svUnit - A framework for unit testing in R

Philippe Grosjean (phgrosjean@sciviews.org)

Version 0.7-3, 2010-09-05

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 choose one or several tests selectively: all are run or not (depending on command line options) during package checking.
- It is very hard, or near to impossible to program in R in a test driven development (*write tests first*) with the standard tools (http://en.wikipedia.org/wiki/Test-driven_development).
- Consequently, the '*test-code-simplify*' cycle is not easily 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 an alternate implementation of test unit: **butler** , but it is not maintained any more and has given up in favor of **RUnit**. **RUnit** implements the following features:

- **Assertions,** `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** that runs a series of related individual tests. 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., the reminder is written in the test log).

- 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 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 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 tests with indication about which ones failed or produced errors. 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 function `inspect()` and function `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), as far as we know. 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 `R CMD check` 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 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 some content. Finally, the concept of test units as sourceable files on disk is a nice idea, but it is too rigid for quick writing test cases for objects not associated (yet) with an 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** is interchangeable).

Finally, **svUnit** is also designed to be integrated in the SciViews GUI (http://www.sciviews.org/SciViews-K), on top of Komodo Edit or IDE (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/IDE implementation could be used as an 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 its features and the various ways you can use it to test your R code.

2 Installation

The **svUnit** package is available on CRAN (http://cran.r-project.org), and its latest development version is also available on R-Forge (http://sciviews.r-forge.r-project.org/). You can install it with[1]:

```
> install.packages("svUnit")
```

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 or Komodo IDE and SciViews-K. The procedure is explaned here: http://www.sciviews.org/SciViews-K.

Once the **svUnit** package is installed, you can check it is working correctly on your machine with the following example code:

```
> library(svUnit)
> Square <- function (x) return(x^2)
> test(Square) <- function () {
+     checkEqualsNumeric(9, Square(3))
+     checkEqualsNumeric(10, Square(3))   # This intentionally fails
+     checkEqualsNumeric(9, SSSquare(3))  # This raises error
+     checkEqualsNumeric(c(1, 4, 9), Square(1:3))
+     checkException(Square("xx"))
+ }
> clearLog()
> (runTest(Square))

* : checkEqualsNumeric(10, Square(3)) run in 0.002 sec ... **FAILS**
Mean relative difference: 0.1
 num 9
* : checkEqualsNumeric(9, SSSquare(3)) run in less than 0.001 sec ... **ERROR**
Error in as.vector(current) : could not find function "SSSquare"

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

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

* : checkEqualsNumeric(10, Square(3)) run in 0.002 sec ... **FAILS**
Mean relative difference: 0.1
 num 9

* : checkEqualsNumeric(9, SSSquare(3)) run in less than 0.001 sec ... **ERROR**
Error in as.vector(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!

[1] Install the development version with `install.packages("svUnit", repos = "http://r-forge.r-project.org")`.

3 Overview of svUnit

You ensure that code you write in your R functions does the expected work 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[2]. Of course, you can also define batteries of tests that are independent of any R object, or that check several of them together (so called, *integration tests*). Here is a couple of examples:

```
> library(svUnit)
> ## Create two R functions that include their own test cases
> Square <- function (x) return(x^2)
> test(Square) <- function () {
+     checkEqualsNumeric(9, Square(3))
+     checkEqualsNumeric(c(4, 9), Square(2:3))
+     checkException(Square("xx"))
+ }
> Cube <- function (x) return(x^3)
> test(Cube) <- function () {
+     checkEqualsNumeric(27, Cube(3))
+     checkEqualsNumeric(c(8, 28), Cube(2:3))
+     checkException(Cube("xx"))
+ }
> ## Add a separate test case
> test_Integrate <- svTest(function () {
+     checkTrue(1 < 2, "check1")
+     v <- c(1, 2, 3)  # The reference
+     w <- 1:3         # The value to compare to the reference
+     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 `clearLog()`. 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//
```

[2] 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_Integrate run in less than 0.1 sec: OK

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

In this report, you see that all your tests succeed. Note that **svUnit** is making the distinction between a test that **fails** (the code is run correctly, but the result is different from 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[3]. 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 happens if we test the `Cube()` function without clearing the logger:

```
> runTest(Cube)

* : checkEqualsNumeric(c(8, 28), Cube(2:3)) run in less than 0.001 sec ... **FAILS**
Mean relative difference: 0.03571429
 num [1:2] 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//

* : checkEqualsNumeric(c(8, 28), Cube(2:3)) run in less than 0.001 sec ... **FAILS**
Mean relative difference: 0.03571429
 num [1:2] 8 27
```

We note this:

[3] `checkException()` can also track `warning()` messages with this little trick: first convert all warnings into errors with `owarn <- options(warn = 2)$warn`, run the code that should generate a warning inside `checkException()`, and then restore default warning behavior with `options(warn = owarn)`.

1. When a test succeeds, nothing is printed by default (result is returned invisibly). But when a test fails, or raises errors, the guilty test(s) results are printed. We expected `c(8, 28)` (made intentionally wrong for the sake of the demonstration) from `Cube(2:3)` in `checkEqualsNumeric(c(8, 28), Cube(2:3))`, and (of course), we got `c(8, 27)`. This test shows how **svUnit** presents test failures.

2. 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 several objects, by adding test results in the logger sequentially. **svUnit** does this naturally and transparently. Starting a new report is equally simple: just use `clearLog()`...

3.1 Assertions in svUnit

The most basic item in a test suite 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 its 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 of the vector 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.

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

DEACTIVATED() makes sure that all tests following this instruction (in a test function, see next paragraph) are deactivated, and inserts a reminder in the logger about the fact that some tests are deactivated in this suite.

For all these functions, you have an additional optional argument `msg` where you can provide a (short) message to print in front of each text in the report. These functions return invisibly: `TRUE` if the test succeeds, or `FALSE` if it fails (code is executed correctly, but does not pass the test), and `NA` if there was an error (the R code of the test 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 various `checkXXX()` functions directly at the command line):

