Question: Software Testing

This homework sheet will test your knowledge of debugging and software testing methods in R.

10

a) Read the following statements. Identify which are wrong and then correct these.

1. For unit tests, erroneous code is replaced by stubs or mocks.
2. Whenever an exception occurs, the underlying code has a bug.
3. A test suite contains unit tests.
4. Tests are used to ensure that software requirements are met.
5. In R, asserts are used to handle exceptions.
6. Integration tests are used to serve as acceptance tests for end-users.
7. A bug is a piece of code that does not behave as intended.
8. A code coverage of 100 % guarantees a software without bugs.
9. Black box testing helps identifying errors in the code.
10. Unit tests help to avoid software bugs.

3

b) Explain how the usage of code coverage is both a good and a bad metric for unit testing (3–5 sentences).

4

c) Inspect the following code of the function `abs_sum` which sums the elements of a vector. If the first value is `TRUE` it takes the absolute of each number. If it is `FALSE`, the absolute is not applied. Run the function using different vectors as arguments and use the debugger to find out why the results are not as expected.

```r
abs_sum <- function(vector)
{
  if (vector[1] == TRUE)
  {
    return(sum(abs(vector)))
  }
  else if (vector[1] == FALSE)
  {
    return(sum(vector))
  }
  else
  {
    stop("First element must be logical")
  }
}
```

9

d) Look at the following VAT calculation functions and write several unit tests for them. Note: for this, you need to create a testing infrastructure as described in the slides.

```
authenticate <- function()
{
  # this function simulates slow internet authentication
  Sys.sleep(2)
}

calculate_gross <- function(net_price)
{
  authenticate()
  if (!is.numeric(net_price))
  {
    stop("Input type is not numeric")
  }
  return(round(net_price*1.19, digits = 2))
}

calculate_net <- function(gross_price)
{
  authenticate()
  if (!is.numeric(gross_price))
  {
    stop("Input type is not numeric")
  }
  return(round(gross_price/1.19, digits = 2))
}
```

9

e) Look at the following VAT calculation functions and write several unit tests for them. Note: for this, you need to create a testing infrastructure as described in the slides.

```r
authenticate <- function()
{
  # this function simulates slow internet authentication
  Sys.sleep(2)
}

calculate_gross <- function(net_price)
{
  authenticate()
  if (!is.numeric(net_price))
  {
    stop("Input type is not numeric")
  }
  return(round(net_price*1.19, digits=2))
}

calculate_net <- function(gross_price)
{
  authenticate()
  if (!is.numeric(gross_price))
  {
    stop("Input type is not numeric")
  }
  return(round(gross_price/1.19, digits=2))
}
```

9

f) Load the following function named `classify`. This function returns a boolean value whether a credit should be granted given three input parameters.

```r
classify <- function(age, income, other_credits) {
  if (!is.numeric(age)) {
    stop("Invalid age paramter, expected numeric")
  } else if (age < 0 || age > 100) {
    stop("Invalid age parameter: " + age + " should be between ↩
       0 and 100")
  }
  if (!is.numeric(income)) {
    stop("Invalid income parameter, expected numeric")
  } else if (income < 0) {
    stop("Invalid income parameter: " + income + " should be ↩
       positive")
  }
  if (!is.numeric(other_credits)) {
    stop("Invalid other_credits parameter, expected numeric")
  } else if (other_credits < 0) {
    stop("Invalid other_credits parameter: " + other_credits + ↩
       " should be positive")
  }
  if (income < 20000) {
    return(FALSE)
  } else if (income < 80000) {
    if (age < 20 || age > 70) {
       return(FALSE)
    } else {
       if (income - other_credits * 20000 >= 20000) {
         return(TRUE)
       } else {
         return(FALSE)
       }
    }
  } else {
    if (other_credits < round(income / 20000)) {
       return(TRUE)
    } else {
       return(FALSE)
    }
  }
}
```

Set up a testing environment with code coverage as described in the slides. Then write unit tests to achieve a code coverage of 100 %.

2

g) Use the debugger to find the error in the following for loop.

```
decimal <- 0
binary <- c(1, 1, 0, 1)
for (i in seq_along(binary)) {
  decimal <- decimal + 2^i * binary[i]
}
# 1101 in decimal should be 13
decimal

## [1] 22
```

2

h) Write sensible asserts to the following function. It prints the vector of ascending numbers as a string. Use any of the three suggested methods from the slides.

```r
print_vector_ascending <- function(vec) {
  print(paste(vec, collapse = " < "))
}

print_vector_ascending(1:5)

## [1] "1 < 2 < 3 < 4 < 5"
```

4

i) Look at the following code. The super secure `log_in` function returns an error if the username or password is not valid. Extend the code to let the user retry up to three times if the password is wrong and up to infinite times if the given name was wrong.

Hint: `grepl('word', 'word is in sentence')` returns true, because the first string is contained in the second string.

```r
log_in <- function(name, password) {
  if (!name %in% c('mike', 'sara', 'root')) {
    stop("Username does not exist")
  } else if (name == 'mike' && password == 'giraffe'
             || name == 'sara' && password == 'secur1ty'
             || name == 'root' && password == '1234') {
    cat(c("Succesfully logged in. Welcome ", name))
  } else {
    stop("Wrong password")
  }
}
# Extend this code
name <- readline(prompt="Enter your name: ")
password <- readline(prompt="Enter your password: ")
log_in(name, password)
```

3

j) In *test-driven development*, the unit tests are created before the actual function is implemented. This may seem counter-intuitive at first, but it is a useful tool to ensure requirements are met.

Write a function `mystery` for which the following test case would produce no errors.

Hint: you can use the test cases in the same file you are writing your function in, if you load the `testthat` library.

```r
test_that('function works', {
  expect_equal(mystery(0,8), 0)
  expect_equal(mystery(5,1), 5)
  expect_equal(mystery(-6,6), -36)
  expect_equal(mystery(2,2), 4)
  expect_equal(mystery(17,55), 935)

  expect_error(mystery(5))
  expect_error(mystery("7",5))
})
```