**

**Abstract Guidance – remove this box before submitting!**

* No more than approx. 500 words (or 3,000 characters).
* Self-contained and concisely describe the reason for the work, methodology, results, and conclusions. Uncommon abbreviations should be spelled out at first use. Do not include footnotes or references.
* Headings in structured abstracts should be bold and followed by a period. Each heading should begin a new paragraph. For example:

**Background.** The background section text goes here. Next line for new section.

**Methods.** Methods section here, then new line.

**Results.** Results section here, then new line.

**Discussion.** Discussion section here.

Writing scientific software has progressed from the work of early pioneers to a range of computer professionals, computational researchers, and self-taught individuals. The educational discipline of computer science, standardized many years ago through recommendations from the Association for Computing Machinery (ACM) (Atchison et al., 1968), has grown in breadth and depth over many years. Software engineering, an offshoot of computer science, is the present discipline that “seeks to develop and use systematic models and reliable techniques to produce high-quality software. These software engineering concerns extend from theory and principles to the development practices that are most visible to those outside the discipline.” (The Joint Task Force on Computing Curricula, 2015)

In a similar manner to the advances in computer science body of knowledge, computational researchers, statisticians, and similar professionals need to advance their skills by adopting principles of software engineering. Wilson et al in their 2014 paper covered key areas where scientists can benefit from software engineering best practices (Wilson et al., 2014). The term “best” as referenced in the previous cited work and others cited later refers to expert consensus based on knowledge and observational reporting of results from application of the practices. They provide a high-level description of 8 important principles of software engineering that should “reduce the number of errors in scientific software, make it easier to reuse, and save the authors of the software time and effort that can used for focusing on the underlying scientific questions.” While their principles are still relevant and important today, there has not been enough progress in this endeavor, especially with respect to software testing and software optimization principles (Wilson, 2016; Nolan & Padilla-Parra, 2017).

The ACM/IEEE recommendations for an undergraduate degree in software engineering describe a range of course work and learning objectives. Their guidelines call out 10 specific knowledge areas that should be part of or guide all software engineering coursework. The major areas are: computer science fundamentals, math and engineering fundamentals, professional interactions/communication, software modeling, requirement gathering, software design, verification, project processes, quality, and security (The Joint Task Force on Computing Curricula, 2015). These major themes are not covered extensively outside software engineering and include such generally applicable items such as software verification, validation, testing, and computer science fundamentals (e.g. software optimization, modeling, and requirement gathering).

In addition to the need for further training, understanding the software lifecycle is necessary: the process of software development from ideation to delivery of code. The largest component of a software’s lifecycle is maintenance. Software maintenance costs are large and increasing (Glass, 2001; Dehaghani & Hajrahimi, 2013; Koskinen, 2015); some put maintenance at 90% of total software cost. The chief factor in cost of maintenance with respect to software is time of the people creating and using the software. From the recent trend on making research results reproducible and replicable, some recommend making code openly available to any who might wish to repeat or further analyze results (Leek & Peng, 2015). With the development of any software artifact, an important consideration for implementation should be maintenance. As research scientists tend to think of their software products as unique tools that will not be used regularly or for a long period, they often do not consider long term maintenance issues during the development phase (Sandve et al., 2013; Prins et al., 2015). While a rigorous and formal software engineering approach is not well suited to the standard lifecycle of research software (Wilson, 2016), there are many techniques that can help to reduce cost of maintenance and speed development. While best practices such as the use of version control software, open access to data, software, and results are becoming more wide spread, other best practices such as testing and optimization need further attention.

In this paper, a brief survey of currently available R packages from The Comprehensive R Archive Network (CRAN) will be used to show the continued need for software testing and optimization. Source code for this analysis is freely available at <https://github.com/magic-lantern/SoftwareEngineeringPrinciples>. Additionally, the R package “pccc: Pediatric Complex Chronic Conditions” (Feinstein et al., 2018; DeWitt et al., 2017) (PCCC), available via CRAN and at <https://github.com/CUD2V/pccc>, is used for code examples in this article. PCCC is a combined R and C++ implementation of the Pediatric Complex Chronic Conditions software released as part of a series of research papers (Feudtner, Christakis & Connell, 2000; Feudtner et al., 2014). The PCCC package takes as input a data set containing International Statistical Classification of Diseases and Related Health Problems (ICD) Ninth revision or Tenth revision diagnosis and procedure codes and outputs which if any complex chronic conditions a patient has.

