**SOFTWARE TESTING**

Whenever software is written as part of a research project, careful consideration should be given 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 literature for over 50 years, software testing is a major component of any software development lifecycle (Atchison et al., 1968) and should also be a key component of research software. As will be shown later, even among R software packages intended to be shared with and used by others, the majority of R packages (67% to 73% depending on metric, see next section) do not have tests that are made available with the package. From our experience, it is estimated that the amount of research software that has been formally tested is much lower.

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 waterfall, where testing is a major phase that occurs at a specific point in time, and agile, where there are many small iterations with testing. While a full discussion of various methods and strategies is beyond the scope of this article, following the “Analysis of testing of R Packages on CRAN” section, three key concepts presented are: when to start testing, what to test, and how to test.

**WHEN TO TEST**

One of the popular movements in recent years has been to develop tests first then work to implement code to meet desired functionality, a strategy called test driven development. While the test driven development strategy has done much to improve the focus of the software engineering world on testing, some have found that it does not meet 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.  
  
**WHAT TO TEST**

In an ideal world, any software developed would be accompanied by 100% test coverage validating all lines of code, all aspects of functionality, 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. General guidance on test design and addressing issues that break software are in the remainder of this section. 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.

**HOW TO TEST**

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 an 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 via post-commit hook in a version control system are 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 (formely 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 RMarkdown (Xie, 2015) 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.

**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.

**APPLICATION OF SOFTWARE TESTING TO R PACKAGE DEVELOPMENT**

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. 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, 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 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.

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 inclusion in automated testing. After these initial tests, further thought went into edge cases to check how the software might behave if the input data was incorrect or if parameters were not specified correctly.

Another common pattern is to create a test case for discovered bugs - 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.

Here’s 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))

}

)