```
> clearLog()
> checkEqualsNumeric(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 (the `checkXXX()` functions also return the result of the test invisibly, so, you can also assign it to a variable if you like). These function are mainly used for their side-effect of adding an entry to the logger.

The last command `Log()` prints the content of the logger. 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 automatically in the *eval* category: 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, and the category will reflect this organization). A detailed report on the tests that failed or raised an error is also 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 the simple text report in 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 list 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

The **svUnit** package provides a series of functions to manipulate the logger from the command line, in particular, `stats()`, `summary()`, `metadata()` and `ls()`:

```
> options(svUnit.excludeList = NULL)
> ## Clear the logger
> clearLog()
> ## Run all currently defined tests
> runTest(svSuiteList(), name = "AllTests")

* : checkEqualsNumeric(c(8, 28), Cube(2:3)) run in less than 0.001 sec ... **FAILS**
Mean relative difference: 0.03571429
 num [1:2] 8 27
*
  runTest(bar) does not work inside test functions:  ... DEACTIVATED

> ## Get some statistics
> stats(Log())[, 1:3]

                                kind timing                time
testCube                   **FAILS**  0.002 2011-03-15 11:35:52
testSquare                        OK  0.001 2011-03-15 11:35:52
```

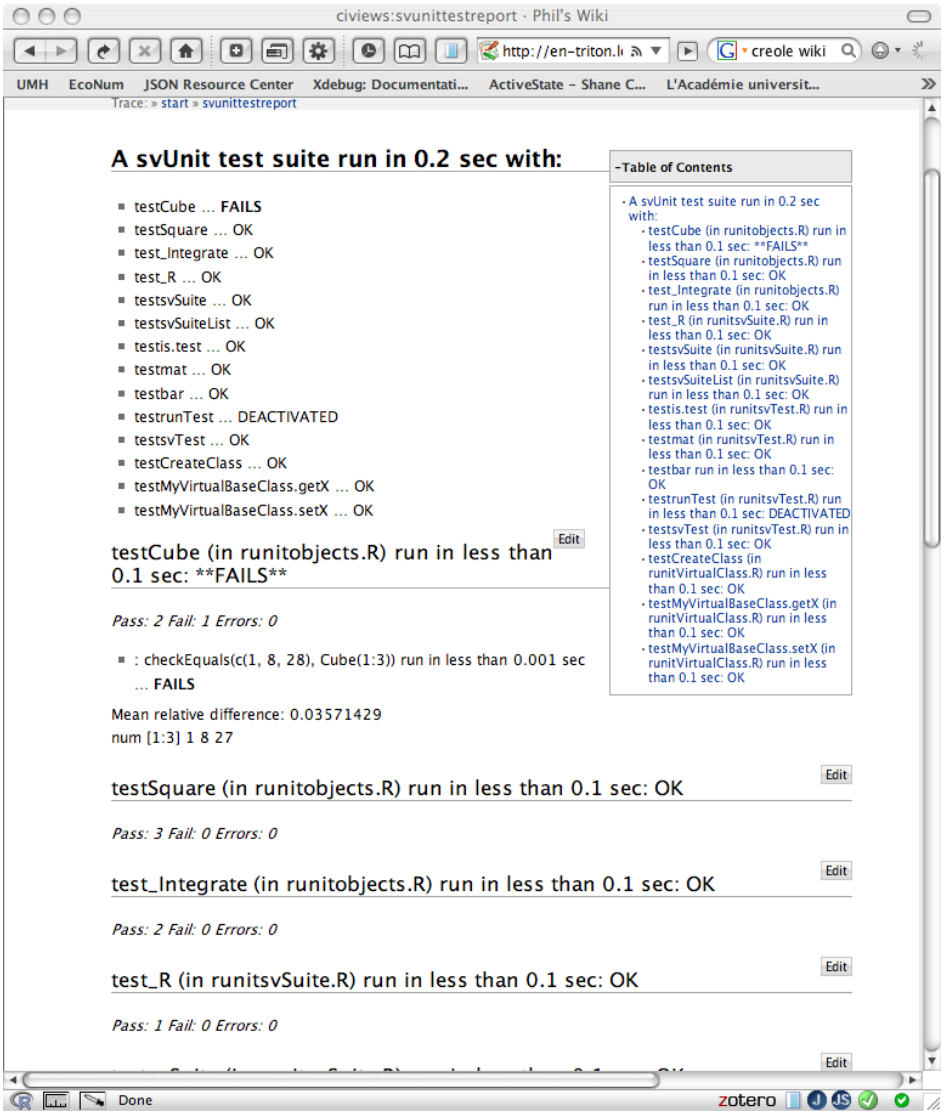

Figure 1: a **svUnit** test report as it appears when inserted in a wiki page (DokuWiki engine with the creole plugin installed). Note the summary of results at the top left of the page, and the clickable table of contents with detailed entries to easily navigate to the test log you are consulting). 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_Integrate                          OK  0.000 2011-03-15 11:35:52
test_R                                  OK  0.000 2011-03-15 11:35:53
testsvSuite                             OK  0.004 2011-03-15 11:35:53
testsvSuiteList                         OK  0.043 2011-03-15 11:35:53
testis.test                             OK  0.003 2011-03-15 11:35:53
testmat                                 OK  0.001 2011-03-15 11:35:53
testbar                                 OK  0.001 2011-03-15 11:35:53
testrunTest                 DEACTIVATED  0.005 2011-03-15 11:35:53
testsvTest                              OK  0.002 2011-03-15 11:35:53
testCreateClass                         OK  0.002 2011-03-15 11:35:53
testMyVirtualBaseClass.getX             OK  0.001 2011-03-15 11:35:53
testMyVirtualBaseClass.setX             OK  0.006 2011-03-15 11:35:53

> ## A slightly different presentation than with print
> summary(Log())

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

* testCube ... **FAILS**
* testSquare ... OK
* test_Integrate ... 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


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

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

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


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

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


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

//Pass: 2 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 collected on the machine where tests are run
> metadata(Log())

$.R.version
               _
platform       i486-pc-linux-gnu
arch           i486
os             linux-gnu
system         i486, linux-gnu
status
major          2
minor          12.0
year           2010
month          10
day            15
