Software engineering principles applied to research software

Seth Russell

**Proposed Title: The application of software engineering principles to improve quality and performance of statistical software for large data sets**

Note: Can’t use full proposed title as it causes problems with GitHub integration.

**Questions:**

• Should this be split into 2 papers?

• Should the paper just focus on one of the 2 high level areas?

• Journal of Statistical Software has no limitations on length nor figures/tables.

**Review of standard software engineering precepts – why dive into these?**

From Wilson et al. - 2014 - Best Practices for Scientific Computing.pdf

basic software development practices such as writing maintainable code, using version control and issue trackers, code reviews, unit testing, and task automation.

# Introduction

In todays environment of Wilson et al in their 2014 paper covered key areas where scientists can benefit from software engineering best practices (Wilson et al. 2014) s

The R package “pccc: Pediatric Complex Chronic Conditions” (DeWitt et al. 2017) (PCCC) is used for code examples in this article. PCCC is an R/C++ implementation of the Pediatric Complex Chronic Conditions software released as part of a series of research papers by Feudtner et al <reference>. The goal of PCCC is to take in a data set containing International Statistical Classification of Diseases and Related Health Problems (ICD) Ninth revision or Tenth revision diagnosis or procedure codes and output which if any complex chronic conditions a patient has.

**Knowledge Gap:**

(Glass 2001; Koskinen 2015; Dehaghani and Hajrahimi 2013)There is a need to adopt principles from the domain of software engineering in the development of statistical software.

**Overall Message:**

Adoption of Software Engineering principles in the development of research focused software for large data sets will yield more accurate and performant software packages.

# Body

## Maintenance

With the development of any software artifact, the key consideration to implementation should be maintenance. As many research scientists tend to think of their software products as unique tools that will not be used regularly or for a long period, most don’t think about long term maintenance when in the development phase. From my professional software experience, starting with a long term view saves time and effort over the long run when a software product unexpectedly has long term use or success.

From various sources the consensus is that software maintenance costs are large and increasing (Glass 2001; Koskinen 2015; Dehaghani and Hajrahimi 2013); some put maintenance at 90% of total software cost. The chief factor in cost of maintenance with respect to research and statistical 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 and Peng 2015). A reproducible and replicable solution is one that requires a long term maintenance oriented view.

There are many techniques that can help to reduce cost of maintenance and speed development time. While best practices such as the use of version control software, open access to results and papers are becoming wide spread, there are some that are important but need further attention: documentation, language choice, and software testing practices.

### Documentation

While the purpose of software is to instruct a computer to perform a specific operation, with current technologies, that instruction must be created by humans. Software documentation conveys information to other users or developers through a richer language than that of the selected computing language selected. One of the most influential papers in this area is “Literate Programming”  (Knuth 1984). Although decades old at this time, its principles have yet to become common practice among non-computer science trained researchers. The key aspects of literate programming are weaving, creation of a single document that is both software code and description of that code, and tangling, a process by which written documentation and machine code is produced from a single file.

In the R language literate programming can be accomplished with specially formatted comments and the package roxygen2 (Wickham et al. 2017). An abbreviated example taken from a function header looks like this:

#' Complex Chronic Conditions (CCC)
#'
#' Generate CCC and CCC subcategory flags and the number of categories.
#'
#' It is recommended that you view the codes defining the CCC via
#' \code{\link{get\_codes}} and make sure that the ICD codes in your data set are
#' formatted in the same way. The ICD codes used for CCC are character strings
#' must be formatted as follows:
#' \itemize{
#' \item \*Do not\* use decimal points or other separators
#' \item ICD 9 codes: Codes less than 10 should be left padded with 2 zeros. Codes
#' less than 100 should be left padded with 1 zero.
#' }
#'
#' See `vignette("pccc-overview")` for more details.
#'
#' @references
#' See \code{\link{pccc-package}} for published paper on the topic of identifying
#' Complex Chronic Conditions
#'
#' @param data a \code{data.frame} containing a patient id and all the ICD-9-CM
#' or ICD-10-CM codes. The \code{data.frame} passed to the function should be
#' in wide format.
#'
#' @return A \code{data.frame} with a column for the subject id and integer (0
#' or 1) columns for each each of the categories.
#'
#' @example examples/ccc.R
#'

Documentation identifies key function input requirements, references for more information, parameters, and return values. As with any written communication, the focus should be on the user new to the software package - or to your future self who has forgotten what the project was due to passage of time.

