

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

Seth Russell<sup>1</sup>, Tellen D. Bennett<sup>1,2</sup> and Debashis Ghosh<sup>1,3</sup>

<sup>1</sup> University of Colorado Data Science to Patient Value, University of Colorado Anschutz Medical Campus, Aurora, CO, USA

<sup>2</sup> Pediatric Critical Care, University of Colorado School of Medicine, Aurora, CO, USA

<sup>3</sup> Department of Biostatistics and Informatics, Colorado School of Public Health, Aurora, CO, USA

## 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 options. Researchers are well versed in their own field, but may lack formal training and appropriate mentorship in software engineering principles. Two major themes not covered in most university coursework nor current literature are software testing and software optimization. Through a survey of all currently available Comprehensive R Archive Network 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.

**Subjects** Computer Education, Data Science, Software Engineering

**Keywords** Unit testing, Profiling, Optimization, Software engineering, R language, Statistical computing, Case study, Reproducible research, Data science

## INTRODUCTION

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, a discipline within computer science, "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 development practices that are most visible to those outside the discipline" (*The Joint Task Force on Computing Curricula, 2015*).

As they gain sophistication, computational researchers, statisticians, and similar professionals need to advance their skills by adopting principles of software engineering. *Wilson et al. (2014)* identified eight key areas where scientists can benefit from software engineering best practices. 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 eight important principles of software engineering that should

Submitted 12 October 2018

Accepted 11 January 2019

Published 4 February 2019

Corresponding author

Seth Russell,

seth.russell@ucdenver.edu

Academic editor

Sebastian Ventura

Additional Information and  
Declarations can be found on  
page 21

DOI 10.7717/peerj-cs.175

© Copyright

2019 Russell et al.

Distributed under

Creative Commons CC-BY 4.0

**OPEN ACCESS**

“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 be 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/Institute of Electrical and Electronics Engineers recommendations for an undergraduate degree in software engineering describe a range of coursework 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 (for example, 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 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 software maintenance cost is the 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 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>. After the presentation of the current state of R packages, general advice on software testing and optimization will be presented. 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](#)). pccc 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

### Testing of R packages

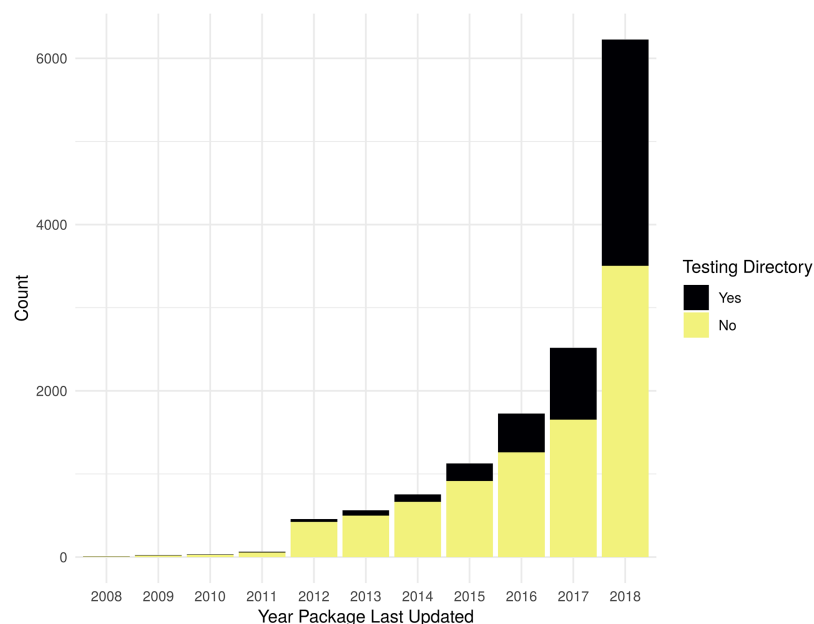
In order to estimate the level of testing common among R software, we analyzed all R packages available through CRAN. Although [Nolan & Padilla-Parra \(2017\)](#) performed a similar analysis in the past, due to the rapid change in the CRAN as a whole, a reevaluation is necessary. At the time of Nolan's work, CRAN contained 10,084 packages; it now contains 13,509. 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 were 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 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



**Figure 1 Packages with non-empty testing directory.** Count of packages with files in standard testing directories by year a package was last updated. Testing directory “Yes” is determined by the presence of files matching the regular expression “[Tt]est[?]\*/.+”; if no matches are found for an R package, it is counted as a “No”. [Full-size DOI: 10.7717/peerj-cs.175/fig-1](https://doi.org/10.7717/peerj-cs.175/fig-1)

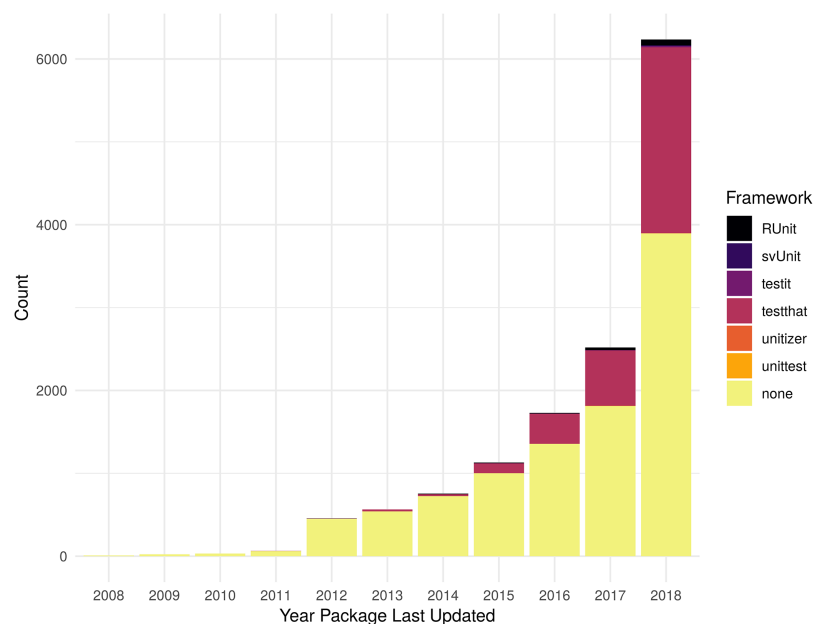
(Lentin & Hennessey, 2017). From the authors of these packages, it is recommended to list dependency (or dependencies) to a testing framework even though standard usage of a package may not require it.

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 Fig. 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 Fig. 1.

As shown in Fig. 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 on more than one testing framework (nine with dependencies on both RUnit and testthat, seven 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 Fig. 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 1,115 packages that do not list a dependency to a testing framework, but have a testing directory; for example, the package xlsx (Dragulescu & Arendt, 2018). Some packages use a testing framework, but do not list it as a dependency; for example, 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 Tables S1 and S2 for more details.



