Unit tests workshop

Kamil Foltyński

11/04/2019

Contents

Agenda

- Intro
  - Clone repo:

    ```
    git clone git@github.com:Tazovsky/unit-tests-workshop.git
    ```

  - Install helper R package:

    ```
    devtools::install("unitTestsWorkshopPackage/")
    ```

- Why do unit testing?
- Correct R package structure
- Exceptions
- Reference data
- Test driven development
- How to test?

Why do unit testing?

1. Existing unit tests ensures that implemented changes, improvements, fixes doen't break anything that already exists and works
   - Fixing specific feature may result in breaking some other functionality
2. Saves time and reduces costs
   - no need/reduced need for manual testing of code results
3. Makes refactoring much easier and faster and reduces risk
   - sometimes some old feature needs to be redesigned and changing already working and tested code is risky, but when unit test are written it's much easier because we know what to expect, e.g. sepcific inputs, outputs or variable types
4. With unit tests we get bonus: *documentation* :)
   - *documentation*, but not literally - what I mean is that when we look at unit tests code we can easily learn how some function or whole feature works, what input it accepts, what is the output, what *edge cases* we can expect, what we need to be careful of, etc.
5. Debug faster
   - Well unit tested code is a boost in debugging - when last commit breaks test(s) it means last code change should be investigated first. Without unit tests we might not even know that something was broken before last commit and it is just coincidence we noticed that now

Sources

- https://dzone.com/articles/top-8-benefits-of-unit-testing

`testthat` - R package for unit testing

Package `testthat` provides framework and multiple functions for unit testing. To start using it we need following R pacakge structure:

```
spendworx
  DESCRIPTION
  NAMESPACE
  R
  inst
  man
  spendworx.Rproj
  tests
      testthat # <-- tests goes there
         test_sankey_plot.R
      testthat.R
```

where `tests` directory and its content can be created using `devtools::use_testthat()` if not exists. Test script names must start with **`test`**.

Test files should contain `testthat::context` which is short decription of its content. To create test we use `testthat::test_that` function.

Example test:

```r
testthat::context("String length")

testthat::test_that("str_length is number of characters", {
  testthat::expect_equal(stringr::str_length("a"), 1)
  testthat::expect_equal(stringr::str_length("ab"), 2)
  testthat::expect_equal(stringr::str_length("abc"), 3)
})
```

How to test R package

- All tests in package: `devtools::test("unitTestsWorkshopPackage/")`
- Single test file: `testthat::test_file("unitTestsWorkshopPackage/tests/testthat/test_unify_case_type.R")`

Make life easier in RStudio

Shortcuts:

Run tests: `Ctrl+Shift+T` (Windows/Linux) or `Cmd+Shift+T` (Mac)

Typical RStudio workflow:

- Write function
- If reference data from file is used then rebuild package (`Ctrl+Shift+B`), see more in reference data part

- Run tests: `Ctrl+Shift+T`
  - if tests passed then continue
  - else make fixes in function and run tests `Ctrl+Shift+T`

...and repeat until all unit tests work.

`testthat` functions

- `testthat::expect_true(object/contition)`
- `testthat::expect_equal(object, expected)`
- `testthat::expect_indentical(object, expected)`

```r
testthat::test_that("iris as data.table is equal", {

  dt_iris <- data.table::data.table(iris)

  # function to be tested
  make_data.table <- function(x) {
    data.table::data.table(x)
  }

  testthat::expect_identical(dt_iris, make_data.table(iris))
  testthat::expect_equal(dt_iris, make_data.table(iris))

})
```

- `testthat::skip_if`, `testthat::skip_if_not`
  - tests can be skipped, for example some tests may not be valid on specific envornment:
    - **example #1**: if we running code locally we may not have access to production DB and we use local one which has different type, e.g. is containerized
    - **example #2**: locally LDAP authetntication doesn't work, so we can't test reading login anywhere else than on server
- `testthat::capture_messages` - messages or logs printed to terminal can also be tested

Practice

Task #1

- clone example repo `https://github.com/Tazovsky/unit-tests-workshop`
- create `testthat` structure

Task #2

- Test log output of function `unitTestsWorkshopPackage::unify_case_type`

Task #3

Use `testthat::expect_equal` and `testthat::expect_identical` to test function `unitTestsWorkshopPackage::coerce2da` **but** save expected data to `rds` file and compare it only after reading `readRDS`.

