svUnit - A framework for unit testing in R

Philippe Grosjean (phgrosjean@sciviews.org)

Version 0.6-2, 2009-01-12

1 Introduction

Unit testing (see http://en.wikipedia.org/wiki/Unit_test) is an approach successfully used to develop software, and to ease code refactoring for keeping bugs to the minimum. It is also the insurance that the software is doing the right calculation (quality insurance). Basically, a test just checks if the code is running and is producing the correct answer/behavior in a given situation. As such, unit tests are build in R package production because all examples in documentation files, and perhaps, test code in '/tests' subdirectory are run during the checking of a package (`R CMD check <Pkg>`). However, the R approach lacks a certain number of features to allow optimal use of unit tests as in extreme programming (test first – code second):

- Tests are related to package compilation and cannot easily be run independently (for instance, for functions developed separately).
- Once a test fails, the checking process is interrupted. Thus one has to correct the bug and launch package checking again... and perhaps get caught by the next bug. It is a long and painful process.
- There is no way to select one or several tests: all are run or not (depending on command line options) during package testing.
- It is impossible to program in R in a test driven development (http://en.wikipedia.org/wiki/Test-driven_development, write tests first).
- Consequently, the 'test-code-simplify' cycle is not accessible yet to R programmer, because of the lack of an interactive and flexible testing mechanism providing immediate, or quasi immediate feedback about changes made.
- We would like also to emphasize that test suites are not only useful to check code, they can also be used to check data, or the pertinence of analyses.

1.1 Unit testing in R without svUnit

Besides the "regular" testing mechanism of R packages, one can find the **RUnit** package on CRAN (http://cran.r-project.org). Another package used to provide test unit, **butler**, but it is not maintained any more and seems to have given up in favor of **RUnit**. **RUnit** implements the following features:

- **Assertions** for tests (`checkEquals()`, `checkEqualsNumeric()`, `checkIdentical()` and `checkTrue()`) and negative tests (tests that check error conditions, `checkException()`).
- Assertions are grouped into R functions to form one **test function**. It is easy to temporarily inactivate one or more tests by commenting lines in the function. To avoid forgetting tests that are commented out later on, there is special function, named `DEACTIVATED()`, that tags the test with a reminder for your deactivated items (i.e., this is written in the test protocol).
- A series of test functions (whose name typically start with `test...`) are collected together in a sourceable R code file (name starting with `runit...`) on disk. This file is called a **test unit**.
- A **test suite** (object *RUnitTestSuite*) is a special object defining a battery of tests to run. It points to one or several directories containing test units. A test suite is defined by `defineTestSuite()`.
- One or more test suites can be run by calling `runTestSuite()`. There is a convenient shortcut to define and run a test suite constituted by only one test unit by using the function `runTestFile()`. Once the test is run, a *RUnitTestData* object is created that contains all the information collected from the various tests run.
- One can print a synthetic report (how many test units, test functions, number of errors, fails and deactivated item), or get a more extensive `summary()` of the test with indication about the tests that failed. The function `printTextProtocol()` does the same, while `printHTMLProtocol()` produces a report in HTML format.
- **RUnit** contains also functions to determine which code is run in the original function when tested, in order to detect the parts of the code not covered by the test suite (code coverage functions `inspect()` and `tracker()`).

As complete and nice as **RUnit** is, there is no tools to integrate the test suite in a given development environment (IDE) or graphical user interface (GUI). In particular, there is no real-time reporting mechanism used to easy the test-code-simplify cycle. The way tests are implemented and run is left to the user, but the implementation suggests that the authors of **RUnit** mainly target batch execution of the tests (for instance, nightly check of code in a server), rather that real-time interaction with the tests.

There is also no integration with the "regular" `R CMD check` mechanism of R in **RUnit**. There is an embryo of organization of these tests units to make them compatible with the `R CMD check` mechanism of R on the R Wiki (http://wiki.r-project.org/rwiki/doku.php?id=developers:runit). This approach works well only on Linux/Unix systems, but needs to be adapted for Windows.

1.2 Unit testing framework for R with svUnit

Our initial goal, in the context of the EU project UNCOVER, was to implement a GUI layer on top of **RUnit**, and to integrate test units as smoothly as possible

in a code editor, as well as, making tests easily accessible and fully compatible with `R CMD check` on all platforms supported by R. Ultimately, the test suite should be easy to create, to use interactively, and should be able to test functions in a complex set of R packages.

However, we encountered several difficulties while trying to enhance **RUnit** mechanism. When we started to work on this project, **RUnit** (version 0.4-17) did not allow to subclass its objects. Moreover, its *RUnitTestData* object is optimized for quick testing, but not at all for easy reviewing of its content: it is a list of lists of lists,... requiring embedded for loop and `lapply()`/`sapply()` procedures to extract its content. Finally, the concept of test units as sourceable files on disk is a nice idea, but it is too rigid for quick writing of test cases for objects not associated (yet) with R packages.

We did a first implementation of the **RUnit** GUI based on these objects, before realizing that it is really not designed for such an use. So, we decide to write a completely different unit testing framework in R: **svUnit**, but we make it test code compatible with **RUnit** (i.e., the engine and objects used are totally different, but the test code run in **RUnit** or **svUnit** can be interchangeable).

Finally, **svUnit** is also designed to be integrated in the SciViews R GUI (http://www.sciviews.org/SciViews-K), on top of Komodo Edit (http://www.activestate.com/komodo_edit), and to approach extreme programming practices with automatic code testing while you write it. A rather simple interface is provided to link and pilot **svUnit** from any GUI/IDE, and the Komodo Edit implementation could be use as example to program similar integration panels for other R GUIs. **svUnit** also formats its report with creole wiki syntax. It is directly readable, but it can also be displayed in a much nicer way using any wiki engine compatible with the creole wiki language. It is thus rather easy to write test reports in wiki servers, possibly through nightly automatic process for your code, if you like.