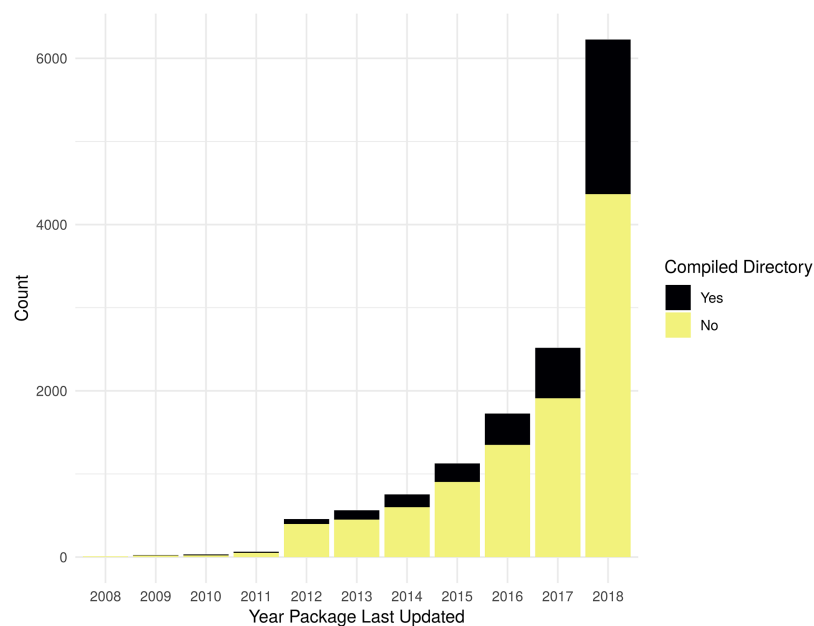
**Figure 2** Packages with testing framework dependency. Count of dependencies on a testing package (RUnit, svUnit, testit, testthat, unitizer, unittest) by year a package was last updated. Packages with no stated dependency from their DESCRIPTION file for one of the specified packages are listed as “none”. Full-size DOI: 10.7717/peerj-cs.175/fig-2

## Optimization of R packages

In order to estimate the level of software optimization common among R software, we performed an analysis of all R packages available through CRAN. 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., 2018a*), 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, 2014a*), profvis (*Chang & Luraschi, 2018*), rbenchmark (*Kusnierczyk, Eddelbuettel & Hasselman, 2012*), snow (*Tierney et al., 2018*), sparklyr (*Luraschi et al., 2018*), tictoc (*Izrailev, 2014*).



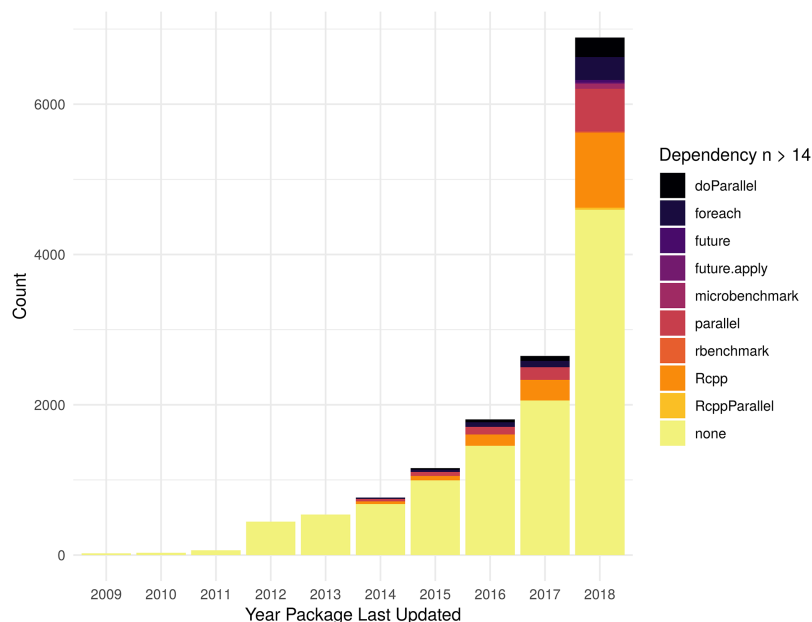
**Figure 3 Packages with non-empty src directory.** Count of packages with files in standard source directories that has code to be compiled by year a package was last updated. Compiled directory “Yes” is determined by the presence of files matching the regular expression “src[?]\*/.+”; if no matches are found for an R package, it is counted as a “No”. [Full-size !\[\]\(1679558f37f6db0dd8360a2a7e913e90\_img.jpg\) DOI: 10.7717/peerj-cs.175/fig-3](https://doi.org/10.7717/peerj-cs.175/fig-3)

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 Fig. 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 Fig. 3.

As shown in Fig. 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



**Figure 4 Packages with optimization framework dependency.** Count of dependencies on an optimization related package, see “Optimization of R packages” section for complete list, by year a package was last updated. Packages with no stated dependency from their DESCRIPTION file for one of the specified packages are listed as “none.” In order to aid visual understanding of top dependencies, we limited display to those packages that had 14 or more dependent packages. [Full-size](#) DOI: [10.7717/peerj-cs.175/fig-4](https://doi.org/10.7717/peerj-cs.175/fig-4)

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 [Fig. 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 1,726 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 [Tables S3](#) and [S4](#) for more details.

## RECOMMENDATIONS TO IMPROVE QUALITY AND PERFORMANCE

### Software testing

Whenever software is written as part of a research project, careful consideration should be given to how to verify that the software performs the desired functionality and produces the desired output. As with bench science, software can often have unexpected and unintended results due to minor or even major problems during the implementation process. Software testing is a well-established component of any software development lifecycle ([Atchison et al., 1968](#)) and should also be a key component of research software. As shown previously, even among R software packages intended to be shared

with and used by others, the majority of R packages (67–73% depending on metric) do not have tests that are made available with the package.

Various methodologies and strategies exist for software testing and validation as well as how to integrate software with a software development lifecycle. Some common testing strategies are no strategy, ad hoc testing (*Agruss & Johnson, 2000*), test driven development (TDD) (*Beck & Gamma, 1998*). There are also common project methodologies where testing fits into the project lifecycle; two common examples are the waterfall project management methodology, where testing is a major phase that occurs at a specific point in time, and the agile project management methodology (*Beck et al., 2001*), where there are many small iterations including testing. While a full discussion of various methods and strategies is beyond the scope of this article, three key concepts presented are: when to start testing, what to test, and how to test.

#### **Key recommendations for when to test:**

- Build tests before implementation.
- Test after functionality has been implemented.

**Discussion:** One of the popular movements in recent years has been to develop tests first and then implement code to meet desired functionality, a strategy called TDD. While the TDD strategy has done much to improve the focus of the software engineering world on testing, some have found that it does not work with all development styles (*Hansson, 2014; Sommerville, 2016*), and others have reported that it does not increase developer productivity, reduce overall testing effort, nor improve code quality in comparison to other testing methodologies (*Fucci et al., 2016*). An approach that more closely matches the theoretically based software development cycle and flexible nature of research software is to create tests after a requirement or feature has been implemented (*Osborne et al., 2014; Kanewala & Bieman, 2014*). As developing comprehensive tests of software functionality can be a large burden to accrue at a single point in time, a more pragmatic approach is to alternate between developing new functionality and designing tests to validate new functionality. Similar to the agile software development strategy, a build/test cycle can allow for quick cycles of validated functionality that help to provide input into additional phases of the software lifecycle.