svn rev        53317
language       R
version.string R version 2.12.0 (2010-10-15)

$.sessionInfo
R version 2.12.0 (2010-10-15)
Platform: i486-pc-linux-gnu (32-bit)

locale:
[1] C

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

other attached packages:
[1] svUnit_0.7-5

loaded via a namespace (and not attached):
[1] tools_2.12.0

$.time
[1] "2011-03-15 11:35:52 CET"

> ## List content of the log
> ls(Log())

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

* : checkEqualsNumeric(c(8, 28), Cube(2:3)) run in less than 0.001 sec ... **FAILS**
Mean relative difference: 0.03571429
 num [1:2] 8 27

> summary(myTest)

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

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

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

> stats(myTest)

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

$timing
timing
 0.002
```

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

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

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

As we will see in the following section, **svUnit** proposes several means to organize individual assertions in modules: **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 such a restriction). Here is an example:

```
> test_function <- function () {
+     checkTrue(1 < 2, "check1")
+     v <- c(1, 2, 3)  # The reference
+     w <- 1:3         # The object to compare to the reference
+     checkEqualsNumeric(v, w)
+ }
> ## Turn this function into a test
> 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 on its own (for instance, in the user's workspace, or anywhere you like). It can be defined in a R script, be saved in a '.RData' file, etc... Note that this is very different from **RUnit** where test must always be located in a unit test file on disk). In **svUnit** (not **RUnit**), you run such a test simply by using `runTest()`, which returns the results invisibly and add it to 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. The **svUnit** package 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)
> ## A test case to associate with the Square() function
> test(Square) <- function () {
+     checkEqualsNumeric(9, Square(3))
```

```
+     checkEqualsNumeric(c(1, 4, 9), Square(1:3))
+     checkException(Square("xx"))
+ }
> is.test(Square)  # Does this object contain tests?

[1] TRUE
```

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

```
> test(Square)

svUnit test function:
{
    checkEqualsNumeric(9, Square(3))
    checkEqualsNumeric(c(1, 4, 9), Square(1:3))
    checkException(Square("xx"))
}
attr(,"srcfile")
svUnit.Rnw
attr(,"wholeSrcref")
%% LyX 1.6.5 created this file.  For more info, see http://www.lyx.org/.
%% Do not edit unless you really know what you are doing.
\documentclass[twoside,english]{article}
\usepackage{mathpazo}
\renewcommand{\sfdefault}{lmss}
\renewcommand{\ttdefault}{lmtt}
\usepackage[T1]{fontenc}
\usepackage[utf8]{inputenc}
\usepackage{listings}
\usepackage[a4paper]{geometry}
\geometry{verbose,tmargin=3cm,bmargin=4.5cm,lmargin=2.5cm,rmargin=3cm}
\usepackage{fancyhdr}
\pagestyle{fancy}
\usepackage{color}
\usepackage{babel}

\usepackage{framed}
\usepackage{url}
\usepackage{graphicx}
\definecolor{shadecolor}{rgb}{0.7,0.7,0.7}
\usepackage[unicode=true,
 bookmarks=true,bookmarksnumbered=false,bookmarksopen=false,
 breaklinks=true,pdfborder={0 0 1},backref=false,colorlinks=true]
 {hyperref}
\hypersetup{pdftitle={svUnit - A framework for unit testing in R},
 pdfauthor={Philippe Grosjean},
 pdfsubject={Test units using the svUnit R package},
 pdfkeywords={test unit, extreme programming, code refactoring, quality insurance}}

\makeatletter

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% LyX specific LaTeX commands.

