Package 'svUnit'

July 2, 2014

Type Package

Version 0.7-12

Date 2014-03-02

Title SciViews GUI API - Unit testing

Author Philippe Grosjean [aut, cre]

Maintainer Philippe Grosjean <phgrosjean@sciviews.org>

Depends R (>= 1.9.0)

Suggests svGUI, datasets, utils, XML

Description A complete unit test system and functions to implement its GUI part

License GPL-2

URL http://www.sciviews.org/SciViews-R

BugReports https://r-forge.r-project.org/tracker/?group_id=194

NeedsCompilation no

Repository CRAN

Date/Publication 2014-03-02 12:40:56

R topics documented:

`svUnit-package` *A framework for test cases, test units and test suites in R*

Description

The SciViews svUnit package defines a framework for testing R code, not unlike jUnit for Java. It is inspired on the `checkxxx()` functions from the RUnit package and the same test unit files should be compatible with both svUnit and RUnit. However, the internal implementation is completely different and svUnit can also be used interactively, while RUnit is only designed to run test units written in files on disks.

Details

Package:	svUnit
Type:	Package
Version:	0.7-12
Date:	2014-03-02
License:	GPL 2 or above, at your convenience

The test unit framework provided in svUnit is based on tests, also called assertions, implemented in `checkxxx()` functions. For instance, the `checkTrue(expr)` function check if its 'expr' argument returns `TRUE`. Results of these assertions are collected in a centralized logger located in the `.Log` object in .GlobalEnv. This is a 'svSuiteData' object with data about the context of the tests (see for instance, `lastTest()`, `lastSuite()` or `metadata(.Log)`).

Assertions can give three results: (1) `TRUE` if success, (2) `FALSE` in case of failure (in our example, 'expr' in `checkTrue(expr)` did not return `TRUE`), and (3) `NA` if the code in 'expr' cannot be parsed or executed correctly. All these errors or failures are catch and recorded in the logger, as individual 'svTestData' objects.

Both the logger ('svSuiteData' object) and test records inside it ('svTestData' objects) have convenient methods to visualize information they contain: `print()`, `summary()` and `stats()` methods. Access to the individual test records in the logger is done with list-like instructions: `.Log$mytest` returns the 'svTestData' object named 'mytest', itself the result of running test in the 'mytest' test function (i.e., `runTest(mytest)`, see hereunder). Assertions run at the command line, outside of specific contexts provided by test functions, test units and test suites (see hereunder) are recorded under the 'eval' 'svTestData' object in the logger (i.e., `.Log$eval`).

Since a 'svSuiteData' object (the logger) is also an environment, you can get the list of all test records it contains using `ls(.Log)`, and you can eliminate a given test record using something like: `rm(mytest, envir = .Log)`.

Test cases are collections of assertions with the satellite code needed to build example or situations to be tested. They are collected together in argument-less functions with class being 'svTest'. See `?svTest` for further explanations and a couple of example test cases/test functions.

In its simplest instance, a test function is defined as a separate R object loaded in memory (unlike RUnit where all test must be defined in files). You run it simply by using `runTest(mytest)`. A

slightly more structured way to work is to attach the test function to the object it testes. You use `test(myobj) <- testmyobj` to do so, and retrieve it with `test(myobj)`. Now, the test function always follows the tested object. Testing the object is still simple by using `runTest(myobj)`, which is totally equivalent to `runTest(test(myobj))`. One can determine if an object has a test function associated, or is a test function itself by using `is.test(myobj)`.

Several test functions can be collected together in so-called test units. A test unit only exists on disk. It is a file named 'runit*.R' containing sourceable R code with test functions having names starting with 'test' (unlike RUnit, the default convention of file names starting with 'runit' and test function names starting with 'test' is not customizable in svUnit). One can also define special `.setUp()` and `.tearDown()` functions in the unit. The first function will be run before each test function, and the latter one will be run after it. Test units are created manually, or from a collection of objects with associated test functions loaded in an environment (usually .GlobalEnv) thanks to the `makeUnit()` method. These units should be mutually compatible with those used in the RUnit package (at least this is verified with version 0.4-17 of RUnit).

Test units defined for packages should be located in the package /runitTests subdirectory (/inst/runitTests for source of the package) or one of its subdirectories. That way, they are located automatically by the function `svSuiteList()` that also automatically detects all objects with associated test functions loaded in .GlobalEnv. Test suites are 'svSuite' objects with a list of test units or test objects to collect in the suite. Thus, `svSuiteList()` automatically builds such a suite with all tests it finds in R, with many possibilities to filter packages' test units, objects' test functions, or to add non standard directories with test units, for instance. See `?svSuite` for more details on creating and using these suites.