**ANALYSIS OF R PACKAGES ON CRAN**

**ANALYSIS OF TESTING OF R PACKAGES**

In order to get an estimate of the level of testing common among R software, an analysis of R packages available through CRAN has been performed. Although Nolan (Nolan & Padilla-Parra, 2017) has performed a similar analysis in the past, due to the rapid change in the CRAN as a whole, a reevaluation is necessary. At the time of Nolan’s work, CRAN contained 10084 packages, now it contains 13509. Furthermore, the analysis by Nolan had a few shortcomings that we have addressed in this analysis: there are additional testing frameworks for which we wanted to analyze their usage; not all testing frameworks and R packages store their test code in a directory named “tests”; only packages modified in the past 2 years was reported - there are many commonly used R packages that have not been updated in the last 2 years.

Although we address some shortcomings in analyzing R code for use of testing best practices, our choice of domain for analysis does have some limitations. Not all research software is written in R; for those that do use R, not all software development results in a package published on CRAN. While other software languages have tools for testing, additional research would be needed to evaluate level of testing in those languages to see how it compares to this analysis. Although care has been taken to identify standard testing use cases and practices for R, testing can be performed in-line through use of core functions such as stop() or stopifnot(). Also, developers may have their own test cases they run while developing their software, but did not include them in the package made available on CRAN. Unit tests can be considered as executable documentation, a key method of conveying how to use software correctly (Reese, 2018). Published research that involves software is not as easy to access and evaluate for use of testing code as CRAN packages are. While some journals have standardized means for storing and sharing code, many leave the storing and sharing of code up to the individual author, creating an environment where code analysis would require significant manual effort.

To analyze use of software testing techniques, we evaluated all CRAN packages on two different metrics:

Metric 1: In the source code of each package, search for non-empty testing directories using the regular expression pattern "[Tt]est[^/]\*/.+". All commonly used R testing packages (those identified for metric 2) recommend placing tests in a directory by themselves, which we look for.

Metric 2: Check for stated dependencies on one of the following testing packages: RUnit (Burger, Juenemann & Koenig, 2015), svUnit (Grosjean, 2014), testit (Xie, 2018), testthat (Wickham, 2011), unitizer (Gaslam, 2017), or unittest (Lentin & Hennessey, 2017). From the authors of these packages, it is recommended to list dependency (or dependencies) to a testing framework even though standard usage of a package may not require it.

For the testing analysis, we used 2008 as the cutoff year for visualizations due to the low number of packages last updated prior to 2008.

As shown in Figure 1, the evaluation for the presence of a non-empty testing directory shows that there is an increasing trend in testing R packages, with 44% of packages updated in 2018 having some tests. Table S1 contains the data used to generate Figure 1.

As shown in Figure 2, reliance upon testing frameworks is increasing over time both in count and as a percentage of all packages. There 16 packages that list dependencies to more than one testing framework (9 with dependencies on both RUnit and testthat, 7 with dependencies on both testit and testthat), so the total number of packages shown in the histogram includes 16 that are double counted. Table S2 contains the data used to generate Figure 2.

As the numbers from Metric 1 do not match the numbers of Metric 2, some additional exploration is necessary. There are 884 more packages identified from Metric 1 vs Metric 2. There are 1115 packages that do not list a dependency to a testing framework, but have a testing directory; e.g. the package xlsx (Dragulescu & Arendt, 2018). Some packages use a testing framework, but do not list it as a dependency; e.g. the package redcapAPI (Nutter & Lane, 2018). There are also 231 packages that list a testing framework as a dependency, but do not contain a directory with tests. See Supplemental tables S1 and S2 for more details.