#### **Key recommendations for what to test:**

- Identify the most important or unique feature(s) of software being implemented. Software bugs are found to follow a Pareto or Zipfian distribution.
- Test data and software configuration.
- If performance is a key feature, build tests to evaluate performance.

**Discussion:** In an ideal world, any software developed would be accompanied by 100% test coverage validating all lines of code, all aspects of functionality, all input, and all interaction with other software. However, due to pressures of research, having time to build a perfect test suite is not realistic. A parsimonious application of the Pareto principle



will go a long way towards improving overall software quality without adding to the testing burden. Large companies such as Microsoft have applied traditional scientific methods to the study of bugs and found that the Pareto principle matches reality: 20% of bugs cause 80% of problems; additionally a Zipfian distribution may apply as well: 1% of bugs cause 50% of all problems (Rooney, 2002).

To apply the Pareto principle to testing, spend some time in a thought experiment to determine answers to questions such as: What is the most important feature(s) of this software? If this software breaks, what is the most likely bad outcome? For computationally intensive components—how long should this take to run?

Once answers to these questions are known, the developer(s) should spend time designing tests to validate key features, avoiding major negatives, and ensuring software performs adequately. Optimization and performance recommendations are covered in the “Software Optimization” section. Part of the test design process should include how to “test” more than just the code. Some specific aspects of non-code tests include validation of approach and implementation choices with a mentor or colleague.

As a brief example of how to apply the aforementioned testing principles, we provide some information on testing steps followed during the pccc package development process. The first tests written were those that were manually developed and manually run as development progressed. Key test cases of this form are ideal candidates for inclusion in automated testing. The first tests were taking a known data set, running our process to identify how many of the input rows had complex chronic conditions, and then report on the total percentages found; this result was then compared with published values.

```
# read in HCUP KID 2009 Database
kid9cols <- read_csv("KID09_core_columns.csv")
kid9core <- read_fwf("KID_2009_Core.ASC",
  fwf_positions(
    start = kid9cols$start,
    end = kid9cols$end,
    col_names = tolower(kid9cols$name)),
  col_types = paste(rep("c", nrow(kid9cols)),
    collapse = ""))

# Output some summary information for manual inspection
table(kid9core$year)
dim(kid9core)
n_distinct(kid9core$recnum)

# Run process to identify complex chronic conditions
kid_ccc <-
  ccc(kid9core[, c(2, 24:48, 74:77, 106:120)],
  id = recnum,
  dx_cols = vars(starts_with("dx"), starts_with("ecode"))),
```

```
pc_cols = vars(starts_with("pr")),
icdv = 09)

# Output results for manual inspection
kid_ccc

# Create summary statistics to compare to published values
dplyr::summarize_at(kid_ccc, vars(-recnum), sum)
  %>% print.data.frame
dplyr::summarize_at(kid_ccc, vars(-recnum), mean)
  %>% print.data.frame
```

For the pccc package there is a large set of ICD codes and code set patterns that are used to determine if an input record meets any complex chronic condition criteria. To validate the correct functioning of the software, we needed to validate the ICD code groupings were correct and were mutually exclusive (as appropriate). As pccc is a re-implementation of existing SAS and Stata code, we needed to validate that the codes from the previously developed and published software applications were identical and were performing as expected. Through a combination of manual review and automated comparison codes were checked to see if duplicates and overlaps existed. Any software dealing with input validation or having a large amount of built-in values used for key functionality should follow a similar data validation process.

As an example of configuration testing, here is a brief snippet of some of the code used to automatically find duplicates and codes that were already included as part of another code:

```
icds <- input.file("../pccc_validation/icd10_codes_r.txt")

unlist(lapply(icds, function(i) {
  tmp <- icds[icds != i]
  output <- tmp[grepl(paste0("'", i, "'.*"), tmp)]
  # add the matched element into the output
  if(length(output) != 0)
    output <- c(i, output)
  output
}))
```

#### Key recommendations for how to test:

- Software developer develops unit tests.
- Intended user of software should perform validation/acceptance tests.
- Run all tests regularly.
- Review key algorithms with domain experts.

**Discussion:** Most programming languages have a multitude of testing tools and frameworks available for assisting developers with the process of testing software. Due to the recurring patterns common across programming languages most languages have a

SUnit ([Wikipedia contributors, 2017a](#)) derived testing tool, commonly referred to as an “xUnit” ([Wikipedia contributors, 2017b](#)) testing framework that focuses on validating individual units of code along with necessary input and output meet desired requirements. Based on software language used, unit tests may be at the class or function/procedure level. Some common xUnit style packages in R are RUnit and testthat. Unit tests should be automated and run regularly to ensure errors are caught and addressed quickly. For R, it is easy to integrate unit tests into the package build process, but other approaches such as post-commit hook in a version control system are also common.

In addition to unit tests, typically written by the developers of the software, users should perform acceptance tests, or high-level functionality tests that validate the software meets requirements. Due to the high-level nature and subjective focus of acceptance tests, they are often manually performed and may not follow a regimented series of steps. Careful documentation of how a user will actually use software, referred to as user stories, are translated into step by step tests that a human follows to validate the software works as expected. A few examples of acceptance testing tools that primarily focus GUI aspects of software are: Selenium ([Selenium Contributors, 2018](#)), Microfocus Unified Functional Testing (formerly known as HP’s QuickTest Professional) ([Micro Focus, 2018](#)), and Ranorex ([Ranorex GmbH, 2018](#)). As research focused software often does not have a GUI, one aide to manual testing processes is for developers of the software or expert users to create a full step by step example via an R Markdown ([Allaire et al., 2018b](#); [Xie, Allaire & Grolemund, 2018](#)) notebook demonstrating use of the software followed by either manually or automatic validation that the expected end result is correct.

In addition to the tool-based approaches already mentioned, other harder to test items such as algorithms and solution approach should be scrutinized as well. While automated tests can validate mathematical operations or other logic steps are correct, they cannot verify that the approach or assumptions implied through software operations are correct. This level of testing can be done through code review and design review sessions with others who have knowledge of the domain or a related domain.

