

# RUnit - A Unit Test Framework for R

Thomas König, Klaus Jünemann, and Matthias Burger

May 18, 2004

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Getting Started: Setting up test cases</b>	<b>4</b>
2.1	R Unit Testing . . . . .	4
2.2	R Code Inspection . . . . .	4
2.2.1	Introduction . . . . .	4
2.2.2	Usage . . . . .	5
2.2.3	Technical Details . . . . .	5
<b>3</b>	<b>Future Development Ideas</b>	<b>6</b>

## Abstract

Software development for production systems presents a challenge to the development team as the quality of the coded package(s) has to be constantly monitored and verified. We present a generic approach to software testing for the R language modelled after successful examples such as JUnit, CppUnit, and PerlUnit. The aim of our approach is to facilitate development of reliable software packages and provide a set of tools to analyse and report the software quality status. The presented framework is completely implemented within R and does not rely on external tools or other language systems. The basic principle is that every function or method is accompanied with a test case that queries many calling situations including incorrect invocations. A test case can be executed instantly without reinstalling the whole package - a feature that is necessary for parallel development of functionality and test cases. On a second level one or more packages can be tested in a single test run, the result of which is reported in an well structured test protocol. To

verify the coverage of the test framework a code inspector is provided that monitors the code coverage of executed test cases. The result of individual test invocations as well as package wide evaluations can be compiled into a summary report exported to HTML. This report details the executed tests, their failure or success, as well as the code coverage. Taking it one step further and combining the build system with a development and release procedure with defined code status description this approach opens the way for a principled software quality monitoring and risk assessment of the developed application. For our code development we have utilized the described system with great benefit w.r.t. code reliability and maintenance efforts in a medium sized development team.

## 1 Introduction

The importance of software testing can hardly be overrated. This is all the more true for interpreted languages where not even a compiler checks the basic consistency of a program. Nonetheless, testing is often perceived more as a burden than a help by the programmer. Therefore it is necessary to provide tools that make the task of testing as simple and systematic as possible. The key goal of such a testing framework should be to promote the creation and execution of test cases to become an integral part of the software development process. Experience shows that such a permanently repeated code - test - simplify cycle leads to faster and more successful software development than the usually futile attempt to add test cases once the software is largely finished. This line of thought has been pushed furthest by the Extreme Programming [[?]] and Test-First paradigms where test cases are viewed as the essential guidelines for the development process. These considerations lead to various requirements that a useful testing framework should satisfy:

- ^ Tests should be easy to execute.
- ^ The results should be accessible through a well structured test protocol.
- ^ It should be possible to execute only small portions of the test cases during the development process.
- ^ It should be possible to estimate the amount of code that is covered by some test case.

**Background** Testing frameworks that address these aspects have been written in a variety of languages such as Smalltalk, Java, C++ and Python.

In particular, the approach described in [[?]] has turned out to be very successful, leading – among others – to the popular JUnit library for Java [[?]], which has been ported to many other languages (see [[?]] for an extensive list of testing frameworks for all kinds of languages). Accordingly, the RUnit package is our version of porting JUnit to R, supplemented by additional functionality to inspect the test coverage of some function under question.

**Motivation** One may wonder why R would need yet another testing framework even though the standard method, namely executing *R CMD check* on ones complete package at the shell prompt, is widely accepted and applied. We think, however, that the RUnit approach is more in line with the above listed requirements and can be seen as a complement to the existing process in that:

- ^ test cases are called and executed from the R prompt
- ^ the programmer decides which result or functionality to put under testing, e.g. formatting issues of textual output do not need to matter
- ^ test and reference data files need not be maintained seperately but are combined into one file
- ^ test cases need not be limited to testing/using functionality from one package checked at a time

Moreover, testing frameworks based on JUnit ports seem to have become a quasi standard in many programming languages. Therefore, programmers new to R but familiar with other languages might appreciate a familiar testing environment. And finally, offering more than one alternative in the important field of code testing is certainly not a bad idea and could turn out useful.

Before explaining the components of the RUnit package in detail, we would like to list some of the lessons learned in the attempt of writing useful test suites for our software (a more complete collection of tips relating to a Test-First development approach can be found in [[?]]):

- ^ Develop test cases parallel to implementing your functionality. Keep testing all the time (code - test - simplify cycle). Do not wait until the software is complete and attempt to add test cases at the very end. This typically leads to crappy test cases and incomplete test cases.
- ^ Distinguish between unit and integration tests: Unit tests should be as small as possible and check one unit of functionality that can't be further decomposed. Integration tests, on the other hand, run through

a whole analysis workflow and check the interplay of various software components.