Sources

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

What is exception?

> Definition: An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

Why should we care?

Let's see example first:

```
library(dplyr)
>
> Attaching package: 'dplyr'
> The following objects are masked from 'package:stats':
>
>     filter, lag
> The following objects are masked from 'package:base':
>
>     intersect, setdiff, setequal, union
# prepare test data
df <- mtcars
df$car <- row.names(mtcars)
row.names(df) <- c()
df_hp <- df[, c("car", "hp")]
df <- df[!names(df) %in% "hp"]
df1 <- df[1:16, ]
df2 <- df[17:nrow(df), ]
df.list <- list(df1, df2)

head(df1, 2)
>   mpg cyl disp drat    wt  qsec vs am gear carb           car
> 1  21   6  160  3.9 2.620 16.46  0  1    4    4     Mazda RX4
> 2  21   6  160  3.9 2.875 17.02  0  1    4    4 Mazda RX4 Wag

head(df2, 2)
>     mpg cyl  disp drat    wt  qsec vs am gear carb               car
> 17 14.7   8 440.0 3.23 5.345 17.42  0  0    3    4 Chrysler Imperial
> 18 32.4   4  78.7 4.08 2.200 19.47  1  1    4    1          Fiat 128

head(df_hp, 2)
>             car  hp
> 1     Mazda RX4 110
> 2 Mazda RX4 Wag 110

# declare fun to convert get N top row from data frames in argument list
subsetTopNAndJoinDataTOresult <- function(list_data, data2join, key, top_n_rows = 3) {

  subsetTopN <- function(input, n) {
    lapply(input, function(x) head(x, n))
  }

  result <- subsetTopN(list_data, top_n_rows)

  # join horse power to each element of list
  lapply(result, function(x) {
    x %>% left_join(data2join, by = key)
  })
}

result <- subsetTopNAndJoinDataTOresult(list_data = df.list, data2join = df_hp, key = "car")
result # everything went well :)
> [[1]]
>    mpg cyl disp drat    wt  qsec vs am gear carb           car  hp
> 1 21.0   6  160 3.90 2.620 16.46  0  1    4    4     Mazda RX4 110
> 2 21.0   6  160 3.90 2.875 17.02  0  1    4    4 Mazda RX4 Wag 110
> 3 22.8   4  108 3.85 2.320 18.61  1  1    4    1    Datsun 710  93
>
> [[2]]
>    mpg cyl  disp drat    wt  qsec vs am gear carb               car  hp
> 1 14.7   8 440.0 3.23 5.345 17.42  0  0    3    4 Chrysler Imperial 230
> 2 32.4   4  78.7 4.08 2.200 19.47  1  1    4    1          Fiat 128  66
> 3 30.4   4  75.7 4.93 1.615 18.52  1  1    4    2       Honda Civic  52

# ...but imagine that list_data argument is not even a list - it is just ONE data frame
result2 <- subsetTopNAndJoinDataTOresult(list_data = df1, data2join = df_hp, key = "hp")
> Error in UseMethod("left_join"): niestosowalna metoda dla 'left_join' zastosowana do obiektu klasy "c
```

The problem is that `df` is both `data.frame` and `list`:

```
df <- mtcars
is.data.frame(df)
> [1] TRUE
is.list(df)
> [1] TRUE
```

and `lapply` function's `X` argument is coerced by `base::as.list`, so output from `subsetTopN` is list and is being joined to `data.frame` named `df_hp`:

```
lapply(mtcars[1:3], function(x) print(head(x, 2)))
> [1] 21 21
> [1] 6 6
> [1] 160 160
> $mpg
> [1] 21 21
>
> $cyl
> [1] 6 6
>
> $disp
> [1] 160 160
```

