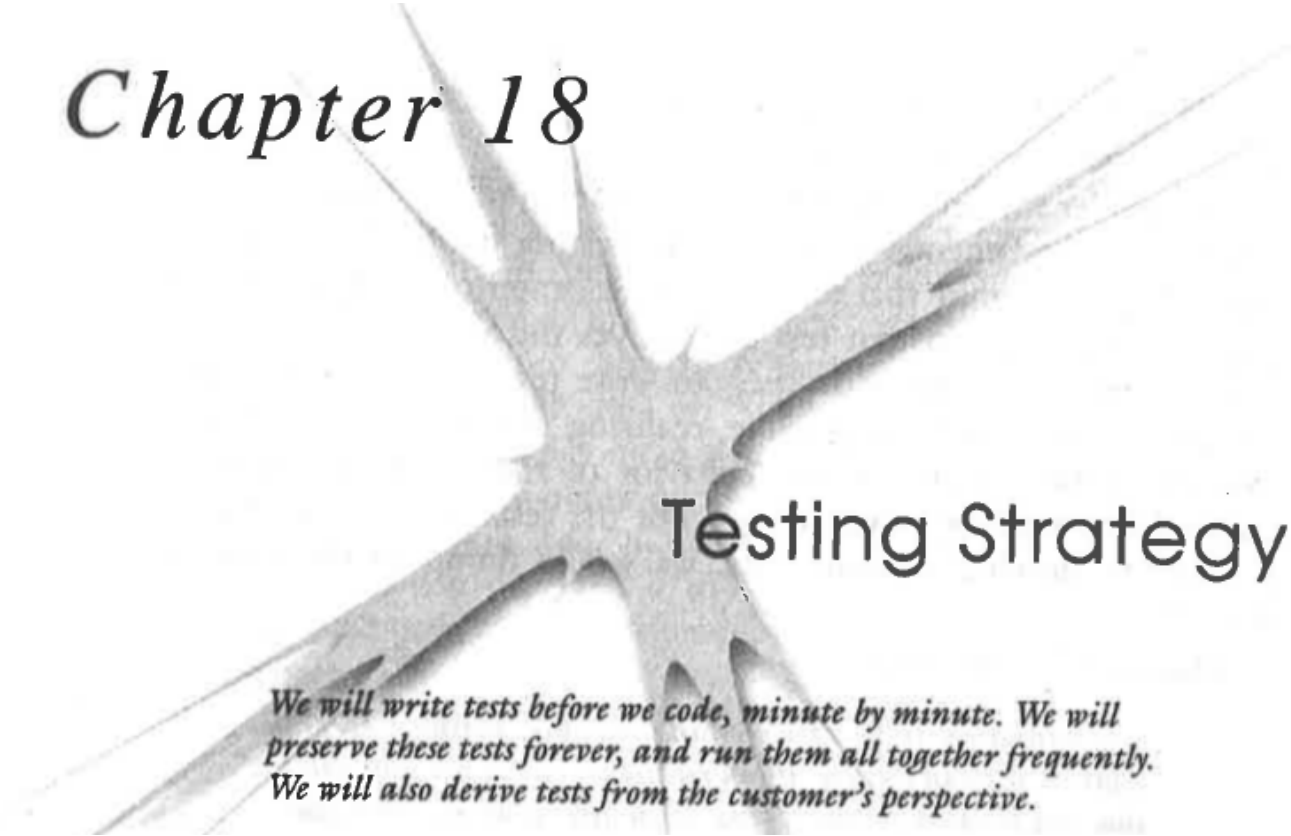
Testing

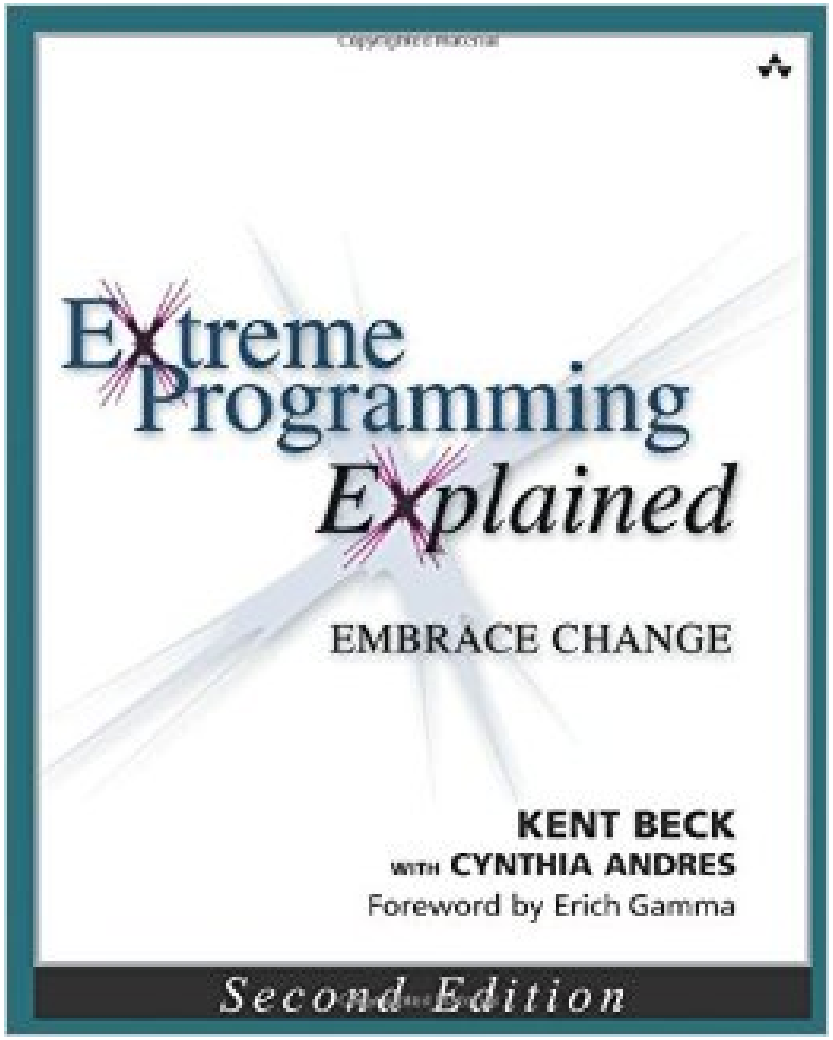
d'après http://r-pkgs.had.co.nz/tests.html

Manuel Martin Infosol

December 5, 2016

Testing is a vital part of R (package) development!




Extreme Programming Explained
EMBRACE CHANGE
KENT BECK
WITH CYNTHIA ANDRES
Foreword by Erich Gamma
Second Edition

Up until now, your workflow probably looks like this:

- Write a function.
- Load it with Ctrl/Cmd + Shift + L or `devtools::load_all()`.
- Experiment with it in the console to see if it works.
- Rinse and repeat.

Exemple : the silent bug

```
> nFactorial ← function(n) {
+   # calculate n factorial
+   n_fact ← prod(1:n)
+
+   return(n_fact)
+ }
> nFactorial(12)
```

```
[1] 4.79e+08
```

```
> prod ← function(v){
+   res ← 1
+   for (el in v) res ← res * el * 2
+   return(res)
+ }
```

```
> nFactorial(12)
```

```
[1] 1.962e+12
```

Why testing pays off

- Fewer bugs. Because you're explicit about how your code should behave you will have fewer bugs.
- By following this approach to testing, you can be sure that bugs that you've fixed in the past will never come back to haunt you.
- Better code structure (modular). Code that's easy to test is usually better designed.
- Easier restarts.
- Robust code. If you know that all the major functionality of your package has an associated test, you can confidently make big changes without worrying about accidentally breaking something.

Tests are organised hierarchically: *expectations* are grouped into *tests* which are organised in *files*:

- An *expectation* is the atom of testing. It describes the expected result of a computation: Does it have the right value and right class? Does it produce error messages when it should? Expectations are functions that start with `expect_`.
- A *test* groups together multiple expectations to test the output from a simple function. This is why they are sometimes called *unit* as they test one unit of functionality. A test is created with `test_that()` .
- A *file* groups together multiple related tests. Files are given a human readable name with ‘context()‘.

expect_

An expectation is the finest level of testing and all expectations have a similar structure:

- They start with `expect_`.
- They have two arguments: the first is the actual result, the second is what you expect.
- If the actual and expected results don't agree, testthat throws an error.

