Software Testing

Algorithm Design & Software Engineering
February 10, 2016
Stefan Feuerriegel
Today’s Lecture

Objectives

1. Learning debugging strategies in R for finding bugs efficiently
2. Understanding approaches for testing software
3. Formalizing software requirements with the help of unit tests
Outline

1. Software Bugs
2. Debugging
3. Software Testing
4. Unit Testing
Outline

1. Software Bugs
2. Debugging
3. Software Testing
4. Unit Testing
Software Bugs

A software bug is an error or flaw that causes a program to behave in an incorrect or unintended way

Well-known examples

► Ariane 5 flight 501 exploded 40 seconds after launch destroying a $1 billion prototype rocket due to a number overflow
  → A 64-bit floating number was converted into a 16-bit integer without exception handling
► Year 2000 bug in which a worldwide collapse was feared
  → Years were stored as a two-digit number, making it indistinguishable from 1900
► The 2003 blackout in North America was caused by a race condition which was not handled

Bugs can have various reasons but different counter measures exist
Programming Bug

Example of a buggy code for calculating $n^k$

```
Power <- 0
for (i in 0:k) {
    power <- power * i
}
```

Question

▸ Which of the following appear as software bugs in the above snippet?
  ▸ Wrong initialization
  ▸ Wrong loop range
  ▸ Wrong variable naming
  ▸ Wrong variables in mathematical operation
  ▸ Overflow

▸ No Pingo available
Debugging and Software Testing

Tools to find and prevent bugs

1. Debugging
   - Locates the source for a programming flaw
   - Helps understanding program execution

2. Software testing
   - Standardized means for quality and correctness checks
   - Sometimes used for specifying requirements
   - Assessing the usability of program interfaces

Rule of thumb: debugging consumes about two thirds of the development
Debugging

- Debugging is recommended when the return value (e.g. of a unit test) is erroneous and the error itself is not obvious
- Tools for examining the control flow and values of variables
- Many programming environments support line-by-line execution debugging, where only one line of code at a time is executed

Debugging strategy

1. Realize that you have a bug
2. Reproduce/generate input values that cause the bug
3. Isolate the flawed component with a binary search
4. Fix it
5. Confirm its successful resolution using the previous input

→ When using unit testing: create an automated test
Debugging in R

Key debugging tools in R

1. **Output variables** to the screen
   → e.g. `print(...)` command or `browser()` for an interactive session

2. **Asserts** (mostly preventative)

3. **Exception handling**

4. **Using built-in commands in R**
   → e.g. `traceback()` for the call stack

5. **Interactive debugger** inside R Studio
Debugging with Print Commands

One commonly write certain values to the screen for manual inspection

▶ Show value of a single variable via `print(variable)`
▶ `print(...)` is necessary to work across all levels of the control flow

▶ Benefits
  ▶ Easy to use
  ▶ Quick implementation
  ▶ Can narrow down the location of bugs

▶ Shortcomings
  ▶ Manual checks necessary
  ▶ Identifies only the approximate location of bugs
  ▶ Cannot handle exceptions

▶ Often combined in practice with a toggle to turn on/off logging messages

▶ `browser()` switches instead to an interactive session at that point
Debugging with Print Commands

Example: if correct, the loop would print 5, 25 and 125

```r
n <- 5
k <- 3

power <- 0
for (i in 0:k) {
  power <- power * i
  print(power)  # print current value in each iteration
}

## [1] 0
## [1] 0
## [1] 0
## [1] 0

print(power)  # should be 5^3 = 125

## [1] 0
```
Asserts

Trigger a specific message when a condition is not satisfied

- **Signal an error** if something is wrong ("fail fast")
- **Syntax options**
  1. `stop(...)`
  2. `stopifnot(...)`
  3. Package `assertthat`

- **Benefits**
  - Makes code and errors **understandable** if something unexpected occurs
  - Easier debugging of functions for other users

- **Shortcomings**
  - Does not guarantee error-free functions
  - Does not avoid bugs directly

- **Often used to check type and range of input** to functions
Asserts

Example that checks input types and range

cube_root <- function(x) {
  if (class(x) != "numeric") {
    stop("Wrong variable class: not a single number")
  }
  if (x < 0) {
    stop("Wrong range: cannot be less than 0")
  }
  if (!is.finite(x)) {
    stop("Wrong range: cannot be infinite or NA")
  }
  return(x^(1/3))
}
cube_root("error")  # should throw an error