**ANALYSIS OF OPTIMIZATION OF R PACKAGES**

In order to get an estimate of the level of software optimization common among R software, an analysis of R packages available through CRAN has been performed. To analyze the use of software optimization tools and techniques we evaluated all CRAN packages on two different metrics:

Metric 1: In the source code of each package, search for non-empty src directories using the regular expression pattern "src[^/]\*/.+". By convention, packages using compiled code (C, C++, Fortran) place those files in a '/src' directory.

Metric 2: Check for stated dependencies on packages that can optimize, scale performance, or evaluate performance of a package. Packages included in analysis are: DSL (Feinerer, Theussl & Buchta, 2015), Rcpp (Eddelbuettel & Balamuta, 2017), RcppParallel (Allaire et al., 2018), Rmpi (Yu, 2002), SparkR (Apache Software Foundation, 2018), batchtools (Bischl et al., 2015), bench (Hester, 2018), benchr (Klevtsov, Antonov & Upravitelev, 2018), doMC (Calaway, Analytics & Weston, 2017), doMPI (Weston, 2017), doParallel (Calaway et al., 2018), doSNOW (Calaway, Corporation & Weston, 2017), foreach (Calaway, Microsoft & Weston, 2017), future (Bengtsson, 2018), future.apply (Bengtsson & R Core Team, 2018), microbenchmark (Mersmann, 2018), parallel (R Core Team, 2018), parallelDist (Eckert, 2018), parallelMap (Bischl & Lang, 2015), partools (Matloff, 2016), profr (Wickham, 2014), profvis (Chang & Luraschi, 2018), rbenchmark (Kusnierczyk, Eddelbuettel & Hasselman, 2012), snow (Tierney et al., 2018), sparklyr (Luraschi et al., 2018), tictoc (Izrailev, 2014).

For the optimization analysis, we used 2008 as the cutoff year for visualizations showing presence of a src directory due to the low number of currently available packages last updated prior to 2008. For optimization related dependencies, in order to aid visual understanding, we used 2009 as the cutoff year and only showed those packages with 15 or greater dependent packages in a given year.

Automatically analyzing software for evidence of optimization has similar difficulties to those mentioned previously related to automatically detecting the use of software testing techniques and tools. The best evidence of software optimization would be in the history of commits, unit tests that time functionality, and package bug reports. While all R packages have source code available, not all have development history available nor unit tests available. Additionally, a stated dependency on one of the optimization packages listed could mean the package creators recommend using that along with their package, not that they are actually using it in their package. Despite these shortcomings, it is estimated that presence of a src directory or the use of specific packages is an indication that some optimization effort was put into a package.

As shown in Figure 3, the evaluation for the presence of a non-empty src directory shows that there is an increasing trend in using compiled code in R packages, by count. However, when evaluated as a percent of all R packages, the change has only been a slight increase over the last few years. Table S3 contains the data used to generate Figure 3.

As shown in Figure 4, in 2018, Rcpp is the most common optimization related dependency followed by parallel and foreach. Those same packages have been the most popular for packages last updated during the entire period shown. There 699 packages that list dependencies to more than one optimization framework (407 with 2 dependencies, 220 w/3, 53 w/4, 16 w/5, 2 w/6, 1 w/7), so the total number of packages shown in the histogram includes some that are double-counted. Table S4 contains the data used to generate Figure 4.

As the numbers from Metric 1 do not match the numbers of Metric 2, some additional exploration is necessary. In terms of total difference, there are 818 more packages using compiled code vs those with one of the searched for dependencies. There are 1726 packages that do not list a dependency to one of the specified packages, but have a src directory for compiled code. There are 908 packages that list a dependency to one of the specified packages but do not have a src directory. See Supplemental tables S3 and S4 for more details.

**CONCLUSION**

Researchers frequently develop software to automate tasks and speed the pace of research. Unfortunately, researchers are rarely trained in software engineering principles necessary to develop robust, validated, and performant software. Software maintenance is an often overlooked and underestimated aspect in the lifecycle of any software product. Software engineering principles and tooling place special focus on the processes around designing, building, and maintaining software. In this paper, the key topics of software testing and software optimization have been discussed along with some analysis of existing software packages in the R language. Our analysis showed that the majority of R packages have neither unit testing nor evidence of optimization available with normally distributed source code. Through self-education on unit testing and optimization, any computational or other researcher can pick up the key principles of software engineering that will enable them to spend less time troubleshooting software and more time doing the research they enjoy.