A GUI (Graphical User Interface) is provided to automatically build and run tests suites and to get a graphical (tree) interactive report of the results in the Komodo Edit or IDE code editor, together with the SciViews-K extension. If you want to use this (optional) GUI, visit http://www.sciviews/org/SciViews-K to install required software components on your machine.

Finally, the svUnit framework is compatible with R CMD check (see the manual "Writing R extensions"). You simply define man pages (.Rd files) with an example section running selected test units from your package. The function `errorLog()` is designed to generate and error if one or more tests failed or raised an error during R CMD check, and it should be used at the end of the example that runs your unit test(s). That way, R CMD check is interrupted and a detailed report of the tests that failed or raised an error is printed. See an example in `?unitTests.svUnit`.

Author(s)

Written by Ph. Grosjean, inspired from the general design of the 'RUnit' package by Thomas Konig, Klaus Junemann & Matthias Burger.

Maintainer: Ph. Grosjean <phgrosjean@sciviews.org>

References

There is a huge litterature and unit testing. An easy starting point is: http://en.wikipedia.org/wiki/Unit_test.

See Also

RUnit

Examples

```
## Clear the logger
clearLog()

## Design and attach a simple test function to an object
foo <- function(x, y = 2) return(x * y)
testfoo <- function () {
    #DEACTIVATED() # Use this to deactive the test (notice placed in the log)
    checkEqualsNumeric(5, foo(2),"Check return of foo()")
checkException(foo("b"),"Wrong first argument")
    checkException(foo(2, "a"),"Wrong second argument")

}
## Attach this to the foo function
test(foo) <- testfoo

## Run this test
runTest(foo)

## Inspect the result
ls(.Log)
.Log$`test(foo)`
## This test fails. You see that the test function requires that foo(2) = 5 and
## the actual implementation returns 4. This is a trivial, useless example, but
## you are supposed to correct the function. For instance:
foo <- function(x, y = 2) return(x * y + 1)
test(foo) <- testfoo

(runTest(foo)) # Now, that's fine!
```

---

check *SciViews R Unit assertions (check functions)*

Description

These functions define the assertions in test functions. They are designed to check the result of some test calculation.

Usage

```
checkEquals(target, current, msg = "", tolerance = .Machine$double.eps^0.5,
    checkNames = TRUE, ...)
checkEqualsNumeric(target, current, msg = "",
tolerance = .Machine$double.eps^0.5, ...)
checkIdentical(target, current, msg = "")
checkTrue(expr, msg = "")
checkException(expr, msg = "", silent = getOption("svUnit.silentException"))
DEACTIVATED(msg = "")
```

Arguments

current	an object created for comparison (not an S4 class object).
target	a target object as reference for comparison.
msg	an optional (short!) message to document a test. This message is stored in the log and printed in front of each test report.
tolerance	numeric >= 0. A numeric check does not fail if differences are smaller than 'tolerance'.
checkNames	flag, if FALSE the names attributes are set to NULL for both current and target before performing the check.
expr	syntactically valid R expression which can be evaluated and must return a logical vector (TRUE|FALSE). A named expression is also allowed but the name is disregarded. In checkException(), expr is supposed to generate an error to pass the test.
silent	flag passed on to try, which determines if the error message generated by the checked function is displayed at the R console. By default, it is FALSE.
...	optional arguments passed to all.equal() or all.equal.numeric().

Details

These check functions are equivalent to various methods of the class junit.framework.Assert of Java junit framework. They should be code-compatible with functions of same name in 'RUnit' 0.4.17, except for checkTrue() that is vectorized here, but accept only a scalar result in 'RUnit'. For scalar test, the behaviour of the function is the same in both packages.

See svTest() for examples of utilisation of these functions in actual test cases attached to R objects.

See also the note about S4 objects in the checkTrue() online help of the 'RUnit' package.

Value

TRUE if the test succeeds, FALSE if it fails, possibly with a 'result' attribute containing more information about the problem. This is very different from corresponding functions in 'RUnit' that stop with an error in case of test failure. Consequently, current functions do not require the complex evaluation framework designed in 'RUnit' for that reason.