## Error in cube_root("error"): Wrong variable class: not a single number

cube_root(-5)  # should throw an error

## Error in cube_root(-5): Wrong range: cannot be less than 0

cube_root(NA)  # should throw an error

## Error in cube_root(NA): Wrong variable class: not a single number

cube_root(125)  # 5

## [1] 5

Testing: Debugging
Exception Handling

Exception handling (or condition handling) allows program to react upon (un)expected failures

- Functions can throw exceptions when an error occurs
- Code can then handle the exception and react upon it
- Syntax options: `try(...) and tryCatch(...)`

Benefits

- Program execution can continue even when errors are present
- Exception can trigger a designated response
- Helpful technique to interact with packages legacy code

Shortcomings

- Helps not to locate unexpected bugs
Exception Handling in R

▶ `try(...)` ignores an error

```r
f.unhandled <- function(x) {
  sqrt(x)
  return(x)
}
# no return value
f.unhandled("string")
## Error in sqrt(x): non-numeric argument to mathematical function

f.try <- function(x) {
  try(sqrt(x))
  return(x)
}
# skips error
f.try("string")
## [1] "string"
```

▶ Returns an object of `try-error` in case of an exception

```r
result <- try(2 + 3)
class(result)
## [1] "numeric"

inherits(result, "try-error")
## [1] FALSE

result
## [1] 5

error <- try("a" + "b")
class(error)
## [1] "try-error"

inherits(error, "try-error")
## [1] TRUE
```
Exception Handling in R

- `tryCatch(...)` can react differently upon errors, warnings, messages, etc. using handlers

```r
handle_type <- function(expr) {
  tryCatch(expr,
    error=function(e) "error",
    warning=function(e) "warning",
    message=function(e) "message"
  )
}
handle_type(stop("..."))
## [1] "error"

handle_type(warning("..."))
## [1] "warning"

handle_type(message("..."))
## [1] "message"

handle_type(10) # otherwise returns value of input
## [1] 10
```

- R allows to define custom exception types
Call Stack

The call stack shows the hierarchy of function calls leading to the error

- **Benefits**
  - Shows location of the error
  - Especially helpful with several, nested functions

- **Shortcomings**
  - Shows where an error occurred but not why
  - Works only for exceptions

- **R Studio usage:** click “Show Traceback” in R Studio

```text
Error in x + "string" : non-numeric argument to binary operator
```

- Show Traceback
- Rerun with Debug
Example: Call Stack in R

- Code including bug

```r
f <- function(x) g(x)
g <- function(x) x + "string"
f(0)
```

- Fired error message

```
## Error in x + "string": non-numeric argument to binary operator
```

- Display call stack manually with `traceback()`

```r
traceback()
```
```
## 2: f(0)
## 1: g(x)
```

First entry is the hierarchy level, followed by function name and possibly file name and line number.
Interactive Debugger in R Studio

Interactive debugging in R Studio allows line-by-line execution

▶ Benefits
  ▶ Helps finding the location of an error
  ▶ Makes it possible to track changes in the values of all variables

▶ Shortcomings
  ▶ Can be still time consuming to find location of a bug

▶ “Rerun with Debug”: repeats execution but stops at the exception

R Studio toolbar

▶ Requirements of R Studio: project, file saved, sourced, etc. → see further readings or website for details
Interactive Debugger in R Studio

- **Next** executes the `next` statement of up to the current hierarchy level
- **Step into** steps into the next function including a deeper hierarchy level
- **Step out** finishes current loop or function
- **Continue** continues execution to the end of the script
- **Stop** stops debugging and switches to the coding stage
- **Breakpoint** stops the execution at a pre-defined point for manual inspection

```r
power <- 1
for (i in 1:k) {
  power <- power * n
}
```

→ can be **conditional** together with an `if`
Debugging

Example: approximate the square root using Newton’s method

```r
n <- 2
dx <- 1
x.old <- NA
while ((x - x.old) >= 10e-5 || is.na(x.old)) {
  x.old <- x
  x <- 1/2 * (x + n/x)
}
x # should be 1.414214, i.e. large error
```

# [1] 1.416667

Question

- Which debugging strategy would you personally prefer?
  - Output variables
  - Asserts
  - Exception handling
  - Insights from call stack
  - Interactive debugger inside R Studio

- No Pingo available
Outline

1. Software Bugs
2. Debugging
3. Software Testing
4. Unit Testing
Software Testing

- Software testing studies the **quality** of a software
- Provides **standardized means and tailored tools** for testing
  → Opposed to simple “run-and-see”

**Reasons**

- External **proof-of-concept**
- Internal **quality assurance**
- Specifying the **requirements** and functionality of components