During development of the pccc package, after the initial tests shown in previous sections, further thought went into how the specifics of the desired functionality should perform. Unit tests were developed to validate core functionality. We also spent time thinking about how the software might behave if the input data was incorrect or if parameters were not specified correctly. If an issue is discovered at this point, a common pattern is to create a test case for discovered bugs that are fixed—this ensures that a re-occurrence, known as a “regression” to software engineers, of this error does not happen again. In the case of pccc, developers expected large input comprised of many observations with many variables. When a tester accidentally just passed 1 observation with many variables, the program crashed. The problem was discovered to be due to the flexible nature of the `sapply()` function returning different data types based on input.

```

The original code from ccc.R:
# check if call didn't specify specific diagnosis columns
if (!missing(dx_cols)) {
  # assume columns are referenced by 'dx_cols'
  dxmat <- sapply(dplyr::select(
    data, !!dplyr::enquo(dx_cols)), as.character)
  # create empty matrix if necessary
  if(! is.matrix(dxmat)) {
    dxmat <- as.matrix(dxmat)
  }
} else {
  dxmat <- matrix("", nrow = nrow(data))
}

```

The new code:

```

if (!missing(dx_cols)) {
  dxmat <- as.matrix(dplyr::mutate_all(
    dplyr::select(
      data, !!dplyr::enquo(dx_cols)),
    as.character))
} else {
  dxmat <- matrix("", nrow = nrow(data))
}

```

One of the tests written to verify the problem didn't reoccur:

```

# Due to previous use of sapply in ccc.R, this would fail
test_that(paste("1 patient with multiple rows of no diagnosis",
  "data--should have all CCCs as FALSE"), {
  expect_true(all(ccc(dplyr::data_frame(
    id = 'a',
    dx1 = NA,
    dx2 = NA),
    dx_cols = dplyr::starts_with("dx"),
    icdv = code) == 0))
}
)

```

**Testing Anti-Patterns:** While the above guidance should help researchers know the basics of testing, it does not cover in detail what not to do. An excellent collection of testing anti-patterns can be found at ([Moilanen, 2014](#); [Carr, 2015](#); [Stack Overflow Contributors, 2017](#)). Some key problems that novices experience when learning how to test software are:

- **Interdependent tests**—Interdependent tests can cause multiple test failures. When a failure in an early test case breaks a later test, it can cause difficulty in resolution and remediation.

- Testing application performance—While testing execution timing or software performance is a good idea and is covered more in the “Software Optimization” section, creating an automated test to perform this is difficult and does not carry over well from one machine to another.
- Slow running tests—As much as possible, tests should be automated but still run quickly. If the testing process takes too long consider refactoring tests or evaluating the performance of the software being tested.
- Only test correct input—A common problem in testing is to only validate expected inputs and desired behavior. Make sure tests cover invalid input, exceptions, and similar items.

### Software optimization

Getting software to run in a reasonable amount of time is always a key consideration when working with large datasets. A mathematical understanding of software algorithms is usually a key component of software engineering curricula, but not widely covered in other disciplines. Additionally, while software engineering texts and curricula highlight the importance of testing for non-functional requirements such as performance ([Sommerville, 2015](#)), they often fail to provide details on how best to evaluate software performance or how to plan for performance during the various phases of software lifecycle.

The survey of R packages at the beginning of this work indicates that approximately 75% of packages do not use optimization related packages nor compiled code to improve performance. While the survey of R packages is not evidence of non-optimization of packages in CRAN, computational researchers can should carefully consider performance aspects of their software before declaring it complete. This section will provide a starting point for additional study, research, and experimentation. The Python Foundation’s Python language wiki provides excellent high-level advice ([Python Wiki Contributors, 2018](#)) to follow before spending too much time in optimization: First get the software working correctly, test to see if it is correct, profile the application if it is slow, and lastly optimize based on the results of code profiling. If necessary, repeat multiple cycles of testing, profiling, and optimization phases. The key aspects of software optimization discussed in this are: identify a performance target, understanding and applying Big O notation, and the use code profiling and benchmarking tools.

#### Key recommendations for identifying and validating performance targets:

- Identify functional and non-functional requirements of the software being developed.
- If software performance is key to the software requirements, develop repeatable tests to evaluate performance.

**Discussion:** The first step to software optimization is to understand the functional and non-functional requirements of the software being built. Based on expected input, output, and platform the software will be run on, one can make a decision as to what is good enough for the software being developed. A pragmatic approach is best—do not

spend time optimizing if it does not add value. Once the functional requirements have been correctly implemented and validated, a decision point is reached: decide if the software is slow and in need of evaluation and optimization. While this may seem a trivial and unnecessary step, it should not be overlooked; a careful evaluation of costs versus benefit from an optimization effort should be evaluated before moving forward. Some methods for gathering the performance target are through an evaluation of other similar software, interdependencies of the software and its interaction with other systems, and discussion with other experts in the field.

Once a performance target has been identified, development of tests for performance can begin. While performance testing is often considered an anti-pattern of testing ([Moilanen, 2014](#)) some repeatable tests should be created to track performance as development progresses. Often a “stress test” or a test with greater than expected input/usage is the best way to do this. A good target is to check an order of magnitude larger input than expected. This type of testing can provide valuable insight into the performance characteristics of the software as well unearth potentials for failure due to unexpected load ([Sommerville, 2015](#)).

Here is an example of performance validation testing that can also serve as a basic reproducibility test calling the main function from pccc using the microbenchmark package (one could also use bench, benchr, or other similar R packages).

```
library(pccc)
rm(list=ls())
gc()

icd10_large <-
  feather::read_feather(
    "../icd_file_generator/icd10_sample_large.feather"
  )
library(microbenchmark)
microbenchmark(
  ccc(icd10_large[1:10000, c(1:45)], # get id, dx, and pc columns
    id = id,
    dx_cols = dplyr::starts_with("dx"),
    pc_cols = dplyr::starts_with("pc"),
    icdv = 10),
  times = 10)

Unit: seconds
  expr      min       lq     mean   median      uq      max  neval
ccc 2.857625 2.908964 2.959805 2.920408 3.023602 3.119937    10
```

Results are from a system with 3.1 GHz Intel Core i7, 16 GB 2133 MHz LPDDR3, PCI-Express SSD, running macOS 10.12.6 and R version 3.5.1 (2018-07-02).

As software runs can differ significantly from one to the next due to other software running on the test system, a good starting point is to run the same test 10 times (rather than the microbenchmark default of 100 due to this being a longer running process) and record the mean run time. `microbenchmark` also shows median, lower and upper quartiles, min, and max run times. The actual `ccc()` call specifics are un-important; the key is to test the main features of your software in a repeatable fashion and watch for performance changes over time. These metrics can help to identify if a test was valid and indicate a need for retesting; that is, a large interquartile range may indicate not enough tests were run or some aspect of environment is causing performance variations. Software benchmarking is highly system specific in that changing OS version, R version, R dependent package version, compiler version (if compiled code involved), or hardware may change the results. As long as all tests are run the same on the same system with the same software, one can compare timings as development progresses.

Lastly, although the example above is focused on runtime, it can be beneficial to also identify targets for disk space used and memory required to complete all desired tasks. As an example, tools such as `bench` and `profvis` demonstrated in our “Code Profiling/Benchmarking” section as well as `object.size()` from core R can give developers insight into memory allocation and usage. There are many resources beyond this work that can provide guidance on how to minimize RAM and disk resources ([Kane, Emerson & Weston, 2013](#); [Wickham, 2014b](#); [Wickham et al., 2016](#); [Klik, Collet & Facebook, 2018](#)).

#### Key recommendations for identifying upper bound on performance:

- Big O notation allows the comparison of theoretical performance of different algorithms.
- Evaluate how many times blocks of code will run as input approaches infinity.
- Loops inside loops are very slow as input approaches infinity.