### Language Choice

While the Internets continuously debate the merits of one programming language over another and the relative performance of one language over another, a more pragmatic approach is recommended that is situation specific. The key consideration of language selection should be efficiency for humans: what can a person both write quickly and understand quickly. This consideration should include not only base language functionality, but the library of capabilities built by others and made available for others to use. After the human concerns have been addressed, attention should be placed on other characteristics including performance, portability, etc.

As discussed previously, software maintenance is a large portion of any software lifecycle. As maintenance involves repeated reading and understanding by humans of what has already been written, it is a large part of the overall cost of software.  While experience, training, and available tools to solve a particular problem has a large impact on both creation and understanding of software, when faced with options, users should choose a language that favors ways people think rather than ways computers operate. <Mashey ACM reference>.

Programming languages such as R that are distributed typically as source files rather than binaries, do not require compilation to run

Low level languages require compilation

<https://queue.acm.org/detail.cfm?id=1039532>

As an example, here is an example of a code snippet written in C++:

**Need to come up with better example - these two are NOT actually equivalent.**

 for (dxitr = 0; dxitr < dx.size(); ++dxitr) {
 for (itr = 0; itr < dx\_codes.size(); ++itr) {
 if (dx[dxitr].compare(0, dx\_codes[itr].size(),dx\_codes[itr]) == 0) {
 return 1;
 }
 }
 }

and one in R:

lapply(dxcodes, function(c) {
 any(stri\_startswith\_fixed(dx, c),na.rm = TRUE)
})

### Software Testing

Any time software is written as part of a research project, careful consideration should be employed on how to verify that the software performs the desired functionality and produces the desired output. As with bench science, software can often have un-expected and un-intended results due to minor or even major problems during the implementation process. In the software engineering field, software testing is a major component of any software development lifecycle and should also be a key component of research software.

Various methodologies exist for software testing and validation; furthermore, there are various strategies as to how software testing should be integrated into the software development lifecycle. Some common testing strategies are no strategy, manual testing, test driven development <reference>, large structured projects with testing phase, and mini iterations with testing along the way. While a full discussion of various methods and strategies is beyond the scope of this article, there are 3 key concepts that are common that should be addressed: when to start testing, what to test, and how to test?

**Testing should include more than just validating output  - code reviews/concept reviews with others**

**When to test**

While some strategies recommend writing automated test cases before any functionality (test driven development), a better approach that matches the flexible and changing nature of research software is to create tests after a requirement has been implemented. As developing comprehensive tests of software functionality can be a large burden to accrue at a single point in time, a recommended 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.

* <http://iansommerville.com/systems-software-and-technology/giving-up-on-test-first-development/>
* <http://david.heinemeierhansson.com/2014/tdd-is-dead-long-live-testing.html>

**What to test**

In an ideal world, any software developed would be accompanied by 100% test coverage validating all aspects of functionality and interaction with other software. However, due to pressures of research, having enough time to build a perfect test suite isn’t always realistic. A parsimonious application of the Pareto principle will go a long way towards improving overall software quality without adding too much testing burden. To apply this principle, 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’s the most likely bad outcome?
* For computationally intensive components - how long should this take to run?

Once answers to these questions are known, spend time developing tests to validate key features, avoiding major negatives, and ensuring software performs adequately.

**How to test**

Most programming languages have As mentioned previously, a pragmatic approach to testing combination of unit level and acceptance style testing - where tests are conducted to verify software meets requirements - is often best suited to the smaller scale and ad-hoc nature of research software.

unit testing

acceptance testing is a test conducted to determine if the requirements of a specification or contract are met

With a focus on developing functionality first, tests should be built once something has been created that needs to be validated.

Unit Testing

End to end testing - It is important to create tooling to help validate functionality beyond basic unit tests.

Common packages RUnit, testthat

Through unit testing, discovered many bugs that had existed in previous versions that hadn’t been caught. Example:

**Application of Software Testing to R package development**

For the PCCC R package there are a large set of ICD codes that and code set patterns that are used to determine if an input record meets a complex chronic condition criteria. To validate the correct functioning of the software, the first priority was to validate the ICD code groupings were correct and were mutually exclusive (as appropriate). As PCCC is a re-implementation of 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. 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
}))

While the specific codes were being compared, validated, and agreed upon, unit tests were written to validate the functionality of the new package. 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 initial automated testing. While manual running of a selection of test cases is a good start, a more efficient approach is to integrate testing into the package build process.  For unit testing, we selected the testthat <reference> package. After these initial tests, further thought went into edge cases and ways a user of the package might mistakenly call our package.