This vignette is a guided tour of svUnit, showing you its features and the various ways you can use it to test your R code.

2 Installation

The **svUnit** package is not available on CRAN (http://cran.r-project.org) yet, but it is available from R-Forge (http://r-forge.r-project.org). You can download it from R by:

```
> install.packages("svUnit", repos = "http://R-Forge.R-project.org")
```

This package has no dependence other than R $\geq$ 1.9.0. However, if you would like to use its interactive mode in a GUI editor, you must also install Komodo Edit and SciViews. The procedure is explaned at http://www.sciviews.org/SciViews-K.

Once the svUnit package is installed, you can check it is working well with the following example code:

```
> library(svUnit)
> Square <- function(x) return(x^2)
> test(Square) <- function() {
+     checkEquals(9, Square(3))
```

```
+     checkEquals(10, Square(3))
+     checkEquals(9, SSSquare(3))
+     checkEquals(c(1, 4, 9), Square(1:3))
+     checkException(Square("xx"))
+ }
> clearLog()
> (runTest(Square))

* : checkEquals(10, Square(3)) run in less than 0.001 sec ... **FAILS**
Mean relative difference: 0.1
 num 9
* : checkEquals(9, SSSquare(3)) run in 0.038 sec ... **ERROR**
Error in mode(current) : could not find function "SSSquare"

== test(Square) run in less than 0.1 sec: **ERROR**

//Pass: 3 Fail: 1 Errors: 1//

* : checkEquals(10, Square(3)) run in less than 0.001 sec ... **FAILS**
Mean relative difference: 0.1
 num 9

* : checkEquals(9, SSSquare(3)) run in 0.038 sec ... **ERROR**
Error in mode(current) : could not find function "SSSquare"
```

> Although test unit code is compatible with both **svUnit** and **RUnit**, do not load both packages in R memory at the same time, or you will badly mix incompatible code between them!

3 Overview of svUnit

You ensure that code you write in R functions does work as you want by defining a battery of tests that will compare the output of your code with reference values. In **svUnit**, the simplest way to define such a battery of tests is by attaching it to functions loaded in R memory[1]. Of course, you can also define batteries of tests that are independent of any R object, or that check several of them together (integration tests). Here is a couple of examples:

```
> library(svUnit)
> Square <- function(x) return(x^2)
> test(Square) <- function() {
+     checkEquals(9, Square(3))
+     checkEquals(c(1, 4, 9), Square(1:3))
+     checkException(Square("xx"))
+ }
> Cube <- function(x) return(x^3)
```

[1] In fact, you can attach **svUnit** tests to any kind of R object, not only function. This could be useful to test S3/S4 objects, or even, datasets.

```
> test(Cube) <- function() {
+     checkEquals(27, Cube(3))
+     checkEquals(c(1, 8, 28), Cube(1:3))
+     checkException(Cube("xx"))
+ }
> test_Integrate <- svTest(function() {
+     checkTrue(1 < 2, "check1")
+     v <- 1:3
+     w <- 1:3
+     checkEquals(v, w)
+ })
```

When you run a test in **svUnit**, it logs its results in a centralized logger. The idea is to get a central repository for tests that you can manipulate as you like (print, summarize, convert, search, display in a GUI, etc.). If you want to start new tests, you should first clean this logger by `cleanLog()`. At any time, the logger is accessible by `Log()`, and a summary of its containt is displayed using `summary(Log())`. So, to run test for your `Square()` function as well as your `test_Integrate` integration test, you simply do the following:

```
> clearLog()
> runTest(Square)
> runTest(test_Integrate)
> Log()

= A svUnit test suite run in less than 0.1 sec with:

* test(Square) ... OK
* test_Integrate ... OK


== test(Square) run in less than 0.1 sec: OK

//Pass: 3 Fail: 0 Errors: 0//


== test_Integrate run in less than 0.1 sec: OK

//Pass: 2 Fail: 0 Errors: 0//
```

From this report, you see that all your tests succeed. Note that **svUnit** is making the difference between a test that **fails** (the code is run correctly, but the result is different that what was expected) and code that raises **error** (it was not possible to run the test because its code is incorrect, or for some other reasons). Note also that the function `checkException()` is designed to explicitly test code that should stop() in R. So, if that test does not raises an exception, it is considered to have failed. This is useful to check that your functions correctly trap wrong arguments, for instance, like in `checkException(Square("xx"))` here above (a character string is provided where a numerical value is expected).

Now, let's look what happen if we test the `Cube()` function without clearing the logger:

```
> runTest(Cube)

* : checkEquals(c(1, 8, 28), Cube(1:3)) run in less than 0.001 sec ... **FAILS**
Mean relative difference: 0.03571429
 num [1:3] 1 8 27

> Log()

= A svUnit test suite run in less than 0.1 sec with:

* test(Square) ... OK
* test_Integrate ... OK
* test(Cube) ... **FAILS**


== test(Square) run in less than 0.1 sec: OK

//Pass: 3 Fail: 0 Errors: 0//


== test_Integrate run in less than 0.1 sec: OK

//Pass: 2 Fail: 0 Errors: 0//


== test(Cube) run in less than 0.1 sec: **FAILS**

//Pass: 2 Fail: 1 Errors: 0//

* : checkEquals(c(1, 8, 28), Cube(1:3)) run in less than 0.001 sec ... **FAILS**
Mean relative difference: 0.03571429
 num [1:3] 1 8 27
```

We note two things:

1. The results of the tests on `Cube()` are added to the previous report. So, it is possible to build rather easily reports that summarize tests on one or several (possibly) complex codes, by adding tests results in the logger. **svUnit** does this naturally and transparently. Starting a new report is equally simple: just use `clearLog()`.

2. This time, we have one test that fails. We expected `c(1, 8, 28)` from `Cube(1:3)` in `checkEquals(c(1, 8, 28), Cube(1:3))`, however, we got `c(1, 8, 29)`. Of course, in this case, the `Cube()` function is correct and it is our test that is misspecified, but that was done purposedly to show how **svUnit** outputs test failures.