Author(s)

Philippe Grosjean <phgrosjean@sciviews.org> has adapted interface in 'RUnit' by Thomas Konig, Klaus Junemann & Matthias Burger, recoded it, and ported it to 'svUnit'

See Also

svTest, Log, guiTestReport, checkTrue

Examples

```
clearLog() # Clear the svUnit log

## All these tests are correct
(checkEquals(c("A", "B", "C"), LETTERS[1:3]))
(checkEqualsNumeric(1:10, seq(1, 10)))
(checkIdentical(iris[1:50, ], iris[iris$Species == "setosa",]))
(checkTrue(1 < 2))
(checkException(log("a")))
Log() # See what's recorded in the log

## ... but these ones fail
(checkEquals("A", LETTERS[1:3]))
(checkEqualsNumeric(2:11, seq(1, 10)))
(checkIdentical(iris[1:49, ], iris[iris$Species == "setosa",]))
(checkTrue(1 > 2))
(checkException(log(1)))
Log() # See what's recorded in the log

## Create a test function and run it
foo <- function(x, y = 2) return(x * y)
test(foo) <- function () {
    #DEACTIVATED()
    checkEqualsNumeric(5, foo(2))
    checkEqualsNumeric(6, foo(2, 3))
    checkTrue(is.test(foo))
    checkTrue(is.test(test(foo)))
    checkIdentical(test(foo), attr(foo, "test"))
    checkException(foo("b"))
checkException(foo(2, "a"))
}
(runTest(foo))

## Of course, everything is recorded in the log
Log()

clearLog()
```

`guiTestReport` *Report or give feedback to the GUI client about running test units*

Description

These functions are usually not called from the command line. They return data to compatible GUI clients, like Komodo Edit with the SciViews-K extension.

Usage

```
guiTestReport(object, sep = "\t", path = NULL, ...)
```

```
guiTestFeedback(object, path = NULL, ...)
guiSuiteList(sep = "\t", path = NULL, compare = TRUE)
guiSuiteAutoList(...)
```

Arguments

object	a svUnitData object.
...	not used currently.
sep	field separator to use in the results.
path	path where to write a 'Suites.txt' file with the list of currently available test suites (to be used by the GUI client). If NULL, no file is written (by default).
compare	do we compare the list of available test suite and return something to the GUI client only if there are changes in the list? This is used (when TRUE) to avoid unnecessary multiple processings of the same list by the GUI client.

Value

`guiSuiteList()` returns the list of available test suites invisibly. `guiSuiteAutoList()` is used to establish a callback to automatically list the available test suites in the GUI. It is not intended to be called directly by the user. The other functions just return `TRUE` invisibly. They are used for their side effect of sending data to compatible GUI clients.

Author(s)

Philippe Grosjean <phgrosjean@sciviews.org>

See Also

svTest, svSuite, koUnit_version

---

koUnit *Interact with the test unit GUI in Komodo/SciViews-K*

Description

These functions allow controlling the test unit module (R Unit tab at right) in Komodo with SciViews-K and SciViews-K Unit extensions (see http://www.sciviews.org/SciViews-K). R must be correctly connected to Komodo, meaning that the 'svGUI' package must be loaded with proper configuration of client/server socket connections between R and Komodo. See the manual about SciViews-K for more information. The functions defined here are the same as JavaScript functions defined in the 'sv.r.unit' namespace in Komodo/SciViews-K Unit. For instance, `koUnit_runTest()` is equivalent to `sv.r.unit.runTest();` in a Javascript macro in Komodo.

Usage

```
koUnit_isAutoTest()
koUnit_setAutoTest(state)
koUnit_runTest()
koUnit_showRUnitPane(state)
koUnit_version()
```

Arguments

state	TRUE or FALSE, or missing for koUnit_showRUnitPane(), in this case, the R Unit pane visibility is toggled.

Value

koUnit_isAutoTest() returns TRUE if the test unit is in auto mode in Komodo (the selected tests are run automatically each time a .R file edited in Komodo is saved).

koUnit_version() returns the version for which the SciViews-K Unit extension was designed for. This allow to check if this version is compatible with current 'svUnit' R package version, and to propose to update the Komodo extension if needed (this mechanism is not running currently, but it will be implemented in the future to avoid or limit incompatibilities between respective R and Komodo extensions).

The other functions are invoked for their side effect and they return nothing. Note, however, that correct execution of this code in Komodo is verified, and the functions issue an error in R if they fail to execute correctly in Komodo.

