COSC345
Intro to Unit Testing
Tuesday 2015.04.28
Every test includes

- a **thing** to be tested
- a **state** to put it in
- a **stimulus** to poke it with
- a **response** that counts as good
An example

- A popular algorithm for converting Julian day number to \((y,m,d)\), Fliegel & van Flandern, CACM vol 11 issue 10.

- They also offered an algorithm to go the other way.

- Both algorithms were written in Fortran.
subroutine DATE(JD,I,J,K)

L = JD + 68569
N = 4*L/146097
L = L - (146097*N + 3)/4
I = 4000*(L + 1)/1461001
L = L - 1461*I/4 + 31
J = 80*L/2447
K = L - 2447*J/80
L = J/11
J = J + 2 - 12*L
I = 100*(N - 49) + I + L
end
Language warping

- Variables I, J, K were used instead of Year, Month, Day so no declaration needed!
- All arguments are variables
- This made transcription particularly hard.
- It didn’t actually have to be that way.
What is to be tested?

• This algorithm.

• No, wait, the algorithm was stated in Fortran. I want it in another language, so

• A *translation* into (C, Smalltalk) of a *transcription* of this algorithm.
What’s the state?

- This algorithm is stateless. The state is the trivial state.
What’s the stimulus?

• 1000 years = 365,242+ days, so we can afford to test every legal input in some range.

• What range depends on what we want to handle.

• The Gregorian calendar was adopted in 1582. Who knows what we’ll use next century? (Revised Julian is better...)
Stimulus continued

• ISO 8601 requires the proleptic Gregorian calendar. That is, using the wrong calendar for dates in the past is the official international standard!

• SQL’92 requires support for years -9999 to +9999.

• Choose the SQL’92 range!
Response

• Check that JD 2440588 \(\Rightarrow\) 1970,1,1

• Check that for all dates in range,
  \[\text{date}(\text{JD}+1) = \text{the day after date}(\text{JD})\]
Result

- Anchor check: passed
- Increment check: FAILED

- What went wrong?
Check 1

• Transcribe the algorithm again, very carefully. Check it against other copies by other people. No error found.

• Probe to find where the algorithm goes wrong.

• Around the year -4714. Light dawns.
Julian day count

- The Julian day count starts at 1 January 4713 BC
- which is 24 November -4714 on the proleptic Gregorian calendar
- Something goes wrong with negative Julian day numbers.
Ad fontes!

- The source is explicit that
  - “truncation ... integer arithmetic” is used
    [got that right]
  - The algorithm is valid for Julian dates greater than zero
    [and the SQL’92 range violates that]
Lessons

• Check that you are not using a function outside its declared range.

• Use assertions to do this:

  assert(JD > 0)

• Use the right division: using floor(x/y) instead of truncate(x/y) just works
Lessons 2

• Just because it’s published and used elsewhere doesn’t mean that you shouldn’t test it.

• Very simple tests can find problems.

• It’s nice when you can test something exhaustively. When you can’t, random testing is better than nothing.
Lessons 3

• I have four copies of this code, one each in DateAndTime, Date, GregorianDate, and ISODate.

• Testing one of them is not enough. They all needed testing. And they all needed fixing.

• I wish I had only had one copy...
Unit Test Frameworks
What’s a “Unit”

- A **unit** or **component** (see Testing-Glossary.pdf) is the smallest chunk of software that you *can* test.

- Typically it is a function, but it might be a class or a script.
Unit Test Frameworks

• A test jig for software.
• We want to *script* tests to *automatically* set up the “unit under test”, apply stimuli, and record the results.
• Having written our scripts, we want running them to be just-push-a-button.
Importance of coupling

• Coupling refers to how tightly a chunk of code is bound to its environment

• Read the Wikipedia page on coupling.

• Coupling through shared mutable data is pretty bad.

• If A is coupled to B, it’s very hard to test A without B
Mocking isn’t this

ARE YOU COMING TO THE CODE MOCKING?

THE WHAT?

CODE MOCKING IS AN ENGINEERING TRADITION. IT HAPPENS WHENEVER A SOFTWARE PROJECT IS HANDLED TO A NEW ENGINEER.

THE NEW ENGINEER IS REQUIRED TO MOCK THE PREVIOUS ENGINEER’S WORK IN A PUBLIC WAY.

WE SPECTATORS GET TO VOTE ON WHETHER THE OLD CODE IS KILLED OR SPARED.

HA HA! HIS CODE IS HILARIOUSLY INEFFICIENT!

OUCH. CHEST PAIN.

KILL IT! KILL IT! KILL IT!

GAAA!!! THE CODE IS OFFENDING MY ENGINEERING SENSIBILITIES! IT’S KILLING ME!

I FORGOT TO MENTION THAT SOMETIMES THE CODE WINS.
Mocking is

• when A is coupled to B
• and you want to test A without B
• create a mock with just enough interface and body to make A think it’s the real thing
• User interfaces are often mocked so you can test without a user
The SUnit model

- A TestCase class
- has `setUp` to set up a state and `tearDown` to release it
- has a bunch of `testXXX` methods
- TestRunner is a GUI; once you’ve found your TestCase you can rerun it with a click
Basic methods

• self assert: should-be-true
• self deny: should-be-false
• self should: [statements] raise: Exception
• self shouldn’t: [statements] raise: Exception