3.1 Assertions in svUnit

The most basic item is an **assertion** represented by a `checkxxx()` function in **svUnit**/**RUnit**. Five such functions are currently defined:

`checkEquals(current, target)` determines if data in *target* is the same as data in *current*.

`checkEqualsNumeric(current, target)` does the same but allows for a better comparison for numbers (variation allowed within a tolerance window).

`checkIdentical(current, target)` checks whether two R objects are strictly identical.

`checkTrue(expr)` only succeed if *expr* is `TRUE`. Note a difference in **svUnit** and **RUnit** (at least, in version 0.4-17). The **RUnit** function is not vectorized and *expr* must return a single atomic logical value. The corresponding **svUnit** function also accepts a vector of logical values. In this case, all elements must be `TRUE` for the test to succeed. When you make sure that *expr* always returns a single logical value (for instance by using `all(expr)`), both functions should be compatible between the two packages.

`checkException(expr)` verifies that a given code raises an exception (in R, it means that a line of code with `stop()` is executed).

`DEACTIVATED()` both makes sure that all tests following this instruction (in a test function, see next paragraph) are deactivated, and inserts a notification in the logger that these tests are deactivated as a reminder.

For all these functions, you have an additional optional argument *msg* indicating a short message to print in front of each text in the report. These functions return invisibly: `TRUE` if the test succeeds, `FALSE` if it fails (code is executed correctly, but does not pass the test), and `NA` if there was an error (the R code was not executed correctly). Moreover, these functions record the results, the context of the test and the timing in a logger (object *svSuiteData*inheriting from *environment*) called `.Log` and located in the user's workspace. So, executing a series of assertions and getting a report is simply done as (in its simplest form, you can use the checkxxx() functions directly at the command line):

```
> clearLog()
> checkTrue(1 == log(exp(1)))
> checkException(log("a"))
> checkTrue(1 == 2)

* : checkTrue(1 == 2) run in less than 0.001 sec ... **FAILS**
 logi FALSE

> Log()

= A svUnit test suite run in less than 0.1 sec with:

* eval ... **FAILS**


== eval run in less than 0.1 sec: **FAILS**
```

```
//Pass: 2 Fail: 1 Errors: 0//

* : checkTrue(1 == 2) run in less than 0.001 sec ... **FAILS**
 logi FALSE
```

As you can see, the `checkxxx()` functions work hand in hand with the test logger. Indeed, the `checkxxx()` functions return the result of the test invisibly, which is discarded in the present case (assign the result to a variable and print it, for instance to see this result). Indeed, they are better used for their side-effect of adding an entry to the svUnit logger.

The last command `Log()` prints its content. You see how a report is printed, with a first part being a short summary by categories (assertions run at the command line are placed in the *eval* category, because there is no better context known for them. Usually, those assertions should be placed in test functions, or in test units, as we will see later in this manual). A detailed report on the tests that failed or raised an error is printed at the end of the report.

Of course, the same report is much easier to manipulate from within the graphical tree in the Komodo's R Unit tab, but this text report from R has the advantage of being independent from any GUI, and from Komodo. It can also be generated in batch mode. Last, but not least, it uses a general Wiki formatting called creole wiki (http://www.wikicreole.org/wiki/Creole1.0). Figure 1 illustrates the way the same report looks like in DokuWiki with the creole plugin (http://www.wikicreole.org/wiki/DokuWiki) installed. Note the convenient table of content that lists here a clickable and quick summary of all tests run. From this point, it is relatively easy to define nightly cron task jobs on a server to run a script that executes these tests and update a wiki page (look at your particular wiki engine documentation to determine how you can access wiki pages on the command line).

3.2 Manipulating the logger data

svUnit provides a series of methods and tools to manipulate the log file from the command line, in particular, `stats()`, `summary()`, `metadata()`, `ls()`:

```
> options(svUnit.excludeList = NULL)
> clearLog()
> runTest(svSuiteList(), name = "AllTests")

* : checkEquals(c(1, 8, 28), Cube(1:3)) run in less than 0.001 sec ... **FAILS**
Mean relative difference: 0.03571429
 num [1:3] 1 8 27
*
  runTest(bar) does not work inside test functions:  ... DEACTIVATED

> stats(Log())[, 1:3]

                                  kind timing                time
test_Integrate                      OK  0.000 2009-10-29 18:44:00
testCube                       **FAILS**  0.000 2009-10-29 18:44:00
testSquare                          OK  0.001 2009-10-29 18:44:00
```