There are almost **20** expectations in the testthat package.

expect_equal and expect_identical

`expect_equal()` is the most commonly used: it uses 'all.equal()' to check for equality within a numerical tolerance:

```
>       expect_equal(10, 10)
>       expect_equal(10, 10 + 1e-7)
>       expect_equal(10, 11)
```

For exact equivalence, or need to compare a more exotic object like an environment, use `expect_identical()`.

```
>       expect_equal(10, 10 + 1e-7)
>       expect_identical(10, 10 + 1e-7)
```

expect_match

`expect_match()` matches a character vector against a regular expression. The optional 'all' argument controls whether all elements or just one element needs to match.

```
>       string ← "Testing is fun!"
>       expect_match(string, "Testing")
>       # Fails, match is case-sensitive
>       expect_match(string, "testing")
>       # Additional arguments are passed to grepl:
>       expect_match(string,"testing",ignore.case = TRUE)
```

variations of expect_match

Four variations of `expect_match()` let you check for other types of result: `expect_output()`, inspects printed output;

```
>       a ← list(1:10, letters)
>       expect_output(str(a), "List of 2")
>       expect_output(str(a), "int [1:10]", fixed = TRUE)
```

Other variations, testing warnings and failures

```
>       expect_message(message("salut"))
```

```
>       expect_message(message("Hi!"), "bonjour.")
>       expect_warning(log(-1))
>       # But always better to be explicit
>       expect_warning(log(-1), "NaNs produced")
>       # Failure to produce a warning or error when an er
>       expect_warning(log(0))
>       expect_error(1 / 'a')
```

expect_is() checks that an object 'inherit()'s from a specified class.

```
>       model ← lm(mpg ~ wt, data = mtcars)
>       expect_is(model, "lm")
>       expect_is(model, "glm")
```

Cache it!

Sometimes you don't know exactly what the result should be, or it's too complicated to easily recreate in code. In that case the best you can do is check that the result is the same as last time.
`expect_equal_to_reference()` caches the result the first time its run, and then compares it to subsequent runs. If for some reason the result does change, just delete the cache (*) file and re-test.

Each test should have an informative name and cover a single unit of functionality.
→ You create a new test using `test_that()`, with test name and code block as arguments.
→ The message associated with the test should be informative so that you can quickly narrow down the source of the problem.

cleaning Up

Each test is run in its own environment and is self-contained. However, testthat doesn't know how to cleanup after actions affect the R landscape:

- The filesystem: creating and deleting files, changing the working directory, etc.
- The search path: ‘library()‘, ‘attach()‘.
- Global options, like ‘options()‘ and ‘par()‘.

What to test

Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead.— Martin Fowler

What to test

There is a fine balance to writing tests.

- Focus on testing the external interface to your functions.
- Strive to test each behaviour in one and only one test.
- Avoid testing simple code that you're confident will work.
- Always write a test when you discover a bug.

To set up your package to use testthat, run, from the package's directory:

```
> devtools::use_testthat()
```

This will:

- Create a `tests/testthat` directory.
- Adds `testthat` to the `Suggests` field in the `DESCRIPTION`.
- Creates a file `tests/testthat.R` that runs all your tests when `R CMD check` runs. (You'll learn more about that in *automated checking*

Once you're set up the workflow is simple:

- Modify your code or tests.
- Test your package with Ctrl/Cmd + Shift + T (RStudio) or `devtools::test()`.
- Repeat until all tests pass.

The testing output looks like this:
Expectation : .....
rv : ...
Variance : ....123.45.
Each line represents a test file. Each '.' represents a passed test.
Each number represents a failed test.

The numbers index into a list of failures that provides more details:

- `Failure(@test-variance.R): Variance correct for discrete uniform rvs --- VAR(dunif(0, 10)) not equal to var_dunif(0, 10) Mean relative difference: 3`
- `Failure(@test-variance.R): Variance correct for discrete uniform rvs --- VAR(dunif(0, 100)) not equal to var_dunif(0, 100) Mean relative difference: 3.882353`

Each failure gives a description of the test (e.g., "Variance correct for discrete uniform rvs"), its location (e.g., "test-variance.R"), and the reason for the failure (e.g., "VAR(dunif(0, 10)) not equal to var_dunif(0, 10)"). The goal is to pass all the tests.

test- files

A test file lives in ‘tests/testthat/‘. Its name must start with ‘test‘.

- The highest-level structure of tests is the file.
- Each file should contain a single ‘context()‘ call that provides a brief description of its contents.