\providecommand{\R}{\textsf{R}}
```

```
\providecommand{\file}[1]{`\textsf{#1}'}

\providecommand{\command}{\bgroup\@noligs\@cmdx}
\def\@cmdx#1{{\normalfont\texttt{#1}}\egroup}

\providecommand{\pkg}[1]{{\fontseries{b}\selectfont #1}}

{\catcode`\ =\active\global\def\code{\bgroup\@noligs\catcode`\ =\active\let \codespace\@codex}}
\def\codespace{{ }}
\def\@codex#1{\normalfont\texttt{#1}\egroup}

\providecommand{\var}[1]{{\normalfont\textsl{#1}}}

\providecommand{\SciViews}[1]{\textsf{\textsl{S}\lower.08em\hbox{ci}\textsl{\kern-.04emV}\kern-.02em\

\providecommand{\menuitem}[1]{\textsf{\textsc{#1}}}


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Textclass specific LaTeX commands.
\providecommand{\Prob}{\mathop{\mathrm{P}}}
\providecommand{\E}{\mathop{\mathrm{E}}}
\providecommand{\VAR}{\mathop{\mathrm{var}}}
\providecommand{\COV}{\mathop{\mathrm{cov}}}
\providecommand{\COR}{\mathop{\mathrm{cor}}}
\newenvironment{lyxlist}[1]
{\begin{list}{}
{\settowidth{\labelwidth}{#1}
 \setlength{\leftmargin}{\labelwidth}
 \addtolength{\leftmargin}{\labelsep}
 \renewcommand{\makelabel}[1]{##1\hfil}}}
{\end{list}}
\newenvironment{lyxcode}
{\par\begin{list}{}{
\setlength{\rightmargin}{\leftmargin}
\setlength{\listparindent}{0pt}% needed for AMS classes
\raggedright
\setlength{\itemsep}{0pt}
\setlength{\parsep}{0pt}
\normalfont\ttfamily}%
 \item[]}
{\end{list}}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% User specified LaTeX commands.
% \VignetteIndexEntry{svUnit - A framework for unit testing in R}
%\VignettePackage{svUnit}

\makeatother

\begin{document}

\title{svUnit - A framework for unit testing in R }
```

```
\date{Version 0.7-3, 2010-09-05}


\author{Philippe Grosjean (phgrosjean@sciviews.org)}

\maketitle

\section{Introduction}

Unit testing (see \url{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 \file{/tests} subdirectory are run
during the checking of a package (\command{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):
\begin{itemize}
\item Tests are related to package compilation and cannot easily be run
independently (for instance, for functions developed separately).
\item 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.
\item There is no way to choose one or several tests selectively: all are
run or not (depending on command line options) during package checking.
\item It is very hard, or near to impossible to program in \R{} in a test
driven development (\emph{write tests first}) with the standard tools
(\url{http://en.wikipedia.org/wiki/Test- driven_development}).
\item Consequently, the {}`\emph{test-code-simplify}' cycle is not easily
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.
\item 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.
\end{itemize}

\subsection{Unit testing in R without svUnit}

Besides the \textquotedbl{}regular\textquotedbl{} testing mechanism
of \R{} packages, one can find the \pkg{RUnit} package on CRAN (\url{http://cran.r-project.org}).
Another package used to provide an alternate implementation of test
unit: \pkg{butler} , but it is not maintained any more and has given
up in favor of \pkg{RUnit}. \pkg{RUnit} implements the following
features:
\begin{itemize}
\item \textbf{Assertions,} \code{checkEquals()}, \code{checkEqualsNumeric()},
```

```
\code{checkIdentical()} and \code{checkTrue()} and negative tests
(tests that check error conditions, \code{checkException()}).
\item Assertions are grouped into \R{} functions to form one \textbf{test
function} that runs a series of related individual tests. 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 \code{DEACTIVATED()} that tags
the test with a reminder for your deactivated items (i.e., the reminder
is written in the test log).
\item A series of test functions (whose name typically start with \code{test....})
are collected together in a sourceable \R{} code file (name starting
with \code{runit....} ) on disk. This file is called a \textbf{test
unit}.
\item A \textbf{test suite} (object \var{RUnitTestSuite}) is a special
object defining a battery of tests It points to one or several directories
containing test units. A test suite is defined by \code{defineTestSuite()}.
\item One or more test suites can be run by calling \code{runTestSuite()}.
There is a shortcut to define and run a test suite constituted by
only one test unit by using the function \code{runTestFile()}. Once
the test is run, a \var{RUnitTestData} object is created that contains
all the information collected from the various tests run.
\item One can print a synthetic report (how many test units, test functions,
number of errors, fails and deactivated item), or get a more extensive
\code{summary()} of the tests with indication about which ones failed
or produced errors. The function \code{printTextProtocol()} does
the same, while \code{printHTMLProtocol()} produces a report in HTML
format.
\item \pkg{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 function
\code{inspect()} and function \code{tracker()}).
\end{itemize}
As complete and nice as \pkg{RUnit} is, there is no tools to integrate
the test suite in a given development environment (IDE) or graphical
user interface (GUI), as far as we know. In particular, there is no
real-time reporting mechanism used to easy the \emph{test-code-simplify}
cycle. The way tests are implemented and run is left to the user,
but the implementation suggests that the authors of \pkg{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 \textquotedbl{}regular\textquotedbl{}
\command{R CMD check} mechanism of \R{} in \pkg{RUnit}. There is
an embryo of organization of these tests units to make them compatible
with \command{R CMD check} on the R Wiki (\url{http://wiki.r-project.org/rwiki/doku.php?id=developers
This approach works well only on Linux/Unix systems, but needs to
be adapted for Windows.