Exception handling

The above code's problem is that it neither verifies any of input data nor edge cases. What can we do is to try to **handle exception**. For example:

```r
library(dplyr)
# prepare data
df <- mtcars
df$car <- row.names(mtcars)
row.names(df) <- c()
df_hp <- df[, c("car", "hp")]
df <- df[!names(df) %in% "hp"]
df1 <- df[1:16, ]
df2 <- df[17:nrow(df), ]
df.list <- list(df1, df2)

subsetTopNAndJoinDataTOresult <- function(list_data, data2join, key, top_n_rows = 3) {

  subsetTopN <- function(input, n) {
    lapply(input, function(x) {

      if (!is.data.frame(x))
        stop(sprintf("'x' argument is not data.frame class but %s", class(x)))

        head(x, n)
    })
  }

  result <- subsetTopN(list_data, top_n_rows)

  # join horse power to each element of list
  lapply(result, function(x) {
    x %>% left_join(data2join, by = key)
  })
}

result <- subsetTopNAndJoinDataTOresult(list_data = df1, data2join = df_hp, key = "car")
> Error in FUN(X[[i]], ...): 'x' argument is not data.frame class but numeric
```

There are also many ways to handle exceptions:

1. are all needed arguments provided?
2. are variables and functions's arguments the expected type/class?
3. if variables' types match expected, then if variables provide enough informations? e.g.:

```r
df <- data.frame()
is.data.frame(df)
> [1] TRUE
nrow(df) > 0
> [1] FALSE
```

Answers

But initial question was ***why should we care about exceptions?***, so let's try to answer:

1. We have **more control over the code** behaviour. We can notice problem quicker, for example by verifying quality of function's arguments in first lines of its body
2. Handling exceptions **saves time on debugging**:

Error message: `'x' argument is not data.frame class but numeric**_`

is more understandable than

`Error in UseMethod("left_join"): niestosowalna metoda dla 'left_join' zastosowana do obiektu klasy "c('double', 'numeric')"`,

isn't it? :)

Testing exceptions

Exceptions and error messages can also be tested. And again it is about code behaviour control. By testing exceptions we expect that code will throw **specific error** in specific case. For example, we can expect that on error further code execution will stop and won't insert do SQL database data frame full of thousands of `NA`s :)

Example:

```r
testthat::context("subsetTopNAndJoinDataTOresult")
testthat::test_that("throws error if input is 1 data frame", {

  df <- mtcars
  df$car <- row.names(mtcars)
  row.names(df) <- c()
  df_hp <- df[, c("car", "hp")]
  df <- df[!names(df) %in% "hp"]

  testthat::expect_error(
    unitTestsWorkshopPackage::subsetTopNAndJoinDataTOresult(list_data = df, data2join = df_hp, key = "ca
    regexp = ".*'x' argument is not data\\.frame class but numeric.*"
  )
})
```

Practice

Task 1

Suggest validation of `unitTestsWorkshopPackage::subsetTopNAndJoinDataTOresult` arguments

Task 2

Handle and test exceptions when:

- `data2join` have 0 rows
- `list_data` has 0 length

Task 3

Write unit test for `unitTestsWorkshopPackage::subsetTopNAndJoinDataTOresult`

What is reference data? And why should we care?

Along with development process, code changes, but its behaviour and results should not - and if they must change, let them change into a way we control.

Reference data is used as input for unit tests. It's frozen, so it ensures **reproducibility** of results. During development process **code changes**, but thanks to **reproducibility** it is possible to run code (with reference data as input) and **always get same, unchanged results**. However if results vary, then it's a sign something went wrong and code is probably broken.

How to prepare reference data?