**REFERENCES**

Agruss C, Johnson B. 2000. Ad Hoc Software Testing. *Viitattu* 4:2009.

Allaire JJ, Francois R, Ushey K, Vandenbrouck G, library MG (TinyThread, http://tinythreadpp.bitsnbites.eu/), RStudio, library I (Intel T, https://www.threadingbuildingblocks.org/), Microsoft. 2018. *RcppParallel: Parallel Programming Tools for “Rcpp.”*

Apache Software Foundation. 2018.SparkR (R on Spark) - Spark 2.3.2 Documentation. *Available at* *https://spark.apache.org/docs/latest/sparkr.html*

Atchison WF, Conte SD, Hamblen JW, Hull TE, Keenan TA, Kehl WB, McCluskey EJ, Navarro SO, Rheinboldt WC, Schweppe EJ, Viavant W, Young DM Jr. 1968. Curriculum 68: Recommendations for Academic Programs in Computer Science: A Report of the ACM Curriculum Committee on Computer Science. *Commun. ACM* 11:151–197. DOI: 10.1145/362929.362976.

Beck K, Gamma E. 1998. Test Infected: Programmers Love Writing Tests. *Java Report* 3.

Bengtsson H. 2018. *future: Unified Parallel and Distributed Processing in R for Everyone*.

Bengtsson H, R Core Team. 2018. *future.apply: Apply Function to Elements in Parallel using Futures*.

Bischl B, Lang M. 2015. *parallelMap: Unified Interface to Parallelization Back-Ends*.

Bischl B, Lang M, Mersmann O, Rahnenführer J, Weihs C. 2015. BatchJobs and BatchExperiments: Abstraction Mechanisms for Using R in Batch Environments. *Journal of Statistical Software* 64:1–25.

Burger M, Juenemann K, Koenig T. 2015. *RUnit: R Unit Test Framework*.

Calaway R, Analytics R, Weston S. 2017. *doMC: Foreach Parallel Adaptor for “parallel.”*

Calaway R, Corporation M, Weston S. 2017. *doSNOW: Foreach Parallel Adaptor for the “snow” Package*.

Calaway R, Corporation M, Weston S, Tenenbaum D. 2018. *doParallel: Foreach Parallel Adaptor for the “parallel” Package*.

Calaway R, Microsoft, Weston S. 2017. *foreach: Provides Foreach Looping Construct for R*.

Chang W, Luraschi J. 2018. *profvis: Interactive Visualizations for Profiling R Code*.

Dehaghani SMH, Hajrahimi N. 2013. Which Factors Affect Software Projects Maintenance Cost More? *Acta Informatica Medica* 21:63–66. DOI: 10.5455/AIM.2012.21.63-66.

DeWitt P, Bennett T, Feinstein J, Russell S. 2017. *pccc: Pediatric Complex Chronic Conditions*.

Dragulescu AA, Arendt C. 2018. *xlsx: Read, Write, Format Excel 2007 and Excel 97/2000/XP/2003 Files*.

Eckert A. 2018. *parallelDist: Parallel Distance Matrix Computation using Multiple Threads*.

Eddelbuettel D, Balamuta JJ. 2017. *Extending R with C++: A Brief Introduction to Rcpp*. PeerJ Inc. DOI: 10.7287/peerj.preprints.3188v1.

Feinerer I, Theussl S, Buchta C. 2015. *DSL: Distributed Storage and List*.

Feinstein JA, Russell S, DeWitt PE, Feudtner C, Dai D, Bennett TD. 2018. R Package for Pediatric Complex Chronic Condition Classification. *JAMA Pediatrics*. DOI: 10.1001/jamapediatrics.2018.0256.

Feudtner C, Christakis DA, Connell FA. 2000. Pediatric Deaths Attributable to Complex Chronic Conditions: A Population-Based Study of Washington State, 1980–1997. *Pediatrics* 106:205–209.