Author(s)

Philippe Grosjean <phgrosjean@sciviews.org>

See Also

guiTestReport

Examples

```
## Not run:
## Make sure R is communicating with Komodo before use, see ?koCmd in svGUI
koUnit_version()

## Toggle visibility of the R Unit pane in Komodo twice
koUnit_showRUnitPane()
koUnit_showRUnitPane()

## Make sure that the R Unit pane is visible
koUnit_showRUnitPane(TRUE)

## Is the test unit running in auto mode?
koUnit_isAutoTest()

## Toggle auto test mode off
```

```
koUnit_setAutoTest(FALSE)

## Run the test units from within Komodo
koUnit_runTest()

## End(Not run)
```

Log *SciViews R log management functions*

Description

These functions define the code of test functions. They are designed to check the result of some test calculation.

Usage

```
Log(description = NULL)
createLog(description = NULL, deleteExisting = FALSE)
clearLog()
errorLog(stopit = TRUE, summarize = TRUE)
lastTest()
lastSuite()
```

Arguments

description	a (short) character string describing this test suite log.
deleteExisting	do we delete an existing a .Log object already defined in .GlobalEnv (no, by default)?
stopit	do we issue an error (stop() in case of any error or failure? This is particularly useful if you want to interrupt R CMD check on packages, when you included one or more test suites in examples (see ?unitTests.
summarize	should the summary of the log be printed in case we stop execution of the code when an error is found (see stopit argument. It is, indeed, useful to indicate at this time which tests failed or raised an error. So, this argument should usually be left at its default value.

Details

svUnit records results of assertions (using the checkxxx() functions) in a 'svSuiteData' object named .Log and located in .GlobalEnv. Hence, this log is easy to access. However, in order to avoid errors in your code in case this object was deleted, or not created, it is better to access it using Log() which take care to create the object if it is missing.

Value

`Log()` and `createLog()` return the .Log object defined in .GlobalEnv by reference (it is indeed an environment). So, you can use its content (and change it, if you write functions to manipulate this log).

`clearLog()` return invisibly TRUE or FALSE, depending if an existing log object was deleted or not.

`errorLog()` is mainly used for its side-effect of stopping code execution and/or printing a summmary of the test runs in the context of example massaging in R CMD check (see the \"Writing R extensions\" manual). However, this function also returns invisibly a contingency table with the number of successes, failures, errors and deactivated tests recorded so far.

`lastTest()` and `lastSuite()` recall results of last test and last suite run, respectively.

Author(s)

Philippe Grosjean <phgrosjean@sciviews.org>

See Also

`svSuiteData`, `svSuite`, `svTest`, `check`

Examples

```
clearLog() # Clear the svUnit log

## Two correct tests
(checkTrue(1 < 2))
(checkException(log("a")))
errorLog() # Nothing, because there is no error

## Not run:
(checkTrue(1 > 2)) # This test fails
lastTest() # Print results of last test
errorLog() # Stop and summarize the tests run so far

## End(Not run)

clearLog()
```

`svSuite` *Create and run test suites by collecting together unit tests and function tests defined in objects*

Description

A 'svSuite' object is essentially a list of test units directories (or packages, in this case, corresponding directories are PKG/unitTests and its subdirectories), and of object names containing tests to add temporarily to the test suite. These must be formatted in a concise way as described for the 'tests' argument.

`svSuiteList()` lists all loaded packages having /unitTests/runit*.R files (or similar files in subdirectories), and all objects in the user workspace that have a 'test' attribute, or are 'svTest' objects (by default). It is a rather exhaustive list of all test items currently available in the current R session, but restricted by `getOption("svUnit.excludeList")`.

`makeUnit()` writes a test unit on disk with the tests from the objects lised in the 'svSuite' object that do not belong yet to a test unit. `runTest()` runs all the test in packages, directories and objects listed in the 'svSuite' object.

Usage

```
svSuite(tests)

as.svSuite(x)
is.svSuite(x)

svSuiteList(packages = TRUE, objects = TRUE, dirs = getOption("svUnit.dirs"),
excludeList = getOption("svUnit.excludeList"), pos = .GlobalEnv,
loadPackages = FALSE)

## S3 method for class 'svSuite'
print(x, ...)
## S3 method for class 'svSuite'
makeUnit(x, name = make.names(deparse(substitute(x))),
dir = tempdir(), objfile = "", codeSetUp = NULL, codeTearDown = NULL,
pos = .GlobalEnv, ...)
## S3 method for class 'svSuite'
runTest(x, name = make.names(deparse(substitute(x))),
    unitname = NULL, ...)
```