- `dput` - the easiest way to prepare reference data is to use `dput` function. It prints object's structure to terminal, so it is just about copy-pasting from there to our test script, see example:

```
library(dplyr)
result <- band_members %>% inner_join(band_instruments)
> Joining, by = "name"
dput(result)
> structure(list(name = c("John", "Paul"), band = c("Beatles",
> "Beatles"), plays = c("guitar", "bass")), row.names = c(NA, -2L
> ), class = c("tbl_df", "tbl", "data.frame"))
```

so unit test can be witten as:

```
library(dplyr)
ref_data <- structure(
  list(
    name = c("John", "Paul"),
    band = c("Beatles",
             "Beatles"),
    plays = c("guitar", "bass")
  ),
  row.names = c(NA, -2L),
  class = c("tbl_df", "tbl", "data.frame")
)

testthat::expect_equal(band_members %>% inner_join(band_instruments), ref_data)
> Joining, by = "name"
testthat::expect_identical(band_members %>% inner_join(band_instruments), ref_data)
> Joining, by = "name"
```

BTW: `dput` is also recommended way to provide Minimal Working Example (MWE) when asking on stackoverflow.com.

- `.rds`/`.RData` - `dput` is useful form small *handy* data, but imagine reference `data.frame` counting thousands of rows and pasted into test script...