- ^ Good test coverage enables refactoring, by which a reorganisation of the implementation is meant. Without testing the attitude ‘*I better do not touch this code anymore*’ once some piece of software appears to be working is frequently encountered. It is very pleasing and time-saving just to run a test suite after some improvement or simplification of the implementation to see that all test cases are still passing (or possibly reveal some newly introduced bug). This refactoring ability is a key benefit of unit testing leading not only to better software quality but also to better design.
- ^ Do not test internal functions but just the public interface of a library. Since R does not provide very much language support for this distinction, the first step here is to clarify which functions are meant to be called by a user of a package and which are not. If internal functions are directly tested, the ability of refactoring gets lost because this typically involves reorganisation of the internal part of a library.
- ^ Once a bug has been found, add a corresponding test case.

## 2 Getting Started: Setting up test cases

### 2.1 R Unit Testing

### 2.2 R Code Inspection

#### 2.2.1 Introduction

The Code Inspector is an additional tool for checking detailed test case coverage and getting profiling information. It records how often a code line will be executed and we use that information for improving our test cases, because we can detect not executed code lines. The Code Inspector is able to handle S4 methods. During the development of the Code Inspector, we noticed, that the syntax of R is very flexible. Because our coding philosophy has an emphasis of maintenance and a clear style, we developed style guides for our R coding. Therefore, one goal for the Code Inspector was to handle our coding styles in a correct manner. This leads to the consequence that not all R expression can be handled correctly. In our implementation the Code Inspector has two main functional parts. The first part is responsible for parsing and modifying the code of the test function. The second part,

called the Tracker, holds the result of the code tracking. The result of the tracking process allows further analysis of the executed code.

### 2.2.2 Usage

The usage of the Code Inspector and the Tracker object is very simple. The following code snippet is an example:

```
> tr <- tracker()
> tr$init()
> resFoo <- inspect(foo(a))
> resBar <- inspect(bar(d))
> resTrack <- tr$getResult()
> printHTML(resTrack)
```

The first two lines creates and initialize the tracking object. The `inspect(...)` function in line three and four awaits a complete function call as argument. The `inspect` function executes and tracks the function. The results will be stored in the tracking object. The result of the function (not of the Tracker) will be returned as usual. The tracking results will received by `tr$getResult()`. With `printHTML(...)` the result of the tracking process will be presented as HTML pages.

### 2.2.3 Technical Details

The general idea for the code tracking was to modify the source code of the function. Therefore, we used the parse and deparse functions and the capability of R to generate functions on runtime. To track the function we tried to include in every code line a hook. That hook calls a function of the tracks object. The information of the tracking will be stored in the closure of the tracking object (actually a function). Because the R parser allows very nested expressions, we didn't try to modify every R expression. This is a task for the future. A simple example for the modifying process is as follow: original:

```
> foo <- function(x) {
+   y <- 0
+   for (i in 1:x) {
+     y <- y + x
+   }
+   return(y)
+ }
```

modified:

```
> foo.mod <- function(x) {  
+   tr$bp(1)  
+   y <- 0  
+   tr$bp(2)  
+   for (i in 1:x) {  
+     tr$bp(4)  
+     y <- y + x  
+   }  
+   tr$bp(6)  
+   return(y)  
+ }
```

Problematic code lines are:

```
> if (a == 1) {  
+   print("do TRUE")  
+ } else print("do FALSE")
```

This must be modified to

```
> if (a == 1) {  
+   tr$bp(2)  
+   print("do TRUE")  
+ } else {  
+   tr$bp(3)  
+   print("do FALSE")  
+ }
```

The problem is the *else* branch, that cannot be modified in the current version.

### 3 Future Development Ideas

Here we briefly list – in an unordered manner – some of the avenues for future development we or someone interested in this package could take:

- ^ extend the `checkEquals` function to handle complex S4 class objects correctly in comparisons
- ^ improve the export of text suite execution data i.e. by adding XML data export support

- ^ add some evaluation methods to the code inspector e.g. use software metrics to estimate standard measures of code quality, complexity, and performance
- ^ overcome the problem of nested calls to registered functions for code inspection
- ^ allow automatic registration of functions & methods