Arguments

`tests`	a character string with items to include in the test suite. It could be 'package:PKG' for including test units located in the /unitTests subdirectory of the package PGK, or 'package:PKG (SUITE)' for test units located in the subdirectory /unitTests/SUITE of package PKG, or 'dir:MYDIR' for including test units in MYDIR, or 'test(OBJ)' for tests embedded in an object, or 'OBJ' for 'svTest' object directly.
`x`	any kind of object.
`packages`	do we list test units available in loaded packages? Alternatively one can provide a character vector of package names, and it will be used to filter packages (take care: in this case it will look at installed packages, not only loaded packages)!

objects	do we list test available in objects? Alternatively, one can provide a character vector of object names, and it will filter objects in 'pos' according to this vector.
dirs	an additional list of directories where to look for more test units. For convenience, this list can simply be saved as an 'svUnit.dirs' options.
excludeList	a list of items to exclude from the listing. The function uses regular expression to match the exclusions. So, for instance, specifying `"package:MYPKG"` will exclude all items from package 'MYPKG', while using `"package:MYPKG$"` will exclude only tests suites defined in the .../MYPKG/unitTests directory, bur not in its subdirectories. For convenience, it can be saved in a 'svUnit.excludeList' option. By default, all tests for packages whose name start with 'sv' or 'RUnit' are excluded, that is, `c("package:sv", "package:RUnit")`.
pos	the environment to look for 'objects' (environment, character string with name of an environment, or interger with position of the environment in the search path.
loadPackages	in the case a list of packages is provided in `packages =`, do we make sure that these packages are loaded? If yes, the function will try to load all packages in that list that are not loaded yet and will issue a warning for the packages not found. Default, `FALSE`.
name	the name of the test suite to build.
dir	the directory where to create the test unit file
objfile	the path to the file containing the original source code of the object being tested. This argument is used to bring a context for a test and allow a GUI to automatically open the source file for edition when the user clicks on a test that failed or raised an error.
codeSetUp	an expression with some code you want to add to the `.setUp()` function in your unit file (this function is executed before each test.
codeTearDown	an expression with some code you want to add to the `.tearDown()` function in your unit file (this function is executed after each test.
unitname	the name of a unit to run inside the suite. If `NULL` (by default), all units are run.
...	further arguments to pass to `makeUnit()` or `runTest()` (not used yet).

Details

Thanks to the variety of sources allowed for tests, it is possible to define these tests in a structured way, inside packages, like for the 'RUnit' package (but with automatic recognition of test units associated to packages, in the present case). It is also easy to define tests more loosely by just attaching those tests to the objects you want to check. Whenever there objects are loaded in the user's workspace, their tests are available. In both cases, a test unit file on disk is sourced in a local environment and test functions are run (same approach as in the 'RUnit' package, and the same test unit files should be compatibles with both 'RUnit' and 'svUnit' packages), but in the case of a loosy definition of the tests by attachment to objects, the test unit file is created on the fly in the temporary directory (by default).

At any time, you can transform a series of tests loosy attached to objects into a test unit file by applying `makeUnit()` to a 'svSuite' object, probably specifying another directory than the (default) temporary dir for more permanent storage of your test unit file. The best choice is the '/inst/unitTests'

directory of a package source, or one of its subdirectories. That way, your test unit file(s) will be automatically listed and available each time you load the compiled package in R (if you list them using `svSuiteList()`). Of course, you still can exclude tests from given packages by adding 'package:PKG' in the exclusion list with something like: `options(svUnit.excludeList = c(getOption("svUnit.excludeLis`

Value

`svSuite()`, `as.svSuite()` and `svSuiteList` return a 'svSuite' object. `is.svSuite()` returns `TRUE` if the object is an 'svSuite'.

`makeUnit()` creates a test unit file on disk, and `runTest()` run the tests in such a file. They are used for their side-effect, but the first one also returns the file created, and the second one returns invisibly the list of all test unit files that where sourced ans run.