  Click to expand

```
dput(iris)
> structure(list(Sepal.Length = c(5.1, 4.9, 4.7, 4.6, 5, 5.4, 4.6,
> 5, 4.4, 4.9, 5.4, 4.8, 4.8, 4.3, 5.8, 5.7, 5.4, 5.1, 5.7, 5.1,
> 5.4, 5.1, 4.6, 5.1, 4.8, 5, 5, 5.2, 5.2, 4.7, 4.8, 5.4, 5.2,
> 5.5, 4.9, 5, 5.5, 4.9, 4.4, 5.1, 5, 4.5, 4.4, 5, 5.1, 4.8, 5.1,
> 4.6, 5.3, 5, 7, 6.4, 6.9, 5.5, 6.5, 5.7, 6.3, 4.9, 6.6, 5.2,
> 5, 5.9, 6, 6.1, 5.6, 6.7, 5.6, 5.8, 6.2, 5.6, 5.9, 6.1, 6.3,
> 6.1, 6.4, 6.6, 6.8, 6.7, 6, 5.7, 5.5, 5.5, 5.8, 6, 5.4, 6, 6.7,
> 6.3, 5.6, 5.5, 5.5, 6.1, 5.8, 5, 5.6, 5.7, 5.7, 6.2, 5.1, 5.7,
> 6.3, 5.8, 7.1, 6.3, 6.5, 7.6, 4.9, 7.3, 6.7, 7.2, 6.5, 6.4, 6.8,
> 5.7, 5.8, 6.4, 6.5, 7.7, 7.7, 6, 6.9, 5.6, 7.7, 6.3, 6.7, 7.2,
> 6.2, 6.1, 6.4, 7.2, 7.4, 7.9, 6.4, 6.3, 6.1, 7.7, 6.3, 6.4, 6,
> 6.9, 6.7, 6.9, 5.8, 6.8, 6.7, 6.7, 6.3, 6.5, 6.2, 5.9), Sepal.Width = c(3.5,
> 3, 3.2, 3.1, 3.6, 3.9, 3.4, 3.4, 2.9, 3.1, 3.7, 3.4, 3, 3, 4,
> 4.4, 3.9, 3.5, 3.8, 3.8, 3.4, 3.7, 3.6, 3.3, 3.4, 3, 3.4, 3.5,
> 3.4, 3.2, 3.1, 3.4, 4.1, 4.2, 3.1, 3.2, 3.5, 3.6, 3, 3.4, 3.5,
> 2.3, 3.2, 3.5, 3.8, 3, 3.8, 3.2, 3.7, 3.3, 3.2, 3.2, 3.1, 2.3,
> 2.8, 2.8, 3.3, 2.4, 2.9, 2.7, 2, 3, 2.2, 2.9, 2.9, 3.1, 3, 2.7,
> 2.2, 2.5, 3.2, 2.8, 2.5, 2.8, 2.9, 3, 2.8, 3, 2.9, 2.6, 2.4,
> 2.4, 2.7, 2.7, 3, 3.4, 3.1, 2.3, 3, 2.5, 2.6, 3, 2.6, 2.3, 2.7,
> 3, 2.9, 2.9, 2.5, 2.8, 3.3, 2.7, 3, 2.9, 3, 3, 2.5, 2.9, 2.5,
> 3.6, 3.2, 2.7, 3, 2.5, 2.8, 3.2, 3, 3.8, 2.6, 2.2, 3.2, 2.8,
> 2.8, 2.7, 3.3, 3.2, 2.8, 3, 2.8, 3, 2.8, 3.8, 2.8, 2.8, 2.6,
> 3, 3.4, 3.1, 3, 3.1, 3.1, 3.1, 2.7, 3.2, 3.3, 3, 2.5, 3, 3.4,
> 3), Petal.Length = c(1.4, 1.4, 1.3, 1.5, 1.4, 1.7, 1.4, 1.5,
> 1.4, 1.5, 1.5, 1.6, 1.4, 1.1, 1.2, 1.5, 1.3, 1.4, 1.7, 1.5, 1.7,
> 1.5, 1, 1.7, 1.9, 1.6, 1.6, 1.5, 1.4, 1.6, 1.6, 1.5, 1.5, 1.4,
> 1.5, 1.2, 1.3, 1.4, 1.3, 1.5, 1.3, 1.3, 1.3, 1.6, 1.9, 1.4, 1.6,
> 1.4, 1.5, 1.4, 4.7, 4.5, 4.9, 4, 4.6, 4.5, 4.7, 3.3, 4.6, 3.9,
> 3.5, 4.2, 4, 4.7, 3.6, 4.4, 4.5, 4.1, 4.5, 3.9, 4.8, 4, 4.9,
> 4.7, 4.3, 4.4, 4.8, 5, 4.5, 3.5, 3.8, 3.7, 3.9, 5.1, 4.5, 4.5,
> 4.7, 4.4, 4.1, 4, 4.4, 4.6, 4, 3.3, 4.2, 4.2, 4.2, 4.3, 3, 4.1,
> 6, 5.1, 5.9, 5.6, 5.8, 6.6, 4.5, 6.3, 5.8, 6.1, 5.1, 5.3, 5.5,
> 5, 5.1, 5.3, 5.5, 6.7, 6.9, 5, 5.7, 4.9, 6.7, 4.9, 5.7, 6, 4.8,
> 4.9, 5.6, 5.8, 6.1, 6.4, 5.6, 5.1, 5.6, 6.1, 5.6, 5.5, 4.8, 5.4,
> 5.6, 5.1, 5.1, 5.9, 5.7, 5.2, 5, 5.2, 5.4, 5.1), Petal.Width = c(0.2,
> 0.2, 0.2, 0.2, 0.2, 0.4, 0.3, 0.2, 0.2, 0.1, 0.2, 0.2, 0.1, 0.1,
> 0.2, 0.4, 0.4, 0.3, 0.3, 0.3, 0.2, 0.4, 0.2, 0.5, 0.2, 0.2, 0.4,
> 0.2, 0.2, 0.2, 0.2, 0.4, 0.1, 0.2, 0.2, 0.2, 0.2, 0.1, 0.2, 0.2,
> 0.3, 0.3, 0.2, 0.6, 0.4, 0.3, 0.2, 0.2, 0.2, 0.2, 1.4, 1.5, 1.5,
> 1.3, 1.5, 1.3, 1.6, 1, 1.3, 1.4, 1, 1.5, 1, 1.4, 1.3, 1.4, 1.5,
> 1, 1.5, 1.1, 1.8, 1.3, 1.5, 1.2, 1.3, 1.4, 1.4, 1.7, 1.5, 1,
> 1.1, 1, 1.2, 1.6, 1.5, 1.6, 1.5, 1.3, 1.3, 1.3, 1.2, 1.4, 1.2,
> 1, 1.3, 1.2, 1.3, 1.3, 1.1, 1.3, 2.5, 1.9, 2.1, 1.8, 2.2, 2.1,
> 1.7, 1.8, 1.8, 2.5, 2, 1.9, 2.1, 2, 2.4, 2.3, 1.8, 2.2, 2.3,
> 1.5, 2.3, 2, 2, 1.8, 2.1, 1.8, 1.8, 1.8, 2.1, 1.6, 1.9, 2, 2.2,
> 1.5, 1.4, 2.3, 2.4, 1.8, 1.8, 2.1, 2.4, 2.3, 1.9, 2.3, 2.5, 2.3,
> 1.9, 2, 2.3, 1.8), Species = structure(c(1L, 1L, 1L, 1L, 1L,
> 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L,
> 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L,
> 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 2L, 2L, 2L,
> 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L,
> 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L,
> 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 3L,
> 3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L,
> 3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L,
> 3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L,
> 3L), .Label = c("setosa", "versicolor", "virginica"), class = "factor")), class = "data.frame", r
> -150L))
```