**Discussion:** Big O notation is a method for mathematically determining the upper bound on performance of a block of code without consideration for language and hardware specifics. Although performance can be evaluated in terms of storage or run time, most examples and comparisons focus on run time. However, when working with large datasets, memory usage and disk usage can be of equal or higher importance than run. Big O notation is reported in terms of input (usually denoted as  $n$ ) and allows one to quickly compare theoretical performance of different algorithms.

The basic steps for evaluating the upper bound of performance of a block of software code is to evaluate what code will run as  $n$  approaches infinity. Items that are constant time (regardless of if they run once or  $x$  times independent of input) are reduced down to  $O(1)$ . The key factors that contribute to Big O are loops—a single for loop or similar construct through recursion that runs once for all  $n$  is  $O(n)$ ; a nested for loop would be  $O(n^2)$ . When calculating Big O for a code block, function, or software system, lower order terms are ignored, and just the largest Big O notation is used; for example, if a code block is  $O(1) + O(n) + O(n^3)$  it would be denoted as  $O(n^3)$ .

Despite the value of understanding the theoretical upper bound of software in an ideal situation, there are many difficulties that arise during implementation that can make Big O difficult to calculate and which could make a large Big O faster than a small Big O under actual input conditions. Some key takeaways to temper a mathematical evaluation of Big O are:

- Constants matter when choosing an algorithm—for example, if one algorithm is  $O(56n^2)$ , there exists some  $n$  where  $O(n^3)$  is faster.
- Average or best case run time might be more relevant.
- Big O evaluation of algorithms in high level languages is often hard to quantify.

For additional details on Big O notation, see the excellent and broadly understandable introduction to Big O notation ([Abrahms, 2016](#)).

#### **Key recommendations for profiling and benchmarking:**

- Profile code to find bottlenecks.
- Modify code to address largest items from profiling.
- Run tests to make sure functionality isn't affected.
- Repeat process if gains are made and additional performance improvements are necessary.

**Discussion:** As discussed throughout this section, optimization is a key aspect of software development, especially with respect to large datasets. Although identification of performance targets and a mathematical analysis of algorithms are important steps, the final result must be tested and verified. The only way to know if your software will perform adequately under ideal (and non-ideal) circumstances is to use benchmarking and code profiling tools. Code profilers show how a software behaves and what functions are being called while benchmarking tools generally focus on just execution time—though some tools combine both profiling and benchmarking. In R, some of the common tools are `bench`, `benchr`, `microbenchmark`, `tictoc`, `Rprof` ([R Core Team, 2018](#)), `prof` ([Tierney & Jarjour, 2016](#)), and `profvis`.

If, after implementation has been completed, the software functions correctly, and performance targets have not been met, look to optimize your code. Follow an iterative process of profiling to find bottlenecks, making software adjustments, testing small sections with benchmarking and then repeating the process with overall profiling again. If at any point in the process you discover that due to input size, functional requirements, hardware limitations, or software dependencies you cannot make a significant impact to performance, consider stopping further optimization efforts ([Burns, 2012](#)).

As with software testing and software bugs, the Pareto principle applies, though some put the balance between code and execution time is closer to 90% of time is in 10% of the code or even as high as 99% in 1% ([Xochellis, 2010](#); [Bird, 2013](#)). Identify the biggest bottlenecks via code profiling and focus only on the top issues first. As an example of how to perform code profiling and benchmarking in R, do the following:





**Figure 5** Profvis flame graph. Visual depiction of memory allocation/deallocation, execution time, and call stack. [Full-size !\[\]\(fcc3264021d438d9732560e78099f674\_img.jpg\) DOI: 10.7717/peerj-cs.175/fig-5](https://doi.org/10.7717/peerj-cs.175/fig-5)

First, use profvis to identify the location with the largest execution time:

```
library(pccc)
icd10_large <- feather::read_feather("icd10_sample_large.feather")
profvis::profvis({ccc(icd10_large[1:10000,],
                      id = id,
                      dx_cols = dplyr::starts_with("dx"),
                      pc_cols = dplyr::starts_with("pc"),
                      icdv = 10)}, torture = 100)
```

In [Fig. 5](#) you can see a visual depiction of memory allocation, known as a “Flame Graph,” as well as execution time and call stack. By clicking on each item in the stack you will be taken directly to the relevant source code and can see which portions of the code take the most time or memory allocations. [Figure 6](#) is a depiction of the data view which shows just the memory changes, execution time, and source file.

Once the bottleneck has been identified, if possible extract that code to a single function or line that can be run repeatedly with a library such as `microbenchmark` or `tictoc` to see if a small change either improves or degrades performance. Test frequently and make sure to compare against previous versions. You may find that something you thought would improve performance degrades performance. As a first step we recommend running `tictoc` to get general timings such as the following:

```
library(tictoc)
tic("timing: r version")
out <- dplyr::bind_cols(ids, ccc_mat_r(dxmat, pccmat, icdv))
toc()
```

```

tic("timing: c++ version")
dplyr::bind_cols(ids, ccc_mat_rcpp(dxmat, pccmat, icdv))
toc()
timing: r version: 37.089 sec elapsed
timing: c++ version: 5.087 sec elapsed

```

As with previous timings, while we're showing pccc calls, any custom function of block of code you have can be compared against an alternative version to see which performs better. The above blocks of code call the core functionality of the pccc package—one implemented all in R, the other with C++ for the matrix processing and string matching components; see sourcecode available at [https://github.com/magic-lantern/pccc/blob/no\\_cpp/R/ccc.R](https://github.com/magic-lantern/pccc/blob/no_cpp/R/ccc.R) for full listing.

After starting with high level timings, next run benchmarks on specific sections of code such as in this example comparing importing a package vs using the package reference operator using bench:

```

library(bench)
set.seed(42)
bench::mark(
  package_ref <- lapply(medium_input, function(i) {
    if(any(stringi::stri_startswith_fixed(i, 'S'), na.rm = TRUE))
      return(1L)
    else
      return(0L)
  })
)

```

```

# A tibble: 1 × 14
  expression    min  mean median    max `itr/sec` mem_alloc
  <chr>         <bch> <bch> <bch:> <bch>    <dbl>   <bch:byt>
1 package_r... 547ms 547ms 547ms 547ms     1.83    17.9MB

```

```

library(stringi)
bench::mark(
  direct_ref <- lapply(medium_input, function(i) {
    if(any(stri_startswith_fixed(i, 'S'), na.rm = TRUE))
      return(1L)
    else
      return(0L)
  })
)