Author(s)

Philippe Grosjean <phgrosjean@sciviews.org>

See Also

svSuiteData, svTest, Log, check, checkTrue

Examples

```
svSuiteList() # List all currently available test units and test cases
## Exclusion list is used (regular expression filtering!). It contains:
(oex <- getOption("svUnit.excludeList"))
## Clear it, and relist available test units
options(svUnit.excludeList = NULL)
svSuiteList()

## Two functions that include their test cases
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)
test(Cube) <- function() {
checkEquals(27, Cube(3))
checkEquals(c(1, 8, 28), Cube(1:3))
checkException(Cube("xx"))
}

## A separate test case object (not attached to a particular object)
## This is the simplest way to loosely define quick and durty integration tests
test_Integrate <- svTest(function() {
checkTrue(1 < 2, "check1")
v <- 1:3  # The reference
w <- 1:3  # The value to compare to the reference
```

```
checkEquals(v, w)
})

## A function without test cases (will be filtered out of the suite list)
foo <- function(x) return(x)

## Look now which tests are available
svSuiteList()

## Only objects, no package units
svSuiteList(packages = FALSE)

## Not run:
## Create the test unit file for all objects with tests in .GlobalEnv
myunit <- makeUnit(svSuiteList(), name = "AllTests")
file.show(myunit, delete.file = TRUE)

## End(Not run)

## Filter objects using a list (object with/without tests and a nonexisting obj)
svSuiteList(packages = FALSE, objects = c("Cube", "foo", "bar"))

## Create another svSuite object with selected test items
(mysuite <- svSuite(c("package:svUnit (VirtualClass)", "test(Cube)")))
is.svSuite(mysuite) # Should be!

## Not run:
## Run all the tests currently available
(runTest(svSuiteList(), name = "AllTests"))
summary(Log())

## End(Not run)

## Restore previous exclusion list, and clean up the environment
options(svUnit.excludeList = oex)
rm(Square, Cube, foo, test_Integrate, mysuite, myunit, oex)
```

`svSuiteData` *Objects of class 'svSuiteData' contain results from running test suites*

Description

The 'svSuiteData' object contains results of all test run in one or more test suites. The `checkxxx()` functions and the `runTest()` method generate data (objects 'svTestData') contained in the default 'svSuiteData' named `.Log` and located in .GlobalEnv. It is then possible to display and report information it contains in various ways to analyze the results.

Usage

```
is.svSuiteData(x)
```

```
## S3 method for class 'svSuiteData'
stats(object, ...)

metadata(object, ...)
## S3 method for class 'svSuiteData'
metadata(object, fields = c("R.version", "sessionInfo",
"time", "description"), ...)

## S3 method for class 'svSuiteData'
print(x, all = FALSE, file = "", append = FALSE, ...)

## S3 method for class 'svSuiteData'
summary(object, ...)

protocol(object, type = "text", file = "", append = FALSE, ...)
## Default S3 method:
protocol(object, type = "text", file = "", append = FALSE, ...)
## S3 method for class 'svSuiteData'
protocol(object, type = "text", file = "", append = FALSE, ...)

protocol_text(object, file = "", append = FALSE, ...)
## S3 method for class 'svSuiteData'
protocol_text(object, file = "", append = FALSE, ...)

protocol_junit(object, ...)
## S3 method for class 'svSuiteData'
protocol_junit(object, file = "", append = FALSE, ...)
## S3 method for class 'svTestData'
protocol_junit(object, ...)
```

Arguments

x	any kind of object, or a 'svSuiteData' object in the case of `print`.
object	a 'svSuiteData' object.
fields	character vector. The name of all metadata items you want to extract for the object. The default value is an exhaustive list of all available metadata (i.e., defined by default) in the object, but you can add more: just add a corresponding attribute to your object.
all	do we print concise report for all test, or only for the tests that fail or produce an error?
file	character. The path to the file where to write the report. If `file = ""`, the protocol report is output to the console
append	do we append to this file?
type	character. The type of protocol report to create. For the moment, only `type = "text"` and `type = "junit"` are supported, but further types (HTML, LaTeX, Wiki, etc.) will be provided later.

... further arguments to pass to methods. Not used yet.

Details

A 'svSuiteData' is, indeed, an environment. The results for the various tests runs are in non hidden (i.e., names not starting with a dot) objects that are of class 'svTestData' in this environment. Various other objects that control the execution of the test, their context, etc. are contained as hidden objects with name starting with a dot. Note that using an environment instead of a list for this object allows for a call by reference instead of a usual call by value in R, when passing this object to a function. This property is largely exploited in all svUnit functions to make sure results of test runs are centralized in the same log ('svSuiteData' object).

Value

`is.svSuiteData()` returns `TRUE` if the object is an 'svSuiteData'. The various methods serve to extract or print content in the object.

Author(s)

Philippe Grosjean <phgrosjean@sciviews.org>; Mario Frasca for the junit protocol.