\subsection{Unit testing framework for R with svUnit}

Our initial goal was to implement a GUI layer on top of \pkg{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
\command{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
\pkg{RUnit} mechanism. When we started to work on this project, \pkg{RUnit}
(version 0.4-17) did not allow to subclass its objects. Moreover,
its \var{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 \code{lapply()} /\code{sapply()}
procedures to extract some content. Finally, the concept of test units
as sourceable files on disk is a nice idea, but it is too rigid for
quick writing test cases for objects not associated (yet) with an
\R{} packages.

We did a first implementation of the \pkg{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{} : \pkg{svUnit}, but we make it test code compatible
with \pkg{RUnit} (i.e., the engine and objects used are totally different,
but the test code run in \pkg{RUnit} or \pkg{svUnit} is interchangeable).

Finally, \pkg{svUnit} is also designed to be integrated in the \SciViews{}
GUI (\url{http://www.sciviews.org/SciViews-K}), on top of Komodo
Edit or IDE (\url{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 \pkg{svUnit} from any GUI/IDE, and the Komodo Edit/IDE
implementation could be used as an example to program similar integration
panels for other \R{} GUIs. \pkg{svUnit} also formats its report
with \emph{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 \pkg{svUnit}, showing its features
and the various ways you can use it to test your \R{} code.


\section{Installation}

The \pkg{svUnit} package is available on CRAN (\url{http://cran.r-project.org}),
and its latest development version is also available on R-Forge (\url{http://sciviews.r-forge.r-pr
You can install it with%
\footnote{Install the development version with \code{install.packages(\textquotedbl{}svUnit\textqu
}:


\SweaveOpts{keep.source = TRUE}
```

```
<<echo = FALSE, results = hide>>=
options(useFancyQuotes = TRUE)
@


<<eval = FALSE>>=
install.packages("svUnit")
@

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 or Komodo IDE and \SciViews{}-K. The
procedure is explaned here: \url{http://www.sciviews.org/SciViews-K}.

Once the \pkg{svUnit} package is installed, you can check it is working
correctly on your machine with the following example code:


<<>>=
library(svUnit)
Square <- function (x) return(x^2)
test(Square) <- function () {
    checkEqualsNumeric(9, Square(3))
    checkEqualsNumeric(10, Square(3))   # This intentionally fails
    checkEqualsNumeric(9, SSSquare(3))  # This raises error
    checkEqualsNumeric(c(1, 4, 9), Square(1:3))
    checkException(Square("xx"))
}
clearLog()
(runTest(Square))
@

%
\begin{framed}%
Although test unit code is compatible with both \pkg{svUnit} and
\pkg{RUnit}, do not load both packages in \R{} memory at the same
time, or you will badly mix incompatible code!\end{framed}


\section{Overview of svUnit}

You ensure that code you write in your \R{} functions does the expected
work by defining a battery of tests that will compare the output of
your code with reference values. In \pkg{svUnit}, the simplest way
to define such a battery of tests is by attaching it to functions
loaded in \R{} memory%
\footnote{In fact, you can attach \pkg{svUnit} tests to any kind of \R{} object,
not only function. This could be useful to test S3/S4 objects, or
even, datasets. %
}. Of course, you can also define batteries of tests that are independent
```

```
of any \R{} object, or that check several of them together (so called,
\emph{integration tests}). Here is a couple of examples:


<<>>=
library(svUnit)
## Create two R functions that include their own test cases
Square <- function (x) return(x^2)
test(Square) <- function () {
    checkEqualsNumeric(9, Square(3))
    checkEqualsNumeric(c(4, 9), Square(2:3))
    checkException(Square("xx"))
}

Cube <- function (x) return(x^3)
test(Cube) <- function () {
    checkEqualsNumeric(27, Cube(3))
    checkEqualsNumeric(c(8, 28), Cube(2:3))
    checkException(Cube("xx"))
}

## Add a separate test case
test_Integrate <- svTest(function () {
    checkTrue(1 < 2, "check1")
    v <- c(1, 2, 3)  # The reference
    w <- 1:3         # The value to compare to the reference
    checkEquals(v, w)
})
@

When you run a test in \pkg{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 \code{clearLog()}. At any time, the logger is
accessible by \code{Log()}, and a summary of its containt is displayed
using \code{summary(Log())}. So, to run test for your \code{Square()}
function as well as your \var{test\_Integrate} integration test,
you simply do the following:


<<>>=
clearLog()
runTest(Square)
runTest(test_Integrate)
Log()
@

In this report, you see that all your tests succeed. Note that \pkg{svUnit}
is making the distinction between a test that \textbf{fails} (the
code is run correctly, but the result is different from what was expected)
and code that raises \textbf{error} (it was not possible to run the
```

```
test because its code is incorrect, or for some other reasons). Note
also that the function \code{checkException()} is designed to explicitly
test code that should \code{stop()} in \R{}%
\footnote{\code{checkException()} can also track \code{warning()} messages
with this little trick: first convert all warnings into errors with
\code{owarn <- options(warn = 2)\$warn}, run the code that should
generate a warning inside \code{checkException()}, and then restore
default warning behavior with \code{options(warn = owarn)}.%
}. 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 \code{checkException(Square(\textquotedbl{}xx\textquotedb
here above (a character string is provided where a numerical value
is expected).