In such cases it's worth keeping reference data in R formats such as `.rds`/`.RData`.

But when we work with R package test data should always go with package, because when installing R package on multiple environments we can run unit tests to see whether everything works properly or not. **So where test data should be stored**? Every package is installed together with `inst` directory. The recommended location for test data is `inst/testdata` directory in R package.

To get path to reference data `system.file` functionshould be used:

```r
ref_data <- system.file(file.path("testdata", "test_data.rds"), package = "SpendWorx")
testthat::expect_equal(functionTotest(), ref_data)
```

- `snapshot/screenshot` - sometimes we want to would like to test output from plotting function. The simplest way to do that is comparing screenshots of plot:

```r
testthat::context("Testing plot(s) output")

testthat::test_that("plotly renders correctly", {
library(plotly)

target.path <- file.path("inst", "testdata", "plot1.png")

p <- economics %>% plot_ly(x = ~date, y = ~unemploy/pop) %>% add_lines()

# compareImages function already contains testing function testthat::expect_lte
unitTestsWorkshopPackage::compareImages(takeScreenshot(p), target.path)
})
```

Recreating reference data

Along with development process, code changes, but its behaviour and results should not - and ***if they must change, let them change into a way we control****.*

Code is being improved, it evolves and there will always be a moment when results, outputs change (but we konw about that, **it is a conscious change**), so reference data must be updated. I really like approach where developer puts few lines of **code to easily recreate reference data**.

For example, if we go back to plotly unit test, the whole test could look like this:

```r
  testthat::context("Testing plot(s) output")

  testthat::test_that("plotly renders correctly", {
    library(plotly)

    target.path <- system.file(file.path("testdata", "plot1.png"), package = "unitTestsWorkshopPackage"

    p <- economics %>% plot_ly(x = ~date, y = ~unemploy/pop) %>% add_lines()

    # compareImages function already contains testing function testthat::expect_lte
    unitTestsWorkshopPackage::compareImages(takeScreenshot(p), target.path)

    # following code recreates reference screenshot
    if (exists("RECREATE_REF_DATA") && isTRUE(RECREATE_REF_DATA)) {

      message("Recreating plotly reference screenshot...")

      testdata_dir <- file.path("unitTestsWorkshopPackage", "inst", "testdata")

      if (!dir.exists(file.path(testdata_dir)))
        dir.create(file.path(testdata_dir), recursive = TRUE)

      target.path <- file.path(testdata_dir, "plot1.png")

      unitTestsWorkshopPackage::takeScreenshot(p, target.path)
    }
  })
```

so to recreate reference screenshot just set `RECREATE_REF_DATA` to `TRUE` and run test.