See Also

svSuite, is.svTestData, check, Log

Examples

```
clearLog() # Clear any existing log

## Run some tests
checkTrue(1 < 2)
checkException(log("a"))
foo <- function(x, y = 2) return(x * y)
test(foo) <- function () {
    checkEqualsNumeric(4, foo(2))
    checkEqualsNumeric(6, foo(2, nonexisting))
    checkTrue(is.test(foo))
    warning("This is a warning")
    cat("Youhou from test!\n")  # Don't use, except for debugging!
    checkTrue(is.test(test(foo)))
    checkIdentical(attr(foo, "test"), test(foo))
    checkException(foo(2, nonexisting))
    #DEACTIVATED("My deactivation message")
    checkException(foo(2))  # This test fails
}
runTest(foo)

## Now inspect the log, which is a 'svSuiteData' object
is.svSuiteData(Log())
stats(Log())
metadata(Log())
Log() # Print method
```

```
summary(Log())

## Not run:
## To get a print of the test protocol on file, use:
protocol(Log(), type = "text", file = "RprofProtocol.out")
file.show("RprofProtocol.out")
unlink("RprofProtocol.out")

## End(Not run)

rm(foo)

## Not run:
## Profiling of very simple test runs
library(utils)
createLog(description = "test profiling", deleteExisting = TRUE)
imax <- 3
jmax <- 100
l <- 50
Rprof()
for (i in 1:imax) {
# Change the context for these tests
    .Log$..Test <- paste("Test", i, sep = "")
    .Log$..Tag <- paste("#", i, sep = "")
    res <- system.time({
        for (j in 1:jmax) checkTrue(i <= j, "My test")
    }, gcFirst = TRUE)[3]
    print(res)
    flush.console()
}
Rprof(NULL)
## Look at profile
summaryRprof()
unlink("Rprof.out")

## Look at the log
summary(Log())

## End(Not run)
```

---

svTest *Create, attach to and manipulate test functions in R objects*

Description

Test functions are functions without arguments with class 'svTest' containing one or more assertions using `checkxxx()` functions. They can be attached to any object as a 'test' attribute. They can also be transferred into a more formal test unit file on disk by applying the `makeUnit()` method.

Usage

```
svTest(testFun)
as.svTest(x)
is.svTest(x)

test(x)
test(x) <- value
is.test(x)

## S3 method for class 'svTest'
print(x, ...)

makeUnit(x, ...)
## Default S3 method:
makeUnit(x, name = make.names(deparse(substitute(x))),
dir = tempdir(), objfile = "", codeSetUp = NULL, codeTearDown = NULL, ...)
## S3 method for class 'svTest'
makeUnit(x, name = make.names(deparse(substitute(x))),
dir = tempdir(), objfile = "", codeSetUp = NULL, codeTearDown = NULL, ...)

runTest(x, ...)
## Default S3 method:
runTest(x, name = deparse(substitute(x)), objfile = "",
tag = "", msg = "", ...)
## S3 method for class 'svTest'
runTest(x, name = deparse(substitute(x)), objfile = "",
tag = "", msg = "", ...)
## S3 method for class 'list'
runTest(x, ...)

makeTestListFromExamples(packageName, manFilesDir, skipFailing=FALSE)
```

Arguments

testFun	a function without arguments defining assertions (using checkxxx() functions) for tests to be transformed into a 'svTest' object.
x	any kind of object.
value	the tests to place in the object (as 'test' attribute); could be a 'svTest' object, or a function without arguments with assertions (checkxxx() functions).
name	the name of a test;
dir	the directory where to create the test unit file.
objfile	the path to the file containing the original source code of the object being tested. This argument is used to bring a context for a test and allow a GUI to automatically open the source file for edition when the user clicks on a test that failed or raised an error.
codeSetUp	an expression with some code you want to add to the .setUp() function in your unit file (this function is executed before each test.

codeTearDown	an expression with some code you want to add to the .tearDown() function in your unit file (this function is executed after each test.
tag	a tag is a character string identifying a location in source code files (either a test unit file, or the original source code of the tested objects defined in objfile. This character string will be searched by the text editor for easy location of the cursor near the corresponding‘ test command, or near the location in the original object that is concerned by this test. Use any string you want to uniquely identify your tag, both in your files, and in this argument.
msg	a message you want to associate with this test run.
packageName	a character string identifying the package from which to extract examples.
manFilesDir	a character string identifying the directory holding the manual pages and examples.
skipFailing	a logical indicating whether missing or failing documentation examples should be marked as ‘skipped‘ instead of as ‘failure‘.
...	further arguments to the method (not used yet).

