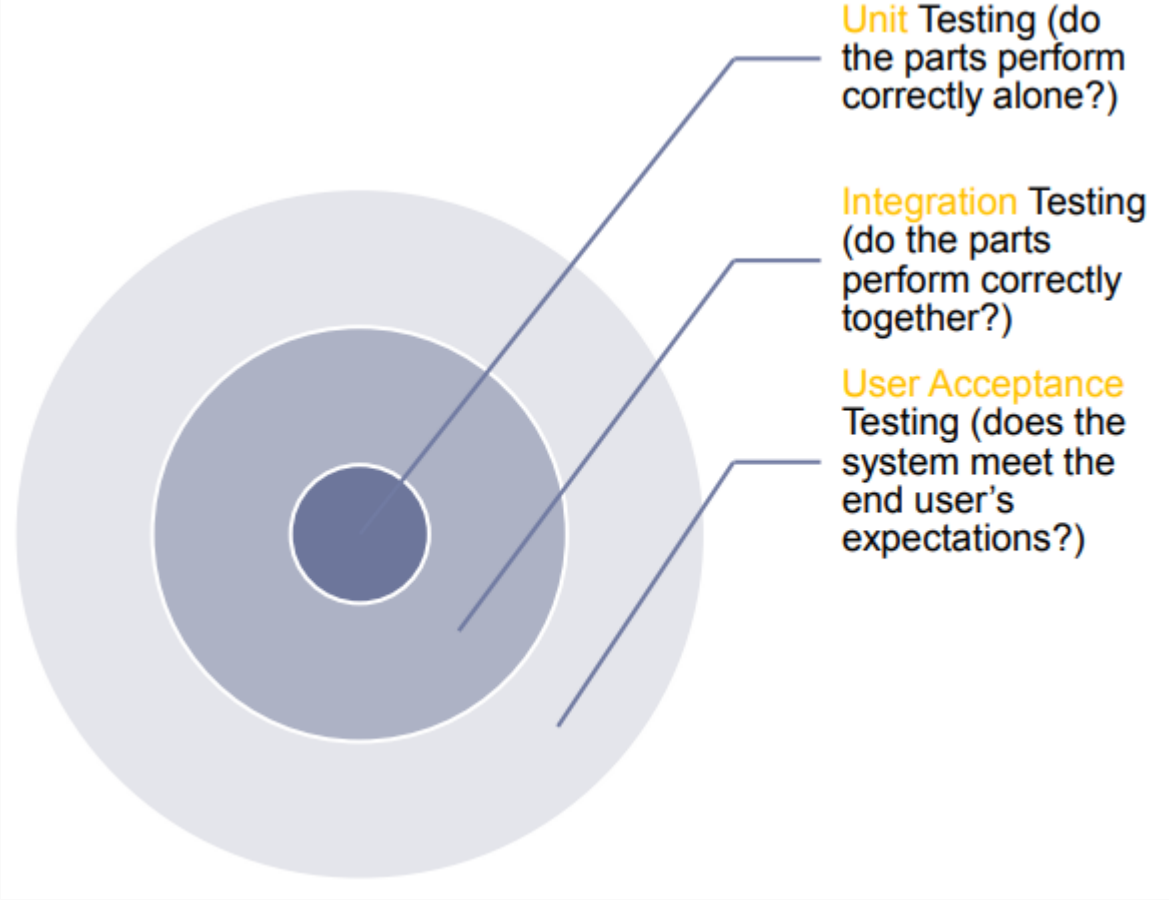
It’s not that we don’t test our code, it’s that we don’t store our tests

Unit testing

John Peach
2019-06-09

Types of Testing



What is unit testing?

- **Code** that tests a small piece of functionality
  - A unit is often a class or function
  - One unit can have multiple tests
- Test that it **does the right thing**, not how it does it
  - *contract*: input and output to the unit
  - Do *not* test how the work is performed
- Tests ensures that the **code meets expectations**
  - and continues to meet expectations over time

Why unit test?

