CS1632, LECTURE 5: MANUAL VS AUTOMATED TESTING, STATIC ANALYSIS, AND MUTATION TESTING

Bill Laboon

Manual Testing

- What we have been doing so far
- We write test plans
- A human executes them on the software

Automated Tests

- Mostly what we’ll be doing from here on out
- We write tests which the computer executes for us
- Automated tests can test at a very low-level (e.g. methods) or at a very high level (full subsystem or system testing)

Benefits of Manual Testing

1. It’s simple!
2. It’s cheap (at first)
3. It’s easy to set up
4. No additional software to learn or write
5. Flexible
6. Can focus on things users care about
7. Humans catch issues that programs may not notice

Drawbacks of Manual Testing

1. It is BORING
2. It can be unrepeatable
3. Some tasks are difficult to test manually, e.g.:
    1. Timing
    2. Low-level interfaces
4. Human error is a possibility
5. It’s time and resource-intensive

Benefits of Automated Testing

1. No chance for human error (during execution)
2. Fast test execution
3. Easy to execute once set up
4. Repeatable
5. Less resource-intensive during testing
6. Ideal for testing some things that manual testing is bad for

Drawbacks of Automated Testing

1. Requires extra time up-front
2. May not catch user-facing bugs
3. Requires learning tools and frameworks (but that's one of the things this class can help with)
4. Requires more skilled staff
5. Big issue: It only tests what it is looking for

Solution: A Mixture

- Most teams will use both manual and automated tests
- Usually, the number of automated tests will far outnumber the number of manual tests

Static vs Dynamic Analysis

- Static analysis – Code is not executed by the test
- Dynamic analysis – Code is executed by the test.
  - *Almost everything that we have done so far!*

Kinds of Static Analysis

- Code coverage
- Code review / walk-through
- Code metrics
- Formal verification
- Compilers (technically!)
- Bug finders
- Linters

Code Coverage

- How much of the codebase is covered by a particular test suite.
- This can have different meanings!

Code Coverage

Consider the following code and tests and (pseudocode) unit tests

```
def quack x
  if x > 0
    "Quack!"
  else
    “Negative Quack!”
  end
end
def quock
  “Quock!"
end

assert_equal “Quack!”, quack(1)
assert_equal “Negative Quack!”, quack(-4)
```

Method Coverage

- What percentage of all methods have been called?
- In previous example, 50%

Code Coverage

- Consider the following code and tests and (pseudocode) unit tests

```
def noogie x
   if x < 10
      1
   else
      if (Math.sqrt(x) % 2 == 0)
         x / 0
      else
         3
      end
   end
end

assert_equal 1, noogie(5)
assert_equal 3, noogie(81)
assert_equal 1, noogie(9)
```

Statement Coverage

- That’s 100% method coverage, but we are missing some statements!
- Statement coverage = percentage of statements that have been tested
- This is usually what’s referred to as “code coverage” (although technically it’s a *kind* of code coverage)

Other Kinds of Code Coverage

- Branch coverage
- Condition coverage
- Decision coverage
- Parameter value coverage
- JJ-path Coverage
- Path Coverage
- Entry/Exit Coverage
- State coverage

What’s the benefit?

Code coverage metrics lets you easily see where more tests would be useful and where tests are missing.

It does NOT tell you whether your tests are valid, or whether that line will always work, only that it will be executed by a test!

Consider the following…

```
def chirp x, y
    z = Math.sqrt(x)
    z / y
end

# 100% statement coverage

assert_equal 1, chirp(1, 1)
```

Note

- Low code coverage is bad, but high code coverage does not always mean good.
- Even 100% of code coverage cannot catch 100% of bugs!

Things Code Coverage Can’t Catch

- Different variable values
- Performance issues
- Race conditions
- Combinatoric issues
- Anything subjective (e.g. usability, readability)
- More!

Code Metrics

- Allow you to check :
  - *Cyclomatic complexity!*
  - *Class fan-out!*
  - *Number of lines per class!*
  - *Number of interfaces!*
  - *Depth of inheritance tree!*
  - *NORM (Number of overridden methods)*
  - *Weighted methods per class!*
  - *Afferent and Efferent Coupling!*

Code Metrics

Honestly, I have not found most of these very useful (except cyclomatic complexity).

Some people/companies swear by them, though.

You can set “triggers” for these - e.g., don’t let anybody check in code if cyclomatic complexity for any method is greater than 50.