**Testing Scope**

- Functional (as specified in the requirements)
- Non-functional
  - Usability, graphical appearance
  - Scalability, performance
  - Compatibility, portability
  - Reliability
Testing Perspectives

Testing objectives vary dependent on the perspective

**End-users**
- Output must **match** expectations
- Internal code and structure not of relevance
- Mostly **black box testing**

**Developers**
- Program must handle all input correctly
- Intermediate values in the code must be correct
- Program needs to **work efficiently**
- Mostly **white box testing**

Testing can be
- **Static**: proofreading, reviews, verification, etc.
- **Dynamic**: automated unit tests, etc.
Black Box and White Box Testing

Software testing divided according to the knowledge of a tester

**Black box testing**

- Tests functionality without any knowledge of the implementation
- Observes the output for a given input
- Testers know what is supposed to come out but not how

**White box testing**

- Checks internal implementation of a program
- Tests are designed with knowledge of the code
- Usually automated, e.g. by unit tests
Levels of Testing

- Different level of testing checks the properties of a software
- A designated testing level corresponds to each stage of the waterfall model
- New approach is named V model
Acceptance and System Testing

Acceptance Testing
► Related to usability testing
► Concerns the interaction with users
► Tests e.g. the ease-to-use of the user interface

System Testing
► Performs end-to-end tests of the integrated system
► Tests mainly that requirements are met
Integration Testing

- Ensure the **correct interoperability of components**
- Thus tests **interfaces and interaction** above unit testing
- Above unit testing on the scale level, as interaction is tested
- **Common in large-scale** software projects
  - Example: Windows 7 was deployed daily on 1000+ different PCs to run automated tests

Regression Testing

- Aims is to find bugs after large code changes
- Checks for **unintended consequences of changes**
- Examples
  - Lost functionality
  - Depreciated features
  - Old bugs that reappeared
Unit Testing

Objectives

▶ Unit tests focus on the lowest level of a program
▶ Validates small code segments, e.g. a function or method
▶ Main use cases
  ▶ Ensure that code matches specification
  ▶ Detect bugs from changing or adding new code

Characteristics

▶ Each unit test usually consists of multiple simple comparisons
▶ Focus on boundary values of parameters
▶ Quick runtimes that allow automated checks after each code change
▶ Common quality metric is code coverage
Outline

1. Software Bugs
2. Debugging
3. Software Testing
4. Unit Testing
Unit Testing

- Refers to testing the functionality of a specific fragment
- Usually at function or class level
- Tests against pre-defined, expected outcomes

Reasons for using unit testing

- Fewer bugs because automated tests check functionality
- Designing of unit tests enforces better code structure
- Tracks progress of development
- Code becomes more robust since unit tests also control for side effects
- Tests help to document functionality
Unit Testing in R

**Package RUnit**
- Designed for unit testing
- Checks values and exceptions
- Generates text or HTML reports
- Limitation: no test stubs

**Package testthat**
- Supports unit testing, test stubs and test suites
- Generates text output, arbitrarily verbose
- Tests can be automated to run after each file change
- Intended for package development but also works well with simple R scripts

- Similar concepts and usage for both packages
- Code coverage measured for both through additional packages
Test Organization

Tests are organized hierarchically

▶ **Expectation** verifies a single assumption
  → Checks that given input values return the desired results

▶ **Tests** (or units) group several expectations
  → Tests a **single function** for a range of input values (including boundaries such as **NA**)

▶ **Suites** group several tests
  → In R, this is a simple file
  → For object oriented code, this tests a full class
Unit Testing in R

High-level procedure

1. Store function \( f \) subject to testing in \( f.R \)
2. Source that file via `source("f.R")`
3. Create file `test.f.R` that contains the tests
4. Write test, e.g.

   ```r
   test_that("Short description", {
     expect_equal(sum(1, 2, 3), 6)
   })
   ```

   where the description should continue “Test that …”
5. Load package `testthat`
6. Run file via `test_file("test.f.R")`, or all files in a directory via `test_dir(...)`
7. Assess results, i.e. failed tests
Unit Testing in R

Example calculates roots of quadratic equation \( x^2 + px + q \)

```r
roots_quadratic_eqn <- function(p, q)
{
  if (!is.numeric(p) || !is.numeric(q)) {
    stop("Wrong input format: expects numeric value")
  }
  return(c(-p/2 + sqrt((p/2)^2 - q),
            -p/2 - sqrt((p/2)^2 - q)))
}
```
Unit Testing in R