- Faster debugging
- Faster development
- Better design
- Detect regressions (when you introduce a bug)
- Reduce maintenance costs
- Provides a living documentation of the system
- Refactoring your code and know that you have not broken anything

Why unit testing is not common

I don't have time to write tests because I am too busy debugging

- **Extra work**: Once I debug my code I have to write test code
  - Unit tests help you debug more quickly
  - Write tests as you write your code
- **I tested my code, it works**:
  - We manually test and throw the test away
  - We test, just not in a way that is reproducible

Why unit testing is not common

- **Writing tests is slow**:
  - Adding and removing print statements is slow
  - Manually retest, over and over again. This is very slow
- **I am too busy tracking down a bug**:
  - Future you does not test all aspects and bugs are created
  - Future bugs that are hard to track down.
- **Do not know how to unit test**:
  - It is simple, I will show you.

testthat

Introduction

- Developed by Hadley Wickham
- testthat: Get Started with Testing
  - The R Journal, vol. 3, no. 1, pp. 5–10, 2011
- Well integrated into RStudio and the tidyverse

Hierarchical structure

The structure of tests are hierarchiral

- **Context**: group tests together by related functionality
- **Tests**: group expectations together
- **Expectations**: describe what the result of a computation should be

Context

- Groups a set of tests that are related by a functionality
- Generally, one `context()` per test file
- Generally, you have one for each class or set of related functions

Command: `testthat::context(desc)`

Example: `testthat::context("Joining strings")`

Tests

- Groups related expectations together
- Check variations of the expectation

Command: `testthat::test_that(desc, code)` Example:

```
test_that("basic case works", {
  test ← c("a", "b", "c")

  expect_equal(str_c(test), test)
  expect_equal(str_c(test, sep = " "), test)
  expect_equal(str_c(test, collapse = ""), "abc")
})
```

Expectations

- The heart of the system. Simple test of the expectations
- Describes the explicit result of a computation
  - does it have the right value, class, length, or throw an exception, warning, message
- starts with `expect_`
- There are two methods families of expectations

1. `expect_that()` - old school
2. `expect_CONDITION()` - new school. `CONDITION` is a label

Expectation Families

- There are several families of expectations built around `expect_that`

comparison

`expect_lt()`, `expect_lte()`, `expect_gt()`, `expect_gte()`

Equality

`expect_equal()`, `expect_equivalent()`, `expect_identical()`

Expectation Families

Length

`expect_length()`

RegEx matching

`expect_match()`

Output

`expect_output()`, `expect_output_file()`, `expect_error()`, `expect_message()`, `expect_warning()`, `expect_silent()`

Expectation Families

Inheritance

`expect_null()`, `expect_type()`, `expect_is()`, `expect_s3_class()`, `expect_s4_class()`

Logical

`expect_true()`, `expect_false()`

Reference file / object

`expect_equal_to_reference()`

Expectation Families

Expectations that will always fail or succeed

`fail()`, `succeed()`

Check the names of an object

`expect_named()`

Putting it all together

```
context("Joining strings")

test_that("basic case works", {
  test ← c("a", "b", "c")

  expect_equal(str_c(test), test)
  expect_equal(str_c(test, sep = " "), test)
  expect_equal(str_c(test, collapse = ""), "abc")
  rm(test)
})

test_that("NULLs are dropped", {
  # more expectations
})
```

Setup and Tear down

- `testthat` does not tear down your environment
- Do not change your environment!
- Setup and clean up your environment manually
  - Clean-up side-effects (writing a file, remove vars)

```
test_that("basic case works", {
  test ← c("a", "b", "c")  # setup
  expect_equal(str_c(test), test) # expectation
  rm(test) # tear down
})
```

Setup the testing environment

```
usethis::use_test('my-test')

#> ✔ Adding 'testthat' to Suggests field in DESCRI
#> ✔ Creating 'tests/testthat/'
#> ✔ Writing 'tests/testthat.R'
#> ✔ Writing 'tests/testthat/test-my-test.R'
```

- Test files are in `test/testthat/`
- filename must start with `test`

Test Workflow Cycle

1. Write tests and code
   - `usethis::use_test()` creates a new test file
2. Running tests (choose one)
   - Ctrl/Cmd-shift-t
   - `devtools::test()`
   - `devtools::check()` runs tests and does other things
   - `R CMD check`
3. Fix bugs
4. Repeat

Test Output

- Number of passed, failed, warnings and skipped tests
- One line per context
- Label is what you put in the context.

```
Testing stringr
✔ | OK F W S | Context
✔ |  4       | case
✔ |  1       | conv
✔ |  4       | Counting matches
✖ | 34 1     | Replacements
```

Failure message:

Error messages printed below the context.

One message for each error

```
test-replace.r:7: failure: basic replacement works
str_replace("abababa", "ba", "BA") not equal to "a
1/1 mismatches
x[1]: "aBAbaba"
y[1]: "aBAbabaBAD"
```

Demo

Thanks!