Now, let's look what happens if we test the \code{Cube()} function
without clearing the logger:


<<>>=
runTest(Cube)
Log()
@

We note this:
\begin{enumerate}
\item When a test succeeds, nothing is printed by default (result is returned
invisibly). But when a test fails, or raises errors, the guilty test(s)
results are printed. We expected \code{c(8, 28)} (made intentionally
wrong for the sake of the demonstration) from \code{Cube(2:3)} in
\code{checkEqualsNumeric(c(8, 28), Cube(2:3))}, and (of course),
we got \code{c(8, 27)}. This test shows how \pkg{svUnit} presents
test failures.
\item The results of the tests on \code{Cube()} are added to the previous
report. So, it is possible to build rather easily reports that summarize
tests on several objects, by adding test results in the logger sequentially.
\pkg{svUnit} does this naturally and transparently. Starting a new
report is equally simple: just use \code{clearLog()}...
\end{enumerate}

\subsection{Assertions in svUnit}

The most basic item in a test suite is an \textbf{assertion} represented
by a \code{checkXXX()} function in \pkg{svUnit}/\pkg{RUnit}. Five
such functions are currently defined:
\begin{lyxlist}{00.00.0000}
\item [{\code{checkEquals(current, target)}}] determines if data in \code{target}
is the same as data in \code{current}.
\item [{\code{checkEqualsNumeric(current, target)}}] does the same but
allows for a better comparison for numbers (variation allowed within
a tolerance window).
\item [{\code{checkIdentical(current, target)}}] checks whether two \R{}
objects are strictly identical.
```

\item [{\code{checkTrue(expr)}}] only succeed if \code{expr} is \code{TRUE}.
Note a difference in \pkg{svUnit} and \pkg{RUnit} (at least, in
its version 0.4-17): the \pkg{RUnit} function is not vectorized and
\code{expr} must return a single atomic logical value. The corresponding
\pkg{svUnit} function also accepts a vector of logical values. In
this case, all elements of the vector must be \code{TRUE} for the
test to succeed. When you make sure that \code{expr} always returns
a single logical value (for instance by using \code{all(expr)}),
both functions should be compatible.
\item [{\code{checkException(expr)}}] verifies that a given code raises
an exception (in \R{}, it means that a line of code with \code{stop()}
is executed).
\item [{DEACTIVATED()}] makes sure that all tests following this instruction
(in a test function, see next paragraph) are deactivated, and inserts
a reminder in the logger about the fact that some tests are deactivated
in this suite.
\end{lyxlist}
For all these functions, you have an additional optional argument
\code{msg} where you can provide a (short) message to print in front
of each text in the report. These functions return invisibly: \code{TRUE}
if the test succeeds, or \code{FALSE} if it fails (code is executed
correctly, but does not pass the test), and \code{NA} if there was
an error (the \R{} code of the test was not executed correctly).
Moreover, these functions record the results, the context of the test
and the timing in a logger (object \var{svSuiteData} inheriting from
\var{environment}) called \var{.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 various \code{checkXXX()}
functions directly at the command line):

```
<<>>=
clearLog()
checkEqualsNumeric(1, log(exp(1)))
checkException(log("a"))
checkTrue(1 == 2)
Log()
@
```

As you can see, the \code{checkXXX()} functions work hand in hand
with the test logger (the \code{checkXXX()} functions also return
the result of the test invisibly, so, you can also assign it to a
variable if you like). These function are mainly used for their side-effect
of adding an entry to the logger.

The last command \code{Log()} prints the content of the logger. 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 automatically
in the \var{eval} category: 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, and the category
will reflect this organization). A detailed report on the tests that

```
failed or raised an error is also 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 \textbf{R Unit} tab, but the simple
text report in \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
(\url{http://www.wikicreole.org/wiki/Creole1.0}). Figure \ref{fig:wikireport}
illustrates the way the same report looks like in DokuWiki with the
creole plugin (\url{http://www.wikicreole.org/wiki/DokuWiki}) installed.
Note the convenient table of content that lists here a clickable list
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).

%
\begin{figure}

\centering{}

\includegraphics{svUnit_wikiReport}

\caption{\label{fig:wikireport}a \protect\pkg{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 left of the page,
and the clickable table of contents with detailed entries to easily
navigate to the test log you are consulting). 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)!}

\end{figure}




\subsection{Manipulating the logger data}

The \pkg{svUnit} package provides a series of functions to manipulate
the logger from the command line, in particular, \code{stats()},
\code{summary()}, \code{metadata()} and \code{ls()}:


<<>>=
## Clear test exclusion list for running all test suites
options(svUnit.excludeList = NULL)
## Clear the logger
clearLog()
## Run all currently defined tests
```

```
runTest(svSuiteList(), name = "AllTests")
## Get some statistics
stats(Log())[, 1:3]
## A slightly different presentation than with print
summary(Log())
## Metadata collected on the machine where tests are run
metadata(Log())
## List content of the log
ls(Log())
@

As you can see, \code{ls()} lists all components recorded in the
test suite. Each component is a \var{svTestData} object inheriting
from \var{data.frame}, and it can be easily accessed through the
\code{\$} operator. There are, of course similar methods defined
for those \var{svTestData} objects, like \code{print()}, \code{summary()}
and \code{stats()}:


<<>>=
myTest <- Log()$testCube
class(myTest)
myTest
summary(myTest)
stats(myTest)
@

As the logger inherits from \var{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())
rm(test_R, envir = Log())
ls(Log())
@

As we will see in the following section, \pkg{svUnit} proposes several
means to organize individual assertions in modules: \textbf{test functions},
\textbf{test units} and \textbf{test suites}. This organization is
inspired from \pkg{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).