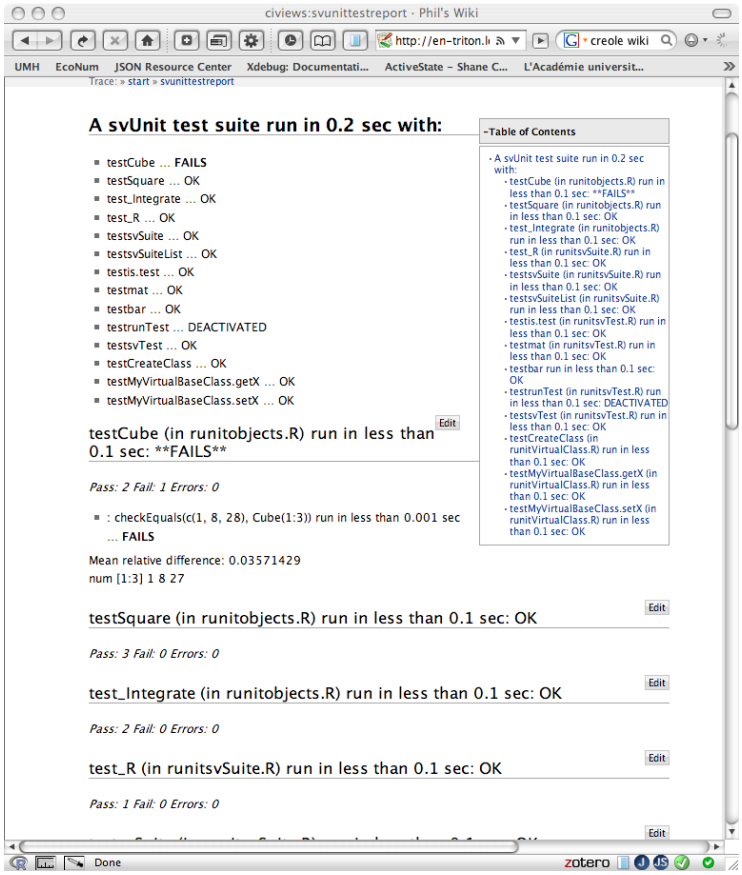

Figure 1: a svUnit test report as it appears when inserted in a wiki page (DokuWiki engine with the `creole` plugin installed). Note the summary of results at the top lef of the page, and the clickable table of contents with exhaustive and detailed entries to easily navigate to the test results you want to consult). Timing of the test is also clearly indicated, since it is a complementary but important information (if a test succeeds, but calculation is way too long, it is good to know it)!

```
test_R                                OK  0.000 2009-10-29 18:44:00
testsvSuite                           OK  0.001 2009-10-29 18:44:00
testsvSuiteList                       OK  0.023 2009-10-29 18:44:00
testis.test                           OK  0.004 2009-10-29 18:44:00
testmat                               OK  0.000 2009-10-29 18:44:00
testbar                               OK  0.001 2009-10-29 18:44:00
testrunTest                DEACTIVATED  0.004 2009-10-29 18:44:00
testsvTest                            OK  0.006 2009-10-29 18:44:00
testCreateClass                       OK  0.002 2009-10-29 18:44:00
testMyVirtualBaseClass.getX           OK  0.001 2009-10-29 18:44:00
testMyVirtualBaseClass.setX           OK  0.005 2009-10-29 18:44:00
```

```
> summary(Log())

= A svUnit test suite run in less than 0.1 sec with:

* test_Integrate ... OK
* testCube ... **FAILS**
* testSquare ... OK
* test_R ... OK
* testsvSuite ... OK
* testsvSuiteList ... OK
* testis.test ... OK
* testmat ... OK
* testbar ... OK
* testrunTest ... DEACTIVATED
* testsvTest ... OK
* testCreateClass ... OK
* testMyVirtualBaseClass.getX ... OK
* testMyVirtualBaseClass.setX ... OK


== test_Integrate (in runitAllTests.R) run in less than 0.1 sec: OK

//Pass: 2 Fail: 0 Errors: 0//


== testCube (in runitAllTests.R) run in less than 0.1 sec: **FAILS**

//Pass: 2 Fail: 1 Errors: 0//

=== Failures
[2] : checkEquals(c(1, 8, 28), Cube(1:3))


== testSquare (in runitAllTests.R) run in less than 0.1 sec: OK

//Pass: 3 Fail: 0 Errors: 0//
```

```
== test_R (in runitsvSuite.R) run in less than 0.1 sec: OK
//Pass: 1 Fail: 0 Errors: 0//

== testsvSuite (in runitsvSuite.R) run in less than 0.1 sec: OK
//Pass: 5 Fail: 0 Errors: 0//

== testsvSuiteList (in runitsvSuite.R) run in less than 0.1 sec: OK
//Pass: 6 Fail: 0 Errors: 0//

== testis.test (in runitsvTest.R) run in less than 0.1 sec: OK
//Pass: 15 Fail: 0 Errors: 0//

== testmat (in runitsvTest.R) run in less than 0.1 sec: OK
//Pass: 2 Fail: 0 Errors: 0//

== testbar run in less than 0.1 sec: OK
//Pass: 2 Fail: 0 Errors: 0//

== testrunTest (in runitsvTest.R) run in less than 0.1 sec: DEACTIVATED
//Pass: 1 Fail: 0 Errors: 0//

== testsvTest (in runitsvTest.R) run in less than 0.1 sec: OK
//Pass: 14 Fail: 0 Errors: 0//

== testCreateClass (in runitVirtualClass.R) run in less than 0.1 sec: OK
//Pass: 2 Fail: 0 Errors: 0//

== testMyVirtualBaseClass.getX (in runitVirtualClass.R) run in less than 0.1 sec: OK
//Pass: 3 Fail: 0 Errors: 0//
```

```
== testMyVirtualBaseClass.setX (in runitVirtualClass.R) run in less than 0.1 sec: OK

//Pass: 6 Fail: 0 Errors: 0//
```

```
> metadata(Log())

$.R.version
               _
platform       i386-apple-darwin8.11.1
arch           i386
os             darwin8.11.1
system         i386, darwin8.11.1
status         Patched
major          2
minor          10.0
year           2009
month          10
day            28
svn rev        50254
language       R
version.string R version 2.10.0 Patched (2009-10-28 r50254)

$.sessionInfo
R version 2.10.0 Patched (2009-10-28 r50254)
i386-apple-darwin8.11.1

locale:
[1] en_US.UTF-8/en_US.UTF-8/C/C/en_US.UTF-8/en_US.UTF-8

attached base packages:
[1] tools     stats     graphics  grDevices utils     datasets  methods
[8] base

other attached packages:
[1] svUnit_0.6-4

$.time
[1] "2009-10-29 18:44:00 CET"
```

```
> ls(Log())

 [1] "test_Integrate"              "test_R"
 [3] "testbar"                     "testCreateClass"
 [5] "testCube"                    "testis.test"
 [7] "testmat"                     "testMyVirtualBaseClass.getX"
 [9] "testMyVirtualBaseClass.setX" "testrunTest"
[11] "testSquare"                  "testsvSuite"
[13] "testsvSuiteList"             "testsvTest"
```