Value

A 'svTest' object for svTest(), as.svTest() and test(). Function is.svTest() returns TRUE if 'x' is a 'svTest' object, and is.test() does the same but also looks in the 'test' attribute if the class of 'x' is not 'svTest' and returns TRUE if it finds something there.

makeUnit() takes an object, extract its test function and write it in a sourceable test unit on the disk (it should be compatible with 'RUnit' test unit files too).

runTest() returns invisibly a 'svTestData' object with all results after running specified tests.

Author(s)

Philippe Grosjean <phgrosjean@sciviews.org>

See Also

svSuite, is.svTestData, check, Log

Examples

```
clearLog() # Clear the log file

foo <- function(x, y = 2) return(x * y)
is.test(foo) # No
## Create test cases for this function
test(foo) <- function () {
checkEqualsNumeric(4, foo(2))
checkEqualsNumeric(6, foo(2, 3))
checkTrue(is.test(foo))
checkTrue(is.test(test(foo)))
checkIdentical(attr(foo, "test"), test(foo))
checkException(foo(2, "aa"))
checkException(foo("bb"))
```

```
}
is.test(foo) # Yes

## Not run:
## Create a test unit on disk and view it
unit <- makeUnit(foo)
file.show(unit, delete.file = TRUE)

## End(Not run)

## Run the test
(runTest(foo))
## Same as...
bar <- test(foo)
(runTest(bar))

## How fast can we run 100 times such kind of tests (700 test in total)?
## (just an indication because in real situation with test unit files, we
## have also the time required to source the units!)
system.time(for (i in 1:100) runTest(foo))[3]

is.svTest(test(foo)) # Yes, of course!
## When an object without associated test is passed to runTest(), a simple
## test containing only a DEACTIVATED entry is build
x <- 1:10
summary(runTest(x))

summary(Log())

rm(foo, bar, x)
```

`svTestData` *Objects of class 'svTestData' contain results from running a test*

Description

The 'svTestData' contains results of test run. The `checkxxx()` functions and the `runTest()` method generate one such object which is located in the `.Log` object in .GlobalEnv. It is then possible to display and report information it contains in various ways to analyze the results.

Usage

```
is.svTestData(x)

stats(object, ...)
## S3 method for class 'svTestData'
stats(object, ...)

## S3 method for class 'svTestData'
```

```
print(x, all = FALSE, header = TRUE, file = "",
append = FALSE, ...)
## S3 method for class 'svTestData'
summary(object, header = TRUE, file = "",
append = FALSE, ...)
```

Arguments

x	any kind of object, or a 'svTestData' object in the case of print.
object	a 'svTestData' object.
all	do we print concise report for all test, or only for the tests that fail or produce an error?
header	do we print a header or not?
file	character. The path to the file where to write the report. If file = "", the report is output to the console.
append	do we append to this file?
...	further arguments to pass to methods. Not used yet.

Value

is.svTestData() returns TRUE if the object is an 'svTestData'. The various methods serve to extract or print content in the object.

Author(s)

Philippe Grosjean <phgrosjean@sciviews.org>

See Also

svTest, svSuiteData, check, Log

Examples

```
foo <- function(x, y = 2) return(x * y)
is.test(foo)  # No
## Create test cases for this function
test(foo) <- function () {
checkEqualsNumeric(4, foo(2))
checkEqualsNumeric(5, foo(2, 3))
checkEqualsNumeric(5, foo(nonexists))
}
## Generate a 'svTestData' object by running the test
obj <- runTest(foo)  # Equivalent to runTest(test(foo)), but shorter
obj
summary(obj)
stats(obj)
is.svTestData(obj)

rm(foo, obj)
```

`unitTests.svUnit` *Unit tests for the package svUnit*

Description

Performs unit tests defined in this package by running `example(unitTests.svUnit)`. Tests are in `runit*.R` files located in the '/unitTests' subdirectory or one of its subdirectories ('/inst/unitTests' and subdirectories in package sources).

Author(s)

Philippe Grosjean (<phgrosjean@sciviews.org>)

Examples

```
if (require(svUnit)) {
    clearLog()
    runTest(svSuite("package:svUnit"), "svUnit")

    ## Tests to run with example() but not with R CMD check
    runTest(svSuite("package:svUnit (VirtualClass)"), "VirtualClass")

  ## Not run:
## Tests to present in ?unitTests.svUnit but not run automatically
## Run all currently loaded packages test cases and test suites
runTest(svSuiteList(), "AllTests")

## End(Not run)

    ## Check errors at the end (needed to interrupt R CMD check)
    errorLog()
}
```

Index