Lecture 14: Testing Strategies for Code Debugging

36-350

13 October 2014

Last Time: Basic Debugging

Basic tricks for debugging:

- Notifications and alerts that you can add
- Localizing issues and changing input parameters
- Precomputed results

Today: Intermediate Debugging

Better success through design!

- Trusting our results through modular design
- Building tests: functional tests (top-level), unit tests (bottom-level)

Procedure versus Substance

Our two competing goals:

- Do we get *the right answer* (substance)?
- Do we get an answer *in the right way* (procedure)?

An important distinction, as these these go back and forth with each other:

- We trust a procedure because it gives the right answer.
- We trust the answer because it came from a good procedure.

Since programming means *making* a procedure, we check the substance primarily.

Testing for particular cases

Test cases with known answers

```
add <- function (part1, part2) { part1 + part2 }
a <- runif(1)
add(2,3) == 5
```

```
## [1] TRUE
```

```
add(a,0) == a
```

```
## [1] TRUE
```

```
add(a,-a) == 0
```

```
## [1] TRUE
```

Testing for particular cases

Real numbers and floating-point precision

```
cor(c(1,-1,1,1),c(-1,1,-1,1))
```

```
## [1] -0.5774
```

```
-1/sqrt(3)
```

```
## [1] -0.5774
```

```
cor(c(1,-1,1,1),c(-1,1,-1,1)) == -1/sqrt(3)
```

```
## [1] FALSE
```

Testing by cross-checking

Compare alternate routes to the same answer:

```
test.unif <- runif(n=3,min=-10,max=10)
add(test.unif[1],test.unif[2]) ==
  add(test.unif[2],test.unif[1])
```

```
## [1] TRUE
```

```
add(add(test.unif[1],test.unif[2]),test.unif[3]) ==
  add(test.unif[1],add(test.unif[2],test.unif[3]))
```

```
## [1] TRUE
```

```
add(test.unif[3]*test.unif[1],test.unif[3]*test.unif[2]) ==
  test.unif[3]*add(test.unif[1],test.unif[2])
```

```
## [1] FALSE
```

wwwwwww ## Testing by cross-checking

Test function: numerical derivative

```
x <- runif(10,-10,10)
f <- function(x) {x^2*exp(-x^2)}
g <- function(x) {2*x*exp(-x^2) -2* x^3*exp(-x^2)}
isTRUE(all.equal(derivative(f,x), g(x)))
```

Testing by cross-checking

If this seems too unstatistical…

```
xx <- runif(10)
aa <- runif(1)
cor(xx,xx) == 1
```

```
## [1] TRUE
```

```
cor(xx,-xx) == -1
```

```
## [1] FALSE
```

```
cor(xx,aa*xx) == 1
```

```
## [1] TRUE
```

Testing by cross-checking

```
pp <- runif(10); mean=0; sd=xx
all(pnorm(0,mean=mean,sd=sd) == 0.5)
```

```
## [1] TRUE
```

```
pnorm(xx,mean,sd) == pnorm((xx-mean)/sd,0,1)
```

```
##  [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

Testing by cross-checking

```
all(pnorm(xx,0,1) == 1-pnorm(-xx,0,1))
```

```
## [1] TRUE
```

```
pnorm(qnorm(pp)) == pp
```

```
##  [1] FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```
qnorm(pnorm(xx)) == xx
```

```
##  [1]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
```

With finite precision we don't really want to insist that these be exact!

Software Testings vs. Hypothesis Testing

Statistical hypothesis testing: risk of false alarm (size) vs. probability of detection (power) – this balances type I vs. type II errors

In software testing: no false alarms allowed (false alarm rate = 0). This must reduce our power to detect errors; code can pass all our tests and still be wrong.

But! we can direct the power to detect certain errors, *including* where the error lies, if we test small pieces.

Combining Testing and Coding

The idea behind unit testing:

- A variety of tests gives us more power to detect errors, more confidence when tests are passed.
- By breaking code into self-enclosed functions, we can better identify problems.
- *Therefore*: for each function, we build a battery of tests that are easy to step through and identify problems.
- This makes it easier to add new tests to a function as well.
- By bundling these tests into their own function, we keep program flow clean and remind ourselves later why this mattered!

The Great Testing Cycle

After making changes to a function, re-run its tests, and those of functions that depend on it.

- If anything's (still) broken, fix it; if not, continue.
- When you meet a new error, write a new test.
- When you add a new capacity, write a new test.

A Ratchet Approach: "Regression Testing"

When we have a version of the code which we are confident gets some cases right, keep it around (under a separate name).

- Now compare new versions to the old, on those cases
- Keep debugging until the new version is at least as good as the old

Test-Driven Development

General strategy for development.

1. Have an idea about what the program should do.
    - Idea is vague and unhelpful
    - Make it clear and useful by writing tests for success
    - Tests come *first*, then the program
2. Modify code until it passes all the tests

3. When you find a new error, write a new test
4. When you add a new capacity, write a new test
5. When you change your mind about the goal, change the tests
6. By the end, the tests specify what the program should do, and the program does it

Awkward Cases

Boundary cases, "at the edge" of something, or non-standard inputs. Such as:

```
add(5,NA)     # NA, presumably
```

```
## [1] NA
```

```
try(add("a","b")) # NA, or error message?
divide <- function (top, bottom) top/bottom
divide(10,0) # Inf, presumably
```

```
## [1] Inf
```

```
divide(0,0)   # NA?
```

```
## [1] NaN
```

Awkward Cases

Pinning down awkward cases helps specify function

```
var(1)        # NA? error?
```

```
## [1] NA
```

```
cor(c(1,-1,1,-1),c(-1,1,NA,1))   # NA? -1? -1 with a warning?
```

```
## [1] NA
```

```
try(cor(c(1,-1,1,-1),c(-1,1,"z",1)))  # NA? -1? -1 with a warning?
try(cor(c(1,-1),c(-1,1,-1,1)))        # NA? 0? -1?
```

Pitfalls

- Writing tests takes time
- Running tests takes time
- Tests have to be debugged themselves
- Tests can provide a false sense of security
- There are costs to knowing about problems (people get upset, responsibility to fix things, etc.)

Summary

- Trusting software means testing it for correctness, both of substance and of procedure
- Software testing is an extreme form of hypothesis testing: no false positives allowed, so any power to detect errors has to be very focused
- $\therefore$ Write and use lots of tests; add to them as we find new errors
- Cycle between writing code and testing it