As you can see, `ls()` lists all components recorded in the test suite. Each component is a *svTestData* object inheriting from *data.frame*, and it can be

easily accessed through the `$` operator. There are, of course similar methods defined for those *svTestData* objects, like `print()`, `summary()`, and `stats()`:

```
> myTest <- Log()$testCube
> class(myTest)

[1] "svTestData" "data.frame"

> myTest

== testCube (in runitAllTests.R) run in less than 0.1 sec: **FAILS**

//Pass: 2 Fail: 1 Errors: 0//

* : checkEquals(c(1, 8, 28), Cube(1:3)) run in less than 0.001 sec ... **FAILS**
Mean relative difference: 0.03571429
 num [1:3] 1 8 27

> summary(myTest)

== testCube (in runitAllTests.R) run in less than 0.1 sec: **FAILS**

//Pass: 2 Fail: 1 Errors: 0//

=== Failures
[2] : checkEquals(c(1, 8, 28), Cube(1:3))

> stats(myTest)

$kind
         OK   **FAILS**   **ERROR** DEACTIVATED
          2           1           0           0

$timing
timing
     0
```

As the logger inherits from *environment*, you can manage individual test data the same way as objects in any other environment. For instance, if you want to delete a particular test data without touching to the rest, you can use:

```
> ls(Log())

 [1] "test_Integrate"              "test_R"
 [3] "testbar"                     "testCreateClass"
 [5] "testCube"                    "testis.test"
 [7] "testmat"                     "testMyVirtualBaseClass.getX"
 [9] "testMyVirtualBaseClass.setX" "testrunTest"
[11] "testSquare"                  "testsvSuite"
[13] "testsvSuiteList"             "testsvTest"

> rm(test_R, envir = Log())
> ls(Log())
```

```
 [1] "test_Integrate"               "testbar"
 [3] "testCreateClass"              "testCube"
 [5] "testis.test"                  "testmat"
 [7] "testMyVirtualBaseClass.getX" "testMyVirtualBaseClass.setX"
 [9] "testrunTest"                  "testSquare"
[11] "testsvSuite"                  "testsvSuiteList"
[13] "testsvTest"
```

As we will see in the following section, **svUnit** proposes several means to organize individual assertions in various modules within a logical organization: **test functions**, **test units** and **test suites**. This organization is inspired from **RUnit**, but with additional ways of using tests in interactive sessions (for instance, the ability to attach a test to the objects to be tested).

3.3 Test function

The first organization level for grouping assertions together is the **test function**. A test function is a function without arguments whose name must start with *test*. It typically contains a series of assertions applied to one object, method, or function to be checked (this is not obligatory, assertions are not restricted to one object, but good practices strongly suggest it). Here is an example:

```
> test_function <- function() {
+     checkTrue(1 < 2, "check1")
+     v <- 1:3
+     w <- 1:3
+     checkEquals(v, w)
+ }
> test_function <- as.svTest(test_function)
> is.svTest(test_function)

[1] TRUE
```

A test function should be made a special object called *svTest*, so that **svUnit** can recognize it. This *svTest* object, is allowed to live alone (for instance, loaded in `.GlobalEnv`, defined in a R script, etc... in **svUnit**, while in **RUnit**, it must be located in a unit test). In **svUnit** (not **RUnit**), you run this test simply by using `runTest()`, which returns the results invisibly (and you are supposed to access results from the logger):

```
> clearLog()
> runTest(test_function)
> Log()

= A svUnit test suite run in less than 0.1 sec with:

* test_function ... OK


== test_function run in less than 0.1 sec: OK

//Pass: 2 Fail: 0 Errors: 0//
```

Now, a test function is most likely designed to test an R object. **svUnit** also provides facilities to attach the test function to the object to be tested. Hence, the test cases and the tested object conveniently form a single entity that one can manipulate, copy, save, reload, etc. with all the usual tools in R. This association is simply made using `test(myobj) <-`:

```
> Square <- function(x) return(x^2)
> test(Square) <- function() {
+     checkEquals(9, Square(3))
+     checkEquals(c(1, 4, 9), Square(1:3))
+     checkException(Square("xx"))
+ }
> is.test(Square)

[1] TRUE
```

One can retrieve the test associated with the object by using:

```
> test(Square)

svUnit test function:
{
    checkEquals(9, Square(3))
    checkEquals(c(1, 4, 9), Square(1:3))
    checkException(Square("xx"))
}
```

And of course, running the test associated with an object is as easy as:

```
> runTest(Square)
> Log()

= A svUnit test suite run in less than 0.1 sec with:

* test_function ... OK
* test(Square) ... OK


== test_function run in less than 0.1 sec: OK

//Pass: 2 Fail: 0 Errors: 0//


== test(Square) run in less than 0.1 sec: OK

//Pass: 3 Fail: 0 Errors: 0//
```

Now that you master test functions, you should discover how you can group them in logical **units**, and associate them to R packages.

3.4 Test units

A **unit** is a coherent piece of software that can be tested separately from the rest. Typically, a R package is a structured way to compile and distribute such code units in R. Hence, we need a mean to organize tests related to this unit conveniently.

Since a package can contain several functions, data frames, or other objects, our unit should collect together individual test functions related to each of these objects composing our package. Also, the test unit should accommodate the well-define organization of a package, and should integrate also in the already existing testing features of R, in particular with `R CMD check`. In both **RUnit**, and **svUnit**, one can define such test units, and they are made code compatible between them (at least for version 0.4-17 of **RUnit**).