Formal Verification

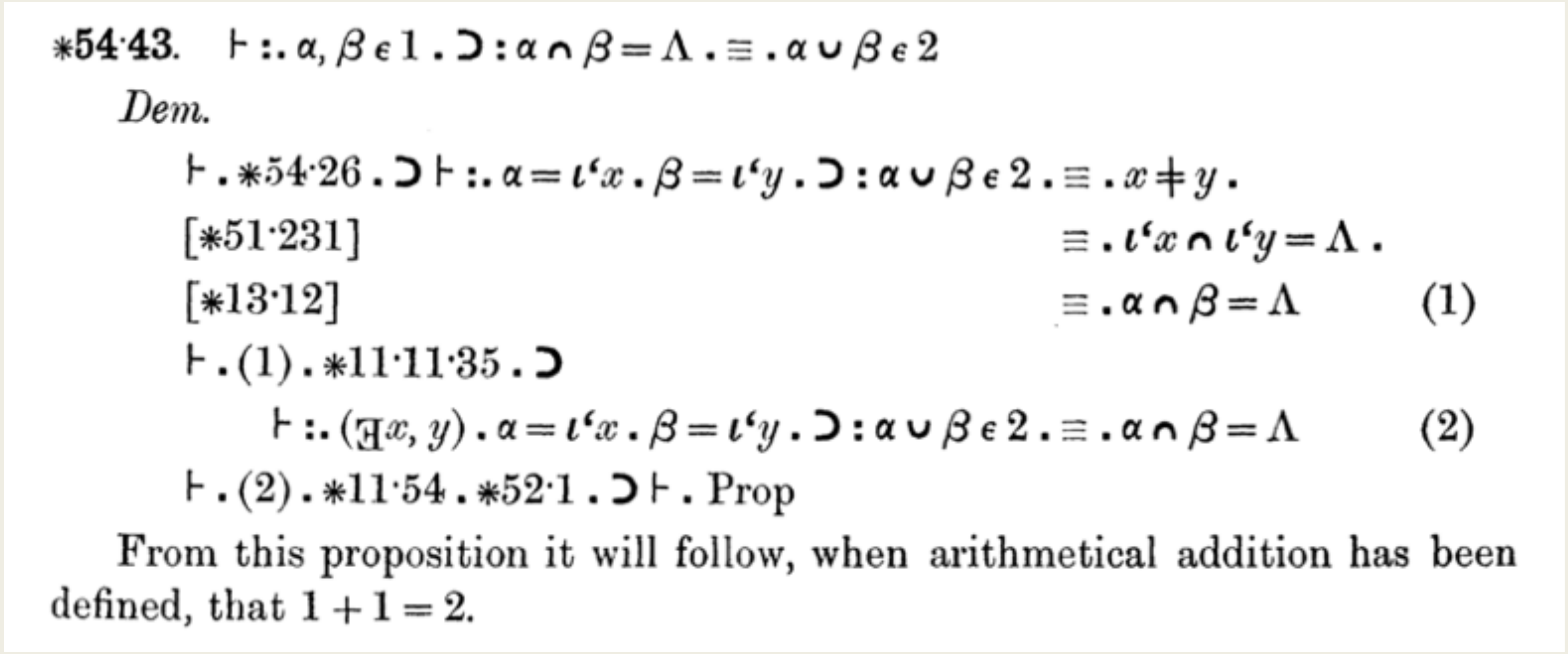
- Mathematically prove the behavior of a program from first principles
- Allows for (in principle) defect-free code
- Often done by reducing to a non-Turing-complete language to avoid the Halting Problem

Wait, did you say defect-free code?!?!

- Sure did. Go back and see.

So why aren’t we using formal verification all the time?

Page 379 of the Principia Mathematica



$*54\cdot43.\quad \vdash :. \alpha, \beta \in 1 . \supset : \alpha \cap \beta = \Lambda . \equiv . \alpha \cup \beta \in 2$

Dem.

$\vdash . *54\cdot26 . \supset \vdash :. \alpha = \iota\text{‘}x . \beta = \iota\text{‘}y . \supset : \alpha \cup \beta \in 2 . \equiv . x \neq y .$

$[*51\cdot231] \qquad \equiv . \iota\text{‘}x \cap \iota\text{‘}y = \Lambda .$

$[*13\cdot12] \qquad \equiv . \alpha \cap \beta = \Lambda \qquad (1)$

$\vdash . (1) . *11\cdot11\cdot35 . \supset$

$\vdash :. (\exists x, y) . \alpha = \iota\text{‘}x . \beta = \iota\text{‘}y . \supset : \alpha \cup \beta \in 2 . \equiv . \alpha \cap \beta = \Lambda \qquad (2)$

$\vdash . (2) . *11\cdot54 . *52\cdot1 . \supset \vdash .$ Prop

From this proposition it will follow, when arithmetical addition has been defined, that $1 + 1 = 2$.