▶ Load testthat package

```r
library(testthat)
```

▶ Simple test file `test.roots_quadratic_eqn.R`

```r
test_that("Roots are numeric and correct", {
  r <- roots_quadratic_eqn(8, 7)
  expect_is(r, "numeric")
  expect_equal(length(r), 2)
  expect_equal(r, c(5, 6))
})
```

▶ Run tests to compare expected and real results of failed tests

```r
test_file("test.roots_quadratic_equation.R")
```

```
## ..1
## 1. Failure (at test.roots_quadratic_equation.R#5): Roots are numeric and correct
## r not equal to c(5, 6)
## 2/2 mismatches (average diff: 9.5).
## First 2:
## pos x y diff
## 1 -1 5 -6
## 2 -7 6 -13
```
Verifying Expectations

- **Syntax** `expect_*(actual, expected)` ensures expectations
- First argument is the **actual**, the second the **expected** result

Built-in expectation comparisons

- **`expect_equal`** checks for **equality within numerical tolerance**

```
expect_equal(1, 1)   # pass
expect_equal(1, 1 + 1e-8) # pass
expect_equal(1, 5)   # expectation fails
```

## Error: 1 not equal to 5
## 1 - 5 == -4

- **`expect_identical`** checks for **exact equality**

```
expect_identical(1, 1)  # pass
expect_identical(1, 1 + 1e-8) # expectation fails
```

## Error: 1 is not identical to 1 + 1e-08. Differences:
## Objects equal but not identical
Verifying Expectations

▶ **expect_true and expect_true** check for TRUE and FALSE value

```r
expect_true(TRUE)  # pass
expect_true(FALSE)  # expectation fails
```

```
## Error: FALSE isn’t true
```

```r
expect_true("str")  # expectation fails
```

```
## Error: "str" isn’t true
```

▶ **expect_is** checks the class type

```r
model <- lm(c(6:10) ~ c(1:5))
expect_is(model, "lm")  # pass
expect_is(model, "class")  # expectation fails
```

```
## Error: model inherits from lm not class
```

▶ **expect_error** checks that an error is thrown

```r
expect_error(0 + "str")  # pass since error was expected
expect_error(3 + 4)  # expectation fails because of no error
```

```
## Error: 3 + 4 code raised an error
```
Stubs and Mocks

- Some functions cannot be executed for testing purposes, e.g.
  - Functions that access different systems, e.g. online authentication
  - Persistent manipulations of databases
  - Hardware controlling functions, e.g. a robot arm
  - Execution of financial transactions, etc.
  - Functions dependency of non-existent code

- Solution: stubs and mocks
Stubs andMocks

Stubs

- The underlying operation is replaced by a stub for testing
- Stubs can perform primitive operations but usually return only a value

Mocks

- In OOP, replacements for full objects are called mock
- Mocks additionally check if methods were called as expected
Mocks in R

Example

- `calculate_gross(p)` calculates gross price for a VAT of 19 %

```r
calculate_gross <- function(net_price) {
  authenticate()  # External function call

  if (!is.numeric(net_price)) {
    stop("Input type is not numeric")
  }
  return(round(net_price*1.19, digits=2))
}
```

- Calls external service `authenticate()` to verify the access

```r
authenticate <- function() {
  library(RCurl)
  if (getURI("127.0.0.1") != "SUCCESS") {
    stop("Not authenticated")
  }
}
```

- `calculate_gross(p)` can be tested without authentication
  → Need a stub to skip or mimic functionality of `authenticate()`
Stubs in R

- Once can redirect the call `authenticate()` to a stub instead
- In this example, the stub skips authentication

```r
authenticate_stub <- function() {
  print("Authentication omitted for testing")
}
```

- Test file `test.calculate_gross.R`

```r
test_that('Gross calculation works correctly', {
  with_mock(authenticate = function() {
    print("Authentication omitted for testing")
  },
  expect_equal(calculate_gross(100), 119),
  expect_equal(calculate_gross(70), 83.30),
  expect_error(calculate_gross("str")),
  expect_error(calculate_gross("100.50"))
})
```

Note: the name `with_mock(...)` is misleading since this is not a mock but a stub
Stubs in R

- Run tests with mock

```r
test_file("test.calculate_gross.R")
```

```
## [1] "Authentication omitted for testing"
## .[1] "Authentication omitted for testing"
## .[1] "Authentication omitted for testing"
## .[1] "Authentication omitted for testing"
## .
## DONE
```

- Note: `authenticate(p)` needs to exist for `with_mock(...)` to work
Code Coverage