A test unit a a sourceable text file that contains one or more test functions, plus possibly `.setUp()` and `.tearDown()` functions (see the online help for further information on these special functions). In **RUnit**, you must write such test unit files from scratch. With **svUnit**, you can "promote" one or several test functions (associated to other objects, or "living" alone as separate *svTest* objects) by using the `makeUnit()` method. Here is how you promote the test associated with our `Square()` function to a simple unit test containing only one function:

```
> unit <- makeUnit(Square)
> file.show(unit, delete.file = TRUE)
```

You got the following file whose name must start with *runit*, with an *.R* extension ('runit*.R'), and located by default in the temporary environment. Specify another directory with the `dir =` argument of `makeUnit()` for a more permanent record of this test unit file. Note also that `.setUp()` and `.tearDown()` functions are constructed automatically for you. They specify the context of these tests. This context is used by the GUI in Komodo to locate the test function and the code being tested.

```
## Test unit 'Square'
.setUp <-
function () {
    ## Specific actions for svUnit: prepare context
    if ("package:svUnit" %in% search()) {
        .Log <- Log() ## Make sure .Log is created
        .Log$..Unit <- "/tmp/RtmpBoZnId/runitSquare.R"
        .Log$..File <- ""
        .Log$..Obj <- ""
        .Log$..Tag <- ""
        .Log$..Msg <- ""
        rm(..Test, envir = .Log)
    }
}
.tearDown <-
function () {
    ## Specific actions for svUnit: clean up context
    if ("package:svUnit" %in% search()) {
```

```
        .Log$..Unit <- ""
        .Log$..File <- ""
        .Log$..Obj <- ""
        .Log$..Tag <- ""
        .Log$..Msg <- ""
        rm(..Test, envir = .Log)
    }
}
"testSquare" <-
function() {
    checkEquals(9, Square(3))
    checkEquals(c(1, 4, 9), Square(1:3))
    checkException(Square("xx"))
}
```

Compatibility of these test unit files between **RUnit** and **svUnit** was a major concern in the design of **svUnit**. Consequently, code specific to **svUnit** (for managing the context of the test) is embedded in a `if ("package:svUnit" %in% search())`. That way, if **svUnit** is not loaded in memory, this code is not executed. *Note that you should avoid loading in memory both **svUnit** and **RUnit** at the same time. If you do so, you will most likely crash your tests.*

You will see further that it is possible to write much more complex test units with the same `makeUnit()` method, starting from test suites. But for the moment, let's discuss a little bit how such test units should be organized in R package.

If you intend to associate test units to R package, you should respect the following conventions:

- Name your test units 'runit*.R'.
- Place them in the '/inst/unitTests' subdirectory of the package sources, or in one of its subdirectories. If you place them in a subdirectory of '/inst/unitTests', then you define secondary unit tests, for instance, for more optional detailed testing of the package. Always keep in mind that all 'runit*.R' files in a directory will be run one after the other. So, if you want to make subgroups you would like to dissociate, define subdirectories.
- When the package will be compiled, all these test units will be located in '/unitTests'.

If you respect these conventions, **svUnit** knows where package unit tests are located and will be able to find and run them quite easily. See, for instance, the examples in the **svUnit** package.

So, with test units associated to packages, you have a very convenient way to run these tests, including from the Komodo R Unit tab panel. With just a little bit more coding you can also include these test units in the `R CMD check` process of your packages. You do that by means of examples (we prefer to use **examples**, instead of '/tests' in the `R CMD check` process, because examples offer a more flexible way to run tests and you can also run them in interactive sessions through the `example()` function, which is not the case for '/tests'). Here is what you do to associate some or all of your unit tests to `R CMD check` (illustrated with the **svUnit** example):

- Define a '.Rd' file in '/man' called 'unitTests.Rd' (or whatever name you prefer).
- Fill the '.Rd' file, making sure that you define an alias as *unitTests.myPKG*. Also place a little bit of information telling how users can run your test in an interactive session.
- The important part of this file is, of course, the `\examples{}` section. You must first clear the log, then run each test, and then, call the `errorLog()` function. That function looks if one or more tests failed or raised an error. In this case, it stops execution of the example and causes a dump in `R CMD check` with a report on the tests that failed. That way, providing that you have the **svUnit** package installed in the machine where you run `R CMD check`, your test units will be included nicely in the checking process of your packages, that is, they will run silently each time you test your package if no error occurs, but will produce a detailed report in case of problems.
- Here is how your '.Rd' file should looks like:

```
\name{unitTests}
\alias{unitTests.svUnit}
\title{ Unit tests for the package svUnit }
\description{ Performs unit tests defined in this
  package by running \code{example(unitTests.svUnit)}.
  Tests are in \code{runit*.R} files Located in the
  '/unitTests' subdirectory or one of its
  subdirectories ('/inst/unitTests' and subdirectories
  in package sources).
}
\author{Philippe Grosjean
  (\email{phgrosjean@sciviews.org})}
\examples{
library(svUnit)
# Make sure to clear log of errors and failures first
clearLog()
# Run all test units defined in 'svUnit' package
(runTest(svSuite("package:svUnit"), "svUnit"))
\donttest{
# These tests are not run in R CMD check but with
# example(unitTests.svUnit)
# Tests to run with example() but not with R CMD check
# Run all test units defined in
# the /unitTests/VirtualClass subdir
(runTest(svSuite("package:svUnit (VirtualClass)"),
  "VirtualClass"))
}
# Check errors at the end of the process
#(needed to interrupt R CMD check)
errorLog()
}
```

```
\keyword{utilities}
```