Another common pattern is to create a test case for discovered bugs - this ensures that a regression of this type does not happen again:

 # Due to previous use of sapply in ccc.R, this would fail - fixed now
 test\_that("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))
 })

Anti-patterns

* Having test cases depend on system state manipulated from previously executed test cases (i.e., you should always start a unit test from a known and pre-configured state).
* Dependencies between test cases. A test suite where test cases are dependent upon each other is brittle and complex. Execution order should not be presumed. Basic refactoring of the initial test cases or structure of the UUT causes a spiral of increasingly pervasive impacts in associated tests.
* Interdependent tests. Interdependent tests can cause cascading false negatives. A failure in an early test case breaks a later test case even if no actual fault exists in the UUT, increasing defect analysis and debug efforts.
* Testing precise execution behavior timing or performance.
* Slow running tests.

<https://www.r-bloggers.com/unit-tests-for-r-packages/>

References:

testthat: Get Started with Testing @Article{testthat,

  author = {Hadley Wickham},

  journal = {The R Journal},

  month = {June},

  number = {1},

  pages = {5—-10},

  title = {testthat: Get Started with Testing},

  url = {<http://journal.r-project.org/archive/2011-1/RJournal_2011-1_Wickham.pdf>},

  volume = {3},

  year = {2011},

  bdsk-url-1 = {<http://journal.r-project.org/archive/2011-1/RJournal_2011-1_Wickham.pdf>},

}

M. Burger, K. Juenemann, and T. Koenig. RUnit: R Unit test framework, 2009. URL http://CRAN. R-project.org/package=RUnit. R package ver- sion 0.4.22.

xUnit reference?

## Software Optimization

Key aspects of software optimization discussed in this article are: identify a performance target, understanding Big O notation, and finally to use code profiling tools.

The first step to any optimization problem is to understand the functionality and requirements of the software being built. Based on expected user input, the expected platform the software will be run on, and under expected circumstances one can decide

* Identify performance target
* Big O Notation - <https://justin.abrah.ms/computer-science/big-o-notation-explained.html>
* Code Profiling - <https://www.codeproject.com/Articles/49023/The-impact-of-the-Pareto-principle-in-optimization#References>, <http://www.crn.com/news/security/18821726/microsofts-ceo-80-20-rule-applies-to-bugs-not-just-features.htm>, <https://en.wikipedia.org/wiki/Pareto_principle>, <https://dzone.com/articles/applying-8020-rule-software>
* R
* C++
* When faced with multiple options to solve same problem, use microbenchmark. Some examples of what is faster – prefer matrix over data.frame, don’t use “::”,an env with no parent environment is about 50x faster than one with a parent env, etc.

<https://peerj.com/preprints/3139.pdf>

<https://peerj.com/preprints/3210.pdf>

<https://peerj.com/preprints/3205.pdf>

<https://peerj.com/preprints/3204.pdf>

<https://peerj.com/preprints/3192.pdf>

# Conclusion

# Acknowledgements

# References

Wilson, Greg, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H. D. Haddock, et al. 2014. “Best Practices for Scientific Computing”. *PLOS Biology* 12 (1): e1001745. doi:10.1371/journal.pbio.1001745.

DeWitt, Peter, Tell Bennett, James Feinstein, and Seth Russell. 2017. “Pccc: Pediatric Complex Chronic Conditions”. <https://cran.r-project.org/web/packages/pccc/index.html.>

Glass, Robert L. 2001. “Frequently Forgotten Fundamental Facts About Software Engineering”. *IEEE Softw.* 18 (3): 112–11. doi:10.1109/MS.2001.922739.

Koskinen, Jussi. 2015. “Software Maintenance Costs”. In . <https://wiki.uef.fi/download/attachments/38669960/SMCOSTS.pdf.>

Dehaghani, Sayed Mehdi Hejazi, and Nafiseh Hajrahimi. 2013. “Which Factors Affect Software Projects Maintenance Cost More?”. *Acta Informatica Medica* 21 (1): 63–66. doi:10.5455/AIM.2012.21.63-66.

Leek, Jeffrey T., and Roger D. Peng. 2015. “Opinion: Reproducible Research Can Still Be Wrong: Adopting a Prevention Approach”. *Proceedings of the National Academy of Sciences* 112 (6): 1645–46. doi:10.1073/pnas.1421412111.

Knuth, D. E. 1984. “Literate Programming”. *The Computer Journal* 27 (2): 97–111. doi:10.1093/comjnl/27.2.97.

Wickham, Hadley, Peter Danenberg, Manuel Eugster, and RStudio. 2017. “roxygen2: In-Line Documentation for R”. [https://CRAN.R-project.org/package=roxygen2.](https://CRAN.R-project.org/package%3Droxygen2.)