- Code coverage shows to which lines of code are tested
- Helps identifying non-tested code regions
- Usually measures coverage as ratio, e.g. 60% of all lines, functions, etc.
  - Warning: a high coverage does not guarantee thorough testing
- As a recommendation, focus especially on the boundaries of parameter ranges (0, NA, Inf, etc.) to identify unhandled problems

R package `covr`

- Supports only coverage when testing full packages
  - Workaround is to create a dummy package
Code Coverage in R

▶ Load devtools and covr

library(devtools)  # for creating packages
library(covr)      # for code coverage

▶ Create empty package testcovr in the current working directory

create("testcovr")  # create default structure
use_testthat("testcovr")  # append testing infrastructure

▶ Create sample absolute_value.R in folder testcovr/R/

absolute_value <- function(x) {
  if (x >= 0) {
    return(x)
  } else {
    return(-x)
  }
}

▶ Create test test.absolute_value.R in folder testcovr/tests/testthat/

test_that("absolute value is correct", {
  expect_is(absolute_value(-3), "numeric")
  expect_equal(absolute_value(-3), 3)
})
Code Coverage in R

- Run all tests of package `testcovr`

  ```r
  test("testcovr")
  ## Loading testcovr
  ## Testing testcovr
  ## ..
  ## DONE
  ```

- Analyze code coverage of package `testcovr`

  ```r
  package_coverage("testcovr")
  ## testcovr Test Coverage: 66.67%
  ## R\absolute_value.R: 66.67%
  ```

- Show locations of zero coverage

  ```r
  zero_coverage(package_coverage("testcovr"))
  ## filename functions first_line value
  ## 2 R\absolute_value.R absolute_value 3 0
  ```
Code Coverage in R

- Visual reports on code coverage via `shiny`

```r
s <- package_coverage("testcovr")
shine(s)
```

- Overall report

<table>
<thead>
<tr>
<th>Files</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Coverage</th>
<th>File</th>
<th>Lines</th>
<th>Relevant</th>
<th>Covered</th>
<th>Missed</th>
</tr>
</thead>
<tbody>
<tr>
<td>80.00</td>
<td>R\absolute_value.R</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>1 2</td>
</tr>
</tbody>
</table>

- Coverage line-by-line

```r
1   absolute_value <- function(x) {
2     if (x >= 0) {
3         return(x)                !
4     } else {
5         return(-x)                2x
6     }
7 }
```
Summary

Debugging

▶ **Locates bugs** or to understand code
▶ Tools: screen output, asserts, exceptions, interactive debuggers (for call stacks and breakpoints)

Software testing

▶ Software testing **measures quality**
▶ Functional vs. non-functional scope
▶ Static vs. dynamic testing
▶ White box vs. black box testing
▶ V model: acceptance, system, integration and unit testing
▶ Unit tests
  ▶ Performs automated checks of expectations
  ▶ Measures code coverage
  ▶ Use stubs/mocks to entangle dependencies
Further Readings: Debugging

- **Advanced R** (CRC Press, 2014, by Wickham)
  
  *Debugging, condition handling, and defensive programming*
  
  Section 9, pp. 149–171
  
  [http://adv-r.had.co.nz/Exceptions-Debugging.html](http://adv-r.had.co.nz/Exceptions-Debugging.html)

- **Debugging with R Studio**
  

- **Breakpoints in R Studio**
  

- **assertthat** package documentation at CRAN
  
  [https://cran.r-project.org/web/packages/assertthat/assertthat.pdf](https://cran.r-project.org/web/packages/assertthat/assertthat.pdf)
Further Readings: Unit Testing

- **Testing (by Wickham)**
  Book chapter: [http://r-pkgs.had.co.nz/tests.html](http://r-pkgs.had.co.nz/tests.html)
  Slides: [http://courses.had.co.nz/11-devtools/slides/7-testing.pdf](http://courses.had.co.nz/11-devtools/slides/7-testing.pdf)

- **testthat: Get Started with Testing**
  *R Journal, vol. 3 (1), 2011, by Wickham*

- **testthat package documentation at CRAN**: [https://cran.r-project.org/web/packages/testthat/testthat.pdf](https://cran.r-project.org/web/packages/testthat/testthat.pdf)

- **Mocks Aren’t Stubs (2007, by Fowler)**

- **Specialized materials for high-level programming languages, e.g.**
  *The Art of Unit Testing (Manning, by Osherove)*

- **covr package documentation at CRAN**
  [https://cran.r-project.org/web/packages/covr/covr.pdf](https://cran.r-project.org/web/packages/covr/covr.pdf)