Also, you provide a very convenient and easy way to test a package from the command line in an interactive session by running:

```
> example(unitTests.svUnit)

untT.U> library(svUnit)

untT.U> # Make sure to clear log of errors and failures first
untT.U> clearLog()

untT.U> # Run all test units defined in the 'svUnit' package
untT.U> (runTest(svSuite("package:svUnit"), "svUnit"))
*
  runTest(bar) does not work inside test functions:  ... DEACTIVATED

[1] "/tmp/Rinst278226573/svUnit/unitTests/runitsvSuite.R"
[2] "/tmp/Rinst278226573/svUnit/unitTests/runitsvTest.R"

untT.U> ## No test:
untT.U> # Tests to run with example() but not with R CMD check
untT.U> # Run all test units defined in the /unitTests/VirtualClass subdir of 'svUnit'
untT.U> (runTest(svSuite("package:svUnit (VirtualClass)"), "VirtualClass"))
[1] "/tmp/Rinst278226573/svUnit/unitTests/VirtualClass/runitVirtualClass.R"

untT.U> ## End(No test)
untT.U>
untT.U> ## Not run:
untT.U> ##D # Tests to present in ?unitTests.svUnit but to never run automatically
untT.U> ##D # Run all currently loaded test cases and test suites of all loaded packages
untT.U> ##D (runTest(svSuiteList(), "AllTests"))
untT.U> ## End(Not run)
untT.U>
untT.U> ## Don't show:
untT.U> # Put here test units you want to run during R CMD check but don't want to show
untT.U> # or run with example(unitTests.svUnit)
untT.U> ## End Don't show
untT.U>
untT.U> # Check errors at the end of the process (needed to interrupt R CMD check)
untT.U> errorLog()
```

3.5 Test suites: collections of test functions and units

The highest level of organization of your tests is the **test suite**. A test suite is an unordered collection of test functions and test units. You can select test units associated with R package in a very convenient way: just specify `package:myPkg` and all test units in the '/unitTests' subdirectory of you package will be included (**svUnit** does all the required work to map these to actual directories where the test unit files are located). Also, if you specify `package:mypkg (subgroup)`, you will include the test units defined in '/unitTests/subgroup'. Of course, you

will be able to add also test units defined in custom directories, outside of R packages (for instance for integration or harness tests that check cross-packages, or multi-packages features of your application).

Test functions associated to your test suite receive a special treatment. Unlike `runTest()` applied to a single test function, or to an object that has an associated test function, these tests are not run from the version loaded in memory. Instead, they are first collected together in a unit test file on disk (located in the R temporary directory, by default), and run from there. Hence, building a more complex unit test file by collecting together several test functions is just a question of constructing a test suite, and then, applying the `makeUnit()` method to this *svSuite* object.

Before we apply all this, you should know an additional function: `svSuiteList()`. This function lists all test units and test functions available in your system at a given time. So, you don't need to manually create lists of components. You are better to list them automatically. Of course, this function has a lot of arguments for listing only test units in packages, only test functions, specifying where (in which environment) the test functions are located, adding custom directories where to look for unit tests, etc, etc. See the online help of this function for the description of all its arguments. One argument is particularly important: `excludeList =`. This argument defines one or several regular expressions that are used as filters to hide items from the list. This is required, since you will certainly not want to run again and again, let's say, the example tests associated with the **svUnit** package (**svUnit** must be loaded in memory to run the tests, so its tests examples risk to be listed all the time,... unless you define an adequate filter expression that will exclude them from your list)! As the default argument suggests it, the regular expression for list exclusion should be kept in the *svUnit.excludeList* R option. Here is how it works:

```
> options(svUnit.excludeList = c("package:sv", "package:Runit"))
> svSuiteList()

A svUnit test suite definition with:

- Test functions:
[1] "test_function"  "test_Integrate" "test(Cube)"     "test(Square)"
```

Thus, every entry matching the regular expressions `package:sv` and `package:RUnit` are currently excluded. The entries `package:svUnit` and `package:svUnit (VirtualClass)` match first pattern and are thus excluded. Now, let's clear the exclusion list to see what happens:

```
> options(svUnit.excludeList = NULL)
> svSuiteList()

A svUnit test suite definition with:

- Test suites:
[1] "package:svUnit"                "package:svUnit (VirtualClass)"

- Test functions:
[1] "test_function"  "test_Integrate" "test(Cube)"     "test(Square)"
```

These are test groups of test units associated with the package **svUnit**. Now, you have noticed that svSuiteList() can also find *svTest* objects, as well as tests attached to objects in the user's workspace. You can create a suite by collecting all these items like this:

```
> (mySuite <- svSuiteList())

A svUnit test suite definition with:

- Test suites:
[1] "package:svUnit"                "package:svUnit (VirtualClass)"

- Test functions:
[1] "test_function"  "test_Integrate" "test(Cube)"     "test(Square)"
```

Now let's make a more complex test unit using test functions collected in this suite:

```
> myUnit <- makeUnit(mySuite, name = "ExampleTests")
> file.show(myUnit, delete.file = TRUE)
```

This produces a file named 'runitExampleTests.R' located (by default) in the R temporary directory, and which contains all tests currently residing in the user's workspace (either as *svTest* objects, or as tests attached to other objects), plus tests suites in packages that are **not** in the exclusion list. Running tests in your suite is also very simple. Always remember the `runTest()` method, and the management of the logger:

```
> clearLog()
> runTest(mySuite)

* : checkEquals(c(1, 8, 28), Cube(1:3)) run in less than 0.001 sec ... **FAILS**
Mean relative difference: 0.03571429
 num [1:3] 1 8 27
*
  runTest(bar) does not work inside test functions:  ... DEACTIVATED

> summary(Log())

= A svUnit test suite run in less than 0.1 sec with:

* test_function ... OK
* test_Integrate ... OK
* testCube ... **FAILS**
* testSquare ... OK
* test_R ... OK
* testsvSuite ... OK
* testsvSuiteList ... OK
* testis.test ... OK
* testmat ... OK
* testbar ... OK
* testrunTest ... DEACTIVATED
```

```
* testsvTest ... OK
* testCreateClass ... OK
* testMyVirtualBaseClass.getX ... OK
* testMyVirtualBaseClass.setX ... OK


== test_function (in runitmySuite.R) run in less than 0.1 sec: OK

//Pass: 2 Fail: 0 Errors: 0//


== test_Integrate (in runitmySuite.R) run in less than 0.1 sec: OK

//Pass: 2 Fail: 0 Errors: 0//


== testCube (in runitmySuite.R) run in less than 0.1 sec: **FAILS**

//Pass: 2 Fail: 1 Errors: 0//

=== Failures
[2] : checkEquals(c(1, 8, 28), Cube(1:3))


== testSquare (in runitmySuite.R) run in less than 0.1 sec: OK

//Pass: 3 Fail: 0 Errors: 0//


== test_R (in runitsvSuite.R) run in less than 0.1 sec: OK

//Pass: 1 Fail: 0 Errors: 0//


== testsvSuite (in runitsvSuite.R) run in less than 0.1 sec: OK

//Pass: 5 Fail: 0 Errors: 0//


== testsvSuiteList (in runitsvSuite.R) run in less than 0.1 sec: OK

//Pass: 6 Fail: 0 Errors: 0//


== testis.test (in runitsvTest.R) run in less than 0.1 sec: OK

//Pass: 15 Fail: 0 Errors: 0//


== testmat (in runitsvTest.R) run in less than 0.1 sec: OK
```

```
//Pass: 2 Fail: 0 Errors: 0//


== testbar run in less than 0.1 sec: OK

//Pass: 2 Fail: 0 Errors: 0//


== testrunTest (in runitsvTest.R) run in less than 0.1 sec: DEACTIVATED

//Pass: 1 Fail: 0 Errors: 0//


== testsvTest (in runitsvTest.R) run in less than 0.1 sec: OK

//Pass: 14 Fail: 0 Errors: 0//


== testCreateClass (in runitVirtualClass.R) run in less than 0.1 sec: OK

//Pass: 2 Fail: 0 Errors: 0//


== testMyVirtualBaseClass.getX (in runitVirtualClass.R) run in less than 0.1 sec: OK

//Pass: 3 Fail: 0 Errors: 0//


== testMyVirtualBaseClass.setX (in runitVirtualClass.R) run in less than 0.1 sec: OK

//Pass: 6 Fail: 0 Errors: 0//
```

There are many other tools to manipulate *svSuite* objects in the **svUnit** package, including functions to define their content completely manually. Look at the online help of respective functions for more info.

4 Using svUnit with Komodo/SciViews

In the case you use the Komodo GUI (see http://www.sciviews.org/SciViews-K), you can integrate **svUnit** tests in this IDE and display reports in a convenient hierarchical tree presentation (Fig. 2). If R is started from Komodo with the SciViews-K plugin installed, then, starting **svUnit** from R automatically installs the R Unit side panel in Komodo. Its use should be straightforward:

- Select the tests you want to run in the top part,
- Click the Run button each time you want to refresh the test tree,
- Browse the tree for failures or errors (the color at the top of the panel immediately indicates if there is a problem somewhere: green -> everything

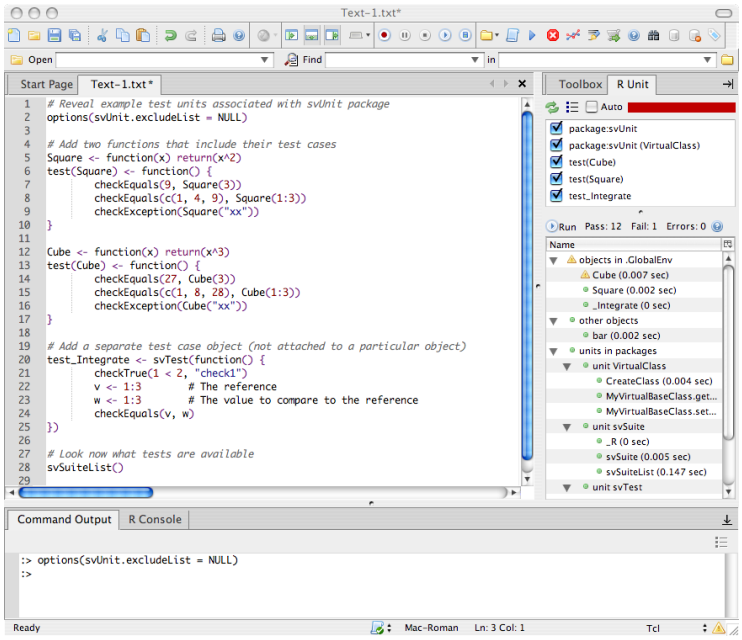

Figure 2: Komodo Edit with SciViews-K and SciViews-K Unit after running tests. At right: the R Unit panel that display (at the top) the list of available tests, and test units where you can select the one to run, and at the bottom, a tree with the results from last tests run. The stripe at the very top is green if all tests succeed, and red (as here), if at least one tests failed or raised an error.

is fine, red -> there is a problem).

- If you have failures or errors, move the mouse on top of the corresponding item in the tree, and you got more information in a tooltip,
- Click on an item to open the test unit at that place in a buffer in Komodo.

The auto mode, when activated, sources R files currently edited in Komodo whenever you save them, and then, refreshes the test report tree. This mode allows you to run automatically your tests on the background while you type your code!

> Make sure also to look at the `koUnit_xxx()` functions in the **svUnit** package. These functions allow to control the GUI in Komodo remotely from within R and R code.

References

[1] Grosjean, Ph., 2003. SciViews: an object-oriented abstraction layer to design GUIs on top of various calculation kernels [online: http://www.ci.tuwien.ac.at/Conferences/DSC-2003]

[2] IEEE Standards Boards, 1993. IEEE standard for software unit testing. ANSI/IEEE Std 1008-1987. 24 pp.

[3] Ihaka R. & R. Gentleman, 1996. R: a language for data analysis and graphics. *J. Comput. Graphic. Stat.*, **5**:299-314.

[4] Jeffries, R., 2006. Extreme programming, web site at: http://www.xprogramming.com.

[5] KÃ¶nig, T., K. JÃŒnemann & M. Burger, 2007. RUnit – A unit test framework for R. Vignette of the package RUnit available on CRAN. 11 pp.

[6] R Development Core Team, 2008. R: A language and environment for statistical computing. [online: http://www.R-project.org].