Feudtner C, Feinstein JA, Zhong W, Hall M, Dai D. 2014. Pediatric complex chronic conditions classification system version 2: updated for ICD-10 and complex medical technology dependence and transplantation. *BMC Pediatrics* 14:199. DOI: 10.1186/1471-2431-14-199.

Gaslam B. 2017. *unitizer: Interactive R Unit Tests*.

Glass RL. 2001. Frequently Forgotten Fundamental Facts About Software Engineering. *IEEE Softw.* 18:112–111. DOI: 10.1109/MS.2001.922739.

Grosjean P. 2014. *SciViews-R: A GUI API for R*. MONS, Belgium: UMONS.

Hester J. 2018. *bench: High Precision Timing of R Expressions*.

Izrailev S. 2014. *tictoc: Functions for timing R scripts, as well as implementations of Stack and List structures.*

Klevtsov A, Antonov A, Upravitelev P. 2018. *benchr: High Precise Measurement of R Expressions Execution Time*.

Koskinen J. 2015.Software Maintenance Costs. *Available at* *https://wiki.uef.fi/download/attachments/38669960/SMCOSTS.pdf*

Kusnierczyk W, Eddelbuettel D, Hasselman B. 2012. *rbenchmark: Benchmarking routine for R*.

Leek JT, Peng RD. 2015. Opinion: Reproducible research can still be wrong: Adopting a prevention approach. *Proceedings of the National Academy of Sciences* 112:1645–1646. DOI: 10.1073/pnas.1421412111.

Lentin J, Hennessey A. 2017. *unittest: TAP-Compliant Unit Testing*.

Luraschi J, Kuo K, Ushey K, Allaire JJ, Macedo S, RStudio, Foundation TAS. 2018. *sparklyr: R Interface to Apache Spark*.

Matloff N. 2016. Software Alchemy: Turning Complex Statistical Computations into Embarrassingly-Parallel Ones. *Journal of Statistical Software* 71:1–15. DOI: 10.18637/jss.v071.i04.

Mersmann O. 2018. *microbenchmark: Accurate Timing Functions*.

Nolan R, Padilla-Parra S. 2017. exampletestr—An easy start to unit testing R packages. *Wellcome Open Research* 2. DOI: 10.12688/wellcomeopenres.11635.2.

Nutter B, Lane S. 2018. *redcapAPI: Accessing data from REDCap projects using the API*. DOI: 10.5281/zenodo.11826.

Prins P, de Ligt J, Tarasov A, Jansen RC, Cuppen E, Bourne PE. 2015. Toward effective software solutions for big biology. *Nature Biotechnology* 33:686–687. DOI: 10.1038/nbt.3240.

R Core Team. 2018. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing.

Reese J. 2018.Best practices for writing unit tests. *Available at* *https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices*

Sandve GK, Nekrutenko A, Taylor J, Hovig E. 2013. Ten Simple Rules for Reproducible Computational Research. *PLOS Computational Biology* 9:e1003285. DOI: 10.1371/journal.pcbi.1003285.

Sommerville I. 2015. *Software Engineering*. Boston: Pearson.

The Joint Task Force on Computing Curricula. 2015. *Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*. New York, NY, USA: ACM.

Tierney L, Rossini AJ, Li N, Sevcikova H. 2018. *snow: Simple Network of Workstations*.

Weston S. 2017. *doMPI: Foreach Parallel Adaptor for the Rmpi Package*.

Wickham H. 2011. testthat: Get Started with Testing. *The R Journal* 3:5–10.

Wickham H. 2014. *profr: An alternative display for profiling information*.

Wilson G. 2016. Software Carpentry: lessons learned. *F1000Research*. DOI: 10.12688/f1000research.3-62.v2.

Wilson G, Aruliah DA, Brown CT, Hong NPC, Davis M, Guy RT, Haddock SHD, Huff KD, Mitchell IM, Plumbley MD, Waugh B, White EP, Wilson P. 2014. Best Practices for Scientific Computing. *PLOS Biology* 12:e1001745. DOI: 10.1371/journal.pbio.1001745.

Xie Y. 2018. *testit: A Simple Package for Testing R Packages*.

Yu H. 2002. Rmpi: Parallel Statistical Computing in R. *R News* 2:10–14.