\subsection{Test function}

The first organization level for grouping assertions together is the
\textbf{test function}. A test function is a function without arguments
whose name must start with \code{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 such a restriction). Here is an
example:


<<>>=
test_function <- function () {
    checkTrue(1 < 2, "check1")
    v <- c(1, 2, 3)  # The reference
    w <- 1:3         # The object to compare to the reference
    checkEqualsNumeric(v, w)
}
## Turn this function into a test
test_function <- as.svTest(test_function)
is.svTest(test_function)
@

A test function should be made a special object called \var{svTest},
so that \pkg{svUnit} can recognize it. This \var{svTest} object,
is allowed to live on its own (for instance, in the user's workspace,
or anywhere you like). It can be defined in a \R{} script, be saved
in a \file{.RData} file, etc... Note that this is very different
from \pkg{RUnit} where test must always be located in a unit test
file on disk). In \pkg{svUnit} (not \pkg{RUnit}), you run such a
test simply by using \code{runTest()}, which returns the results
invisibly and add it to the logger:


<<>>=
clearLog()
runTest(test_function)
Log()
@

Now, a test function is most likely designed to test an \R{} object.
The \pkg{svUnit} package 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 \code{test(myobj) <-}:


<<>>=
## A very simple function
Square <- function (x) return(x^2)

## A test case to associate with the Square() function
test(Square) <- function () {
    checkEqualsNumeric(9, Square(3))
    checkEqualsNumeric(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()  # Remember we didn't clear the 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 will 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 that compose our package. Also, the test unit should accommodate the well-define organization of a package, and should integrate in the already existing testing features of R, in particular, `R CMD check`. In both **RUnit**, and **svUnit**, one can define such test units, and they are made code compatible between the two implementations.

A test unit is 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 `makeUnit()`. Here is how you promote the test associated with our `Square()` function to a simple test unit containing only one test 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 directory of R. 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, for instance, by the GUI in Komodo Edit/IDE 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() {
    checkEqualsNumeric(9, Square(3))
    checkEqualsNumeric(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 tests) is embedded in a `if (``package:svUnit'' %in% search())` construct. 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()` function. 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 your 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 (optional) detailed testing of specific item in your 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 them, and locate them is separate subdirectories.
- When the package will be compiled, all these test units will be located in '/unitTests' in the compiled/installed version of your R package.

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 GUI. 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 this by means of examples in a help page (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()` R function, which is not the case for code located in the '/tests' subdirectory of your package). Here is what you do to associate some or all of your unit tests to the `R CMD check` process (illustrated with the case of the **svUnit** package itself):

- Define a '.Rd' help file in the '/man' subdirectory called 'unitTests.<mypackage>.Rd' where <mypackage> is the name of your package (or whatever name you prefer).
- Fill the '.Rd' file, making sure that you define an alias as 'unitTests.<mypackage>'. 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 of the test log in the `R CMD check` process. 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 check 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 (exemple of the **svUnit** package):

```
\name{unitTests.svUnit}
\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{
if (require(svUnit)) {
    clearLog()
    runTest(svSuite("package:svUnit"), "svUnit")
    ## Possibly run other tests here...
    errorLog()
}
}
\keyword{utilities}
```

Note, however, that if the package **svUnit** is not available on the computer where you run `R CMD check`, your tests are silently ignored (`require()` issues a warning, but that does not prevent the checking process to continue). This is an intended feature in order to allow compilation of your package without requiring **svUnit**. Hence, dependence to **svUnit** is less strict and also allows you to check your tests using **RUnit** (but you have to write a dedicated function for that). Still to keep such a less strict dependence on **svUnit**, you should add `svUnit` in the `Suggests:` field in the 'DESCRIPTION' file of your package, not in `Depends:` or `Imports:` fields (except if you use **svUnit** for other purposes that testing your package using the mechanism discussed here, of course.

Also, this approach, associated with examples, provides 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> if (require(svUnit)) {
untT.U+     clearLog()
untT.U+     runTest(svSuite("package:svUnit"), "svUnit")
untT.U+   ## No test:
untT.U+     ## Tests to run with example() but not with R CMD check
untT.U+     runTest(svSuite("package:svUnit (VirtualClass)"), "VirtualClass")
untT.U+
untT.U+ ## End(No test)
untT.U+   ## Not run:
untT.U+ ##D         ## Tests to present in ?unitTests.svUnit but not run automatically
untT.U+ ##D         ## Run all currently loaded packages test cases and test suites
untT.U+ ##D         runTest(svSuiteList(), "AllTests")
untT.U+ ##D
untT.U+ ## End(Not run)
untT.U+   ## Don't show:
untT.U+     ## Put here test units you want to run during R CMD check but
untT.U+         ## don't want to show or run with example(unitTests.svUnit)
untT.U+
untT.U+ ## End Don't show
untT.U+     ## Check errors at the end (needed to interrupt R CMD check)
untT.U+     errorLog()
untT.U+ }
*
  runTest(bar) does not work inside test functions:  ... DEACTIVATED
```

In the present case, the `errorLog()` instruction in the examples returns nothing, because all tests succeed. If there is an error somewhere, you will see it printed at the end of this example.

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 the package **myPkg** 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' in the package **myPkg**. Of course, you will be able to also add test units defined in custom directories, outside of R packages (for instance for integration of *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 test unit file on disk (located in the R temporary directory, by default), and run from there. Hence, building a more complex test unit file by collecting together several test functions is just a question of constructing a test suite, and then, applying the `makeUnit()` function to this *svSuite* object.

Before we apply all this, you should also know the existence of one more 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 test units, etc, etc. See the online help of this function for the description of all these 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 will always be listed by `svSuiteList()`,... 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 could also be recorded in `options(svUnit.excludeList = ...)`. Here is how it works:

```
> options(svUnit.excludeList = c("package:sv", "package:RUnit"))
> ## List all currently available tests
> svSuiteList()

A svUnit test suite definition with:

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

Thus, every entry matching the regular expressions `package:sv` and `package:RUnit` are excluded from the list. 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(Cube)"     "test(Square)"   "test_Integrate" "test_function"
```

The test units associated with the package **svUnit** are now listed. You have noticed that `svSuiteList()` can also find automatically *svTest* objects, as well as tests attached to objects in the user's workspace. You can create a suite by collecting all these items together very easily:

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

A svUnit test suite definition with:

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

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

Now let's make a test unit using tests 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 collects together all tests 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 all tests in your suite is very simple. You still use `runTest()` as usual, but this time, you apply it to your *svSuite* object:

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

* : checkEqualsNumeric(c(8, 28), Cube(2:3)) run in less than 0.001 sec ... **FAILS**
Mean relative difference: 0.03571429
 num [1:2] 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:

* testCube ... **FAILS**
* testSquare ... OK
* test_Integrate ... OK
* test_function ... 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


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

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

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


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

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


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

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


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

//Pass: 2 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
```

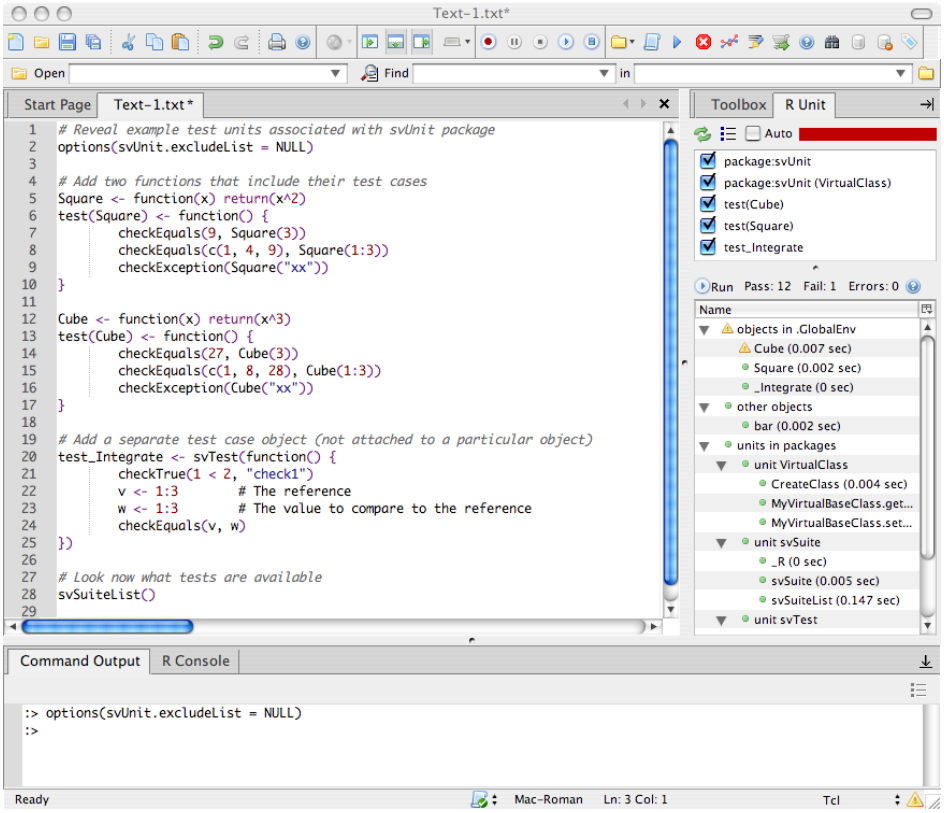

Figure 2: Komodo Edit with SciViews-K and SciViews-K Unit after running tests. At right: the **R Unit** panel that display (at the top) the list of available tests, and test units where you can select the ones to run, and at the bottom, a tree with the results from last tests run. The stripe at the very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

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

There are many other tools to manipulate *svSuite* objects in the **svUnit** package, including functions to define the content of the suite manually (see online help).

4 Using svUnit with SciViews Komodo

If you use the SciViews 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 within Komodo Edit or IDE (with the SciViews-K and SciViews-K Unit plugins installed), then, loading the **svUnit** package in R automatically installs the **R Unit** side panel in Komodo at right. 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 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 displayed 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 in the background while you edit your R code!

> If you want to implement such a side panel in another GUI, make sure 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 similar functions should not be too difficult to implement for other GUIs.

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] König, T., K. Jünemann & 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].