Practice

Task #1

Test output of function `unitTestsWorkshopPackage::renderBoxplot`.

- Use following code as input data:

```r
dat <- data.frame(xval = sample(100, 1000, replace = TRUE),
                  group = as.factor(sample(c("a", "b", "c"), 1000, replace = TRUE)))
```

- Use `unitTestsWorkshopPackage::takeScreenshot` to prepare reference plot
- Use function `unitTestsWorkshopPackage::compareImages` to test output

What is Test Driven Development (TDD)?

It is development technique where new feature starts with wirting unit tests, so development cycle is:

1. Write new unit tests (`tests/testthat/test_new_feature.R`)
2. Run all tests (including old ones) and make sure that new tests fail (`Ctrl+Shift+T` or `devtools::test()`)
3. Write feature code (`R/new_feature.R`) and verify whether tests pass (`Ctrl+Shift+T` or `devtools::test()`)
4. If tests pass, new feature's code met requirements and did not break any old features, else new code must be improved (fix code in `R/new_feature.R`) and **cycle repeated** (start from `#1` again)

Why/when to use?

This technique requires developer to understand feature's requirement and specification.

Real life example(s)

- SpendWorx case: when we were moving data from `.csv` file to `PostgreSQL` database. We knew exactly what were the requirements, be we knew output data
- SpendWorx case: when refactoring code to fit Shiny modules - again we know what are requirements both from UI and server sides

Why it is not often used in practice?

- Fast growing apps are usually focused on effect. In such apps some time we don't know exactly what are the requirements, because important part of development process is research and testing best *made-to-measure* solutions.
- To much `TDD` makes our life harder. It may happen that we will have too many *high-level*, too detailed tests which may mean that our test code will require as much maintenance as production code.
- Onbarding of new developers may cost more - especially when they are less experienced

How to write unit tests?

- I really encourage to write unit tests just after new code is written or while writing it. New feature is still *warm*, *fresh*, generally well known for developer and developer probably encountered some edge cases/expceptions. If developer postpone writing tests for few days, then all of those will be probably forgotten (partially at least)
- Split big functionalities into separate tests, e.g. 1st test: test connection to database, 2nd test: test function's output on prepared reference data from database
- Take care of **reproducibility** - prepare tests so that the tested feature result can be repeated, especially focus on preparging correct reference (expected) test data
- Avoid test interdependence - each should be able to run independently - *on its own*; imagine that tests can be run in parallel and should interfere with each other
- Follow the **KISS** principle (**K**eep **I**t **S**hort and **S**imple) - keep tests as simple and clean as possible to help other developers easily find out how tested feature works; there is also no need for making tests flexible, for example output, variable can be hard coded

How to measure tests coverage?

We can use `covr` R package:

```
code_coverage_summary <- covr::package_coverage("unitTestsWorkshopPackage/")
code_coverage_summary
> unitTestsWorkshopPackage Coverage: 88.33%
> R/unify_case_type.R: 77.42%
> R/screenshot.R: 81.08%
> R/coerce2data.table.R: 100.00%
> R/renderBoxplot.R: 100.00%
> R/subsetTopNAndJoinDataTOresult.R: 100.00%
```

To see nice report in `DT`:

```
covr::report(code_coverage_summary)
```

There are also many CI/CD solutions online to track test coverage:

- `coveralls` - see example: https://coveralls.io/github/Tazovsky/cachemeR?branch=devel
- `codecov`

Practice

Task #1

1. Type `git fetch origin master` and then `git checkout master`
2. In core repo directory run `covr::report(package_coverage("unitTestsWorkshopPackage/"))`
3. Try to write unit test to insrease `unitTestsWorkshopPackage` test coverage to 100%

Source code

Repository with code and examples can be found here:

`https://github.com/Tazovsky/unit-tests-workshop`