```

```

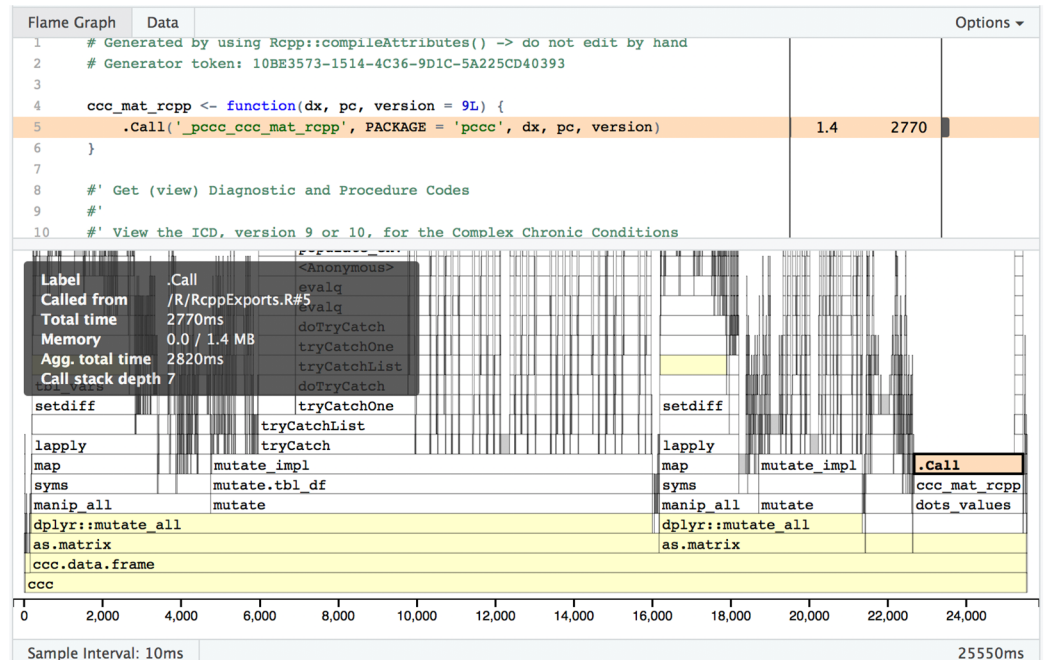
# A tibble: 1 × 14
  expression    min  mean median    max `itr/sec` mem_alloc
  <chr>         <bch> <bch> <bch:> <bch>    <dbl>   <bch:byt>
1 direct_re... 271ms 274ms 274ms 277ms     3.65    17.9MB

```

The above test was run on a virtual machine running Ubuntu 16.04.5 LTS using R 3.4.4.

Code	File	Memory (MB)	Time (ms)
ccc	<expr>	-1.0   6.2	25510
▼ ccc.data.frame	ccc.R	-1.0   6.1	25380
▼ as.matrix	ccc.R	-1.0   4.5	21270
▼ dplyr::mutate_all	ccc.R	-0.9   1.1	21000
▼ mutate		-0.4   0.5	13890
▼ mutate.tbl_df		-0.4   0.5	13890
▶ mutate_impl		-0.4   0.5	13870
lazyLoadDBfetch		0   0.0	10
▶ named_quos		0   0.0	10
▶ manip_all		-0.5   0.6	7100
lazyLoadDBfetch		0   0.0	10
▶ as.matrix.data.frame		0.0   1.2	190
▶ get		0.0   0.1	50
dxmat <- as.matrix(dplyr::mutate_all(dplyr::select(data, !!dplyr::enquo(d	ccc.R	0   0.0	20

**Figure 6 Profvis data chart.** Table view of memory allocation/deallocation, execution time, and call stack. [Full-size](#) DOI: 10.7717/peerj-cs.175/fig-6



**Figure 7 Profvis flame graph .Call().** Visual depiction of memory allocation/deallocation, execution time, and call stack; note the limitations in detail at the .Call() function where custom compiled code is called. [Full-size](#) DOI: 10.7717/peerj-cs.175/fig-7

One benefit of `bench::mark` over `microbenchmark` is that `bench` reports memory allocations as well as timings, similar to data shown in `profvis`. Through benchmarking we found that for some systems/configurations the use of the “:” operator, as opposed to importing a package, worsened performance noticeably. Also widely known (*Gillespie & Lovelace, 2017*) and found to be applicable here is that the use of matrices are preferred for performance reasons over data.frames or tibbles. Matrices do have different functionality, which can require some re-work when converting from one to another. For example, a matrix can only contain 1 data type such as character or numeric;

data.frames and tibbles support shortcut notations such as `mydf$colname`. Another key point found is that an “env” with no parent environment is significantly faster (up to 50x) than one with a parent env. In the end, optimization efforts resulted in reducing run time by 80%.

One limitation with R profiling tools is that if the code to be profiled executes C++ code, you will get no visibility into what is happening once the switch from R to C++ has occurred. As shown in Fig. 7, visibility into timing and memory allocation stops at the `.Call()` function. In order to profile C++ code, you need to use non-R specific tools such as XCode on macOS or gprof on non-macOS Unix-based operating system (OS). See “R\_with\_C++\_profiling.md” in our source code repository for some guidance on this topic.

Some general lessons learned from profiling and benchmarking:

- “Beware the dangers of premature optimization of your code. Your first duty is to create clear, correct code.” (*Knuth, 1974; Burns, 2012*) Never optimize before you actually know what is taking all the time/memory/space with your software. Different compilers and core language updates often will change or reverse what experience has previously indicated as sources of slowness. Always benchmark and profile before making a change.
- Start development with a high-level programming language first—Developer/Researcher time is more valuable than CPU/GPU time. Choose the language that allows the developer/researcher to rapidly implement the desired functionality rather than selecting a language/framework based on artificial benchmarks (*Kelleher & Pausch, 2005; Jones & Bonsignour, 2011*).
- Software timing is highly OS, compiler, and system configuration specific. What improves results greatly on one machine and configuration may actually slow performance on another machine. Once you decided to put effort into optimization, make sure you test on a range of realistic configurations before deciding that an “improvement” is beneficial (*Hyde, 2009*).
- If you’ve exhausted your options with your chosen high-level language, C++ is usually the best option for further optimization. For an excellent introduction to combining C++ with R via the library Rcpp, see (*Eddelbuettel & Balamuta, 2017*).

For some additional information on R optimization, see (*Wickham, 2014b; Robinson, 2017*).

## 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.

## ADDITIONAL INFORMATION AND DECLARATIONS

### Funding

The University of Colorado Data Science to Patient Value initiative provided funding for this work. The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

### Grant Disclosure

The following grant information was disclosed by the authors:  
University of Colorado Data Science to Patient Value.

### Competing Interests

The authors declare that they have no competing interests.

### Author Contributions

- Seth Russell conceived and designed the experiments, performed the experiments, analyzed the data, contributed reagents/materials/analysis tools, prepared figures and/or tables, performed the computation work, authored or reviewed drafts of the paper, approved the final draft.
- Tellen D. Bennett conceived and designed the experiments, authored or reviewed drafts of the paper, approved the final draft.
- Debashis Ghosh conceived and designed the experiments, authored or reviewed drafts of the paper, approved the final draft.

### Data Availability

The following information was supplied regarding data availability:

Source code, images, generated data for R package analysis are available at:  
<https://github.com/magic-lantern/SoftwareEngineeringPrinciples>.

Materials from paper including figures, references, and text are available at:  
<https://github.com/magic-lantern/SoftwareEngineeringPrinciples/tree/master/paper>.

### Supplemental Information

Supplemental information for this article can be found online at <http://dx.doi.org/10.7717/peerj-cs.175#supplemental-information>.

## REFERENCES

- Abrahms J. 2016. Big-O notation explained by a self-taught programmer. Available at <https://justin.abrah.ms/computer-science/big-o-notation-explained.html>.

- 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. 2018a. *RcppParallel: parallel programming tools for "Rcpp"*. Available at <https://CRAN.R-project.org/package=RcppParallel>.
- Allaire JJ, Xie Y, McPherson J, Luraschi J, Ushey K, Atkins A, Wickham H, Cheng J, Chang W, Iannone R. 2018b. *rmarkdown: dynamic documents for R*. Available at <https://rmarkdown.rstudio.com>.
- 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. *Communications of the ACM* 11(3):151–197  
DOI 10.1145/362929.362976.
- Beck K, Beedle M, Van Bennekum A, Cockburn A, Cunningham W, Fowler M, Grenning J, Highsmith J, Hunt A, Jeffries R, Kern J, Marick B, Martin RC, Mellor S, Schwaber K, Sutherland J, Thomas D. 2001. Manifesto for agile software development. Available at <http://agilemanifesto.org/>.
- Beck K, Gamma E. 1998. Test infected: programmers love writing tests. *Java Report* 3:37–50. Available at <http://members.pingnet.ch/gamma/junit.htm>.
- Bengtsson H. 2018. *future: unified parallel and distributed processing in R for everyone*. Available at <https://CRAN.R-project.org/package=future>.
- Bengtsson H, R Core Team. 2018. *future.apply: apply function to elements in parallel using futures*. Available at <https://CRAN.R-project.org/package=future.apply>.
- Bird J. 2013. Applying the 80:20 rule in software development - DZone Agile. Available at <https://dzone.com/articles/applying-8020-rule-software>.
- Bischi B, Lang M. 2015. *parallelMap: unified interface to parallelization back-ends*. Available at <https://CRAN.R-project.org/package=parallelMap>.
- Bischi 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*. Available at <https://CRAN.R-project.org/package=RUnit>.
- Burns P. 2012. *The R Inferno*. Available at <https://lulu.com>.
- Calaway R, Analytics R, Weston S. 2017. *doMC: foreach parallel adaptor for "parallel"*. Available at <https://CRAN.R-project.org/package=doMC>.
- Calaway R, Corporation M, Weston S. 2017. *doSNOW: foreach parallel adaptor for the "snow" package*. Available at <https://CRAN.R-project.org/package=doSNOW>.
- Calaway R, Corporation M, Weston S, Tenenbaum D. 2018. *doParallel: foreach parallel adaptor for the "parallel" package*. Available at <https://CRAN.R-project.org/package=doParallel>.
- Calaway R, Microsoft, Weston S. 2017. *foreach: provides foreach looping construct for R*. Available at <https://CRAN.R-project.org/package=foreach>.
- Carr J. 2015. TDD anti-patterns. Available at <https://web.archive.org/web/20150726134212/http://blog.james-carr.org:80/2006/11/03/tdd-anti-patterns/>.

- Chang W, Luraschi J. 2018.** *profvis: interactive visualizations for profiling R code.* Available at <https://CRAN.R-project.org/package=profvis>.
- Dehaghani SMH, Hajrahimi N. 2013.** Which factors affect software projects maintenance cost more? *Acta Informatica Medica* **21(1)**:63–66 DOI 10.5455/AIM.2012.21.63-66.
- DeWitt P, Bennett T, Feinstein J, Russell S. 2017.** *pccc: pediatric complex chronic conditions.* Available at <https://cran.r-project.org/package=pccc>.
- Dragulescu AA, Arendt C. 2018.** *xlsx: read, write, format excel 2007 and excel 97/2000/XP/2003 files.* Available at <https://CRAN.R-project.org/package=xlsx>.
- Eckert A. 2018.** *parallelDist: parallel distance matrix computation using multiple threads.* Available at <https://CRAN.R-project.org/package=parallelDist>.
- Eddelbuettel D, Balamuta JJ. 2017.** Extending R with C++: a brief introduction to *Rcpp*. *PeerJ* **5**:e3188v1 DOI 10.7287/peerj.preprints.3188v1.
- Feinerer I, Theussl S, Buchta C. 2015.** *DSL: distributed storage and list.* Available at <https://CRAN.R-project.org/package=DSL>.
- Feinstein JA, Russell S, DeWitt PE, Feudtner C, Dai D, Bennett TD. 2018.** R package for pediatric complex chronic condition classification. *JAMA Pediatrics* **172(6)**:596 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(1)**:199 DOI 10.1186/1471-2431-14-199.
- Fucci D, Scanniello G, Romano S, Shepperd M, Sigweni B, Uyaguari F, Turhan B, Juristo N, Oivo M. 2016.** An external replication on the effects of test-driven development using a multi-site blind analysis approach. In: *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. ESEM '16*, Vol. 3. New York: ACM, 1–3, 10.
- Gaslam B. 2017.** *unitizer: interactive R unit tests.* Available at <https://CRAN.R-project.org/package=unitizer>.
- Gillespie C, Lovelace R. 2017.** *Efficient R programming: a practical guide to smarter programming.* Sebastopol: O'Reilly Media.
- Glass RL. 2001.** Frequently forgotten fundamental facts about software engineering. *IEEE Software* **18(3)**:112–111 DOI 10.1109/MS.2001.922739.
- Grosjean P. 2014.** *SciViews-R: A GUI API for R.* Mons: UMONS.
- Hansson DH. 2014.** TDD is dead. Long live testing. (DHH). Available at <http://david.heinemeierhansson.com/2014/tdd-is-dead-long-live-testing.html>.
- Hester J. 2018.** *bench: high precision timing of R expressions.* Available at <https://cran.r-project.org/package=bench>.
- Hyde R. 2009.** The fallacy of premature optimization. *Ubiquity* **2009**:1 DOI 10.1145/1569886.1513451.
- Izrailev S. 2014.** *tictoc: functions for timing R scripts, as well as implementations of Stack and List structures.* Available at <https://CRAN.R-project.org/package=tictoc>.
- Jones C, Bonsignour O. 2011.** *The economics of software quality.* Upper Saddle River: Addison-Wesley Professional.
- Kane M, Emerson J, Weston S. 2013.** Scalable strategies for computing with massive data. *Journal of Statistical Software* **55(14)**:1–19 DOI 10.18637/jss.v055.i14.

- Kanewala U, Bieman JM. 2014.** Testing scientific software: a systematic literature review. *Information and Software Technology* **56(10)**:1219–1232 DOI [10.1016/j.infsof.2014.05.006](https://doi.org/10.1016/j.infsof.2014.05.006).
- Kelleher C, Pausch R. 2005.** Lowering the barriers to programming: a taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys* **37(2)**:83–137 DOI [10.1145/1089733.1089734](https://doi.org/10.1145/1089733.1089734).
- Klevtsov A, Antonov A, Upravitelev P. 2018.** *benchr: high precise measurement of R expressions execution time*. Available at <https://cran.r-project.org/package=benchr>.
- Klik M, Collet Y, Facebook. 2018.** *fst: lightning fast serialization of data frames for R*. Available at <https://CRAN.R-project.org/package=fst>.
- Knuth DE. 1974.** Structured programming with go to statements. *ACM Computing Surveys* **6(4)**:261–301 DOI [10.1145/356635.356640](https://doi.org/10.1145/356635.356640).
- 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*. Available at <https://CRAN.R-project.org/package=rbenchmark>.
- Leek JT, Peng RD. 2015.** Opinion: reproducible research can still be wrong: adopting a prevention approach. *Proceedings of the National Academy of Sciences of the United States of America* **112(6)**:1645–1646 DOI [10.1073/pnas.1421412111](https://doi.org/10.1073/pnas.1421412111).
- Lentin J, Hennessey A. 2017.** *unittest: TAP-compliant unit testing*. Available at <https://CRAN.R-project.org/package=unittest>.
- Luraschi J, Kuo K, Ushey K, Allaire JJ, Macedo S, RStudio, Foundation TAS. 2018.** *sparklyr: R interface to Apache Spark*. Available at <https://CRAN.R-project.org/package=sparklyr>.
- Matloff N. 2016.** Software alchemy: turning complex statistical computations into embarrassingly-parallel ones. *Journal of Statistical Software* **71(4)**:1–15 DOI [10.18637/jss.v071.i04](https://doi.org/10.18637/jss.v071.i04).
- Mersmann O. 2018.** *microbenchmark: accurate timing functions*. Available at <https://CRAN.R-project.org/package=microbenchmark>.
- Micro Focus. 2018.** Unified functional testing. Available at <https://software.microfocus.com/en-us/products/unified-functional-automated-testing/overview> (accessed 12 April 2018).
- Moilanen J. 2014.** Test driven development details. Available at <https://github.com/educloudalliance/educloud-development/wiki/Test-Driven-Development-details> (accessed 12 April 2018).
- Nolan R, Padilla-Parra S. 2017.** *exampletestr—An easy start to unit testing R packages*. *Wellcome Open Research* **2**:31 DOI [10.12688/wellcomeopenres.11635.2](https://doi.org/10.12688/wellcomeopenres.11635.2).
- Nutter B, Lane S. 2018.** *redcapAPI: accessing data from REDCap projects using the API*. *Zenodo*. DOI [10.5281/zenodo.592833](https://doi.org/10.5281/zenodo.592833).
- Osborne JM, Bernabeu MO, Bruna M, Calderhead B, Cooper J, Dalchau N, Dunn S-J, Fletcher AG, Freeman R, Groen D, Knapp B, McNerny GJ, Mirams GR, Pitt-Francis J, Sengupta B, Wright DW, Yates CA, Gavaghan DJ, Emmott S, Deane C. 2014.** Ten simple rules for effective computational research. *PLOS Computational Biology* **10(3)**:e1003506 DOI [10.1371/journal.pcbi.1003506](https://doi.org/10.1371/journal.pcbi.1003506).
- Prins P, De Ligt J, Tarasov A, Jansen RC, Cuppen E, Bourne PE. 2015.** Toward effective software solutions for big biology. *Nature Biotechnology* **33(7)**:686–687 DOI [10.1038/nbt.3240](https://doi.org/10.1038/nbt.3240).
- Python Wiki Contributors. 2018.** Performance tips. Available at <https://wiki.python.org/moin/PythonSpeed/PerformanceTips> (accessed 12 April 2018).



- R Core Team. 2018.** *R: a language and environment for statistical computing*. Vienna: R Foundation for Statistical Computing. Available at <http://www.R-project.org/>.
- Ranorex GmbH. 2018.** Ranorex. Available at <https://www.ranorex.com> (accessed 12 April 2018).
- Reese J. 2018.** Best practices for writing unit tests. Available at <https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices>.
- Robinson E. 2017.** Making R code faster : a case study. Available at <https://robinsones.github.io/Making-R-Code-Faster-A-Case-Study/>.
- Rooney P. 2002.** Microsoft's CEO: 80-20 rule applies to bugs, not just features. Available at <http://www.crn.com/news/security/18821726/microsofts-ceo-80-20-rule-applies-to-bugs-not-just-features.htm>.
- Sandve GK, Nekrutenko A, Taylor J, Hovig E. 2013.** Ten simple rules for reproducible computational research. *PLOS Computational Biology* **9**(10):e1003285  
DOI 10.1371/journal.pcbi.1003285.
- Selenium Contributors. 2018.** Selenium. Available at <https://www.seleniumhq.org> (accessed 12 April 2018).
- Sommerville I. 2015.** *Software engineering*. Boston: Pearson.
- Sommerville I. 2016.** Giving up on test-first development. Available at <http://iansommerville.com/systems-software-and-technology/giving-up-on-test-first-development/>.
- Stack Overflow Contributors. 2017.** Unit testing anti-patterns catalogue. Available at <https://stackoverflow.com/questions/333682/unit-testing-anti-patterns-catalogue> (accessed 12 April 2018).
- The Joint Task Force on Computing Curricula. 2015.** *Curriculum guidelines for undergraduate degree programs in software engineering*. New York: ACM.
- Tierney L, Jarjour R. 2016.** *proftools: profile output processing tools for R*. Available at <https://CRAN.R-project.org/package=proftools>.
- Tierney L, Rossini AJ, Li N, Sevcikova H. 2018.** *snow: simple network of workstations*. Available at <https://CRAN.R-project.org/package=snow>.
- Weston S. 2017.** *doMPI: foreach parallel adaptor for the Rmpi package*. Available at <https://CRAN.R-project.org/package=doMPI>.
- Wickham H. 2011.** testthat: get started with testing. *R Journal* **3**:5–10.
- Wickham H. 2014a.** *profr: an alternative display for profiling information*. Available at <https://CRAN.R-project.org/package=profr>.
- Wickham H. 2014b.** *Advanced R*. Boca Raton: Chapman and Hall/CRC.
- Wickham H, RStudio, Feather developers, Google, LevelDB Authors. 2016.** *feather: R bindings to the feather "API"*. Available at <https://CRAN.R-project.org/package=feather>.
- Wikipedia contributors. 2017a.** SUnit — Wikipedia, the free encyclopedia. Available at <https://en.wikipedia.org/w/index.php?title=SUnit&oldid=815835062>.
- Wikipedia contributors. 2017b.** XUnit — Wikipedia, the free encyclopedia. Available at <https://en.wikipedia.org/w/index.php?title=XUnit&oldid=807299841>.
- Wilson G. 2016.** Software carpentry: lessons learned. *F1000Research* **3**:62  
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**(1):e1001745  
DOI 10.1371/journal.pbio.1001745.

- Xie Y. 2018.** *testit: a simple package for testing R packages.* Available at <https://CRAN.R-project.org/package=testit>.
- Xie Y, Allaire JJ, Golemund G. 2018.** *R markdown: the definitive guide.* Boca Raton: Chapman and Hall/CRC.
- Xochellis J. 2010.** The impact of the Pareto principle in optimization - CodeProject. Available at <https://www.codeproject.com/Articles/49023/The-impact-of-the-Pareto-principle-in-optimization>.
- Yu H. 2002.** Rmpi: parallel statistical computing in R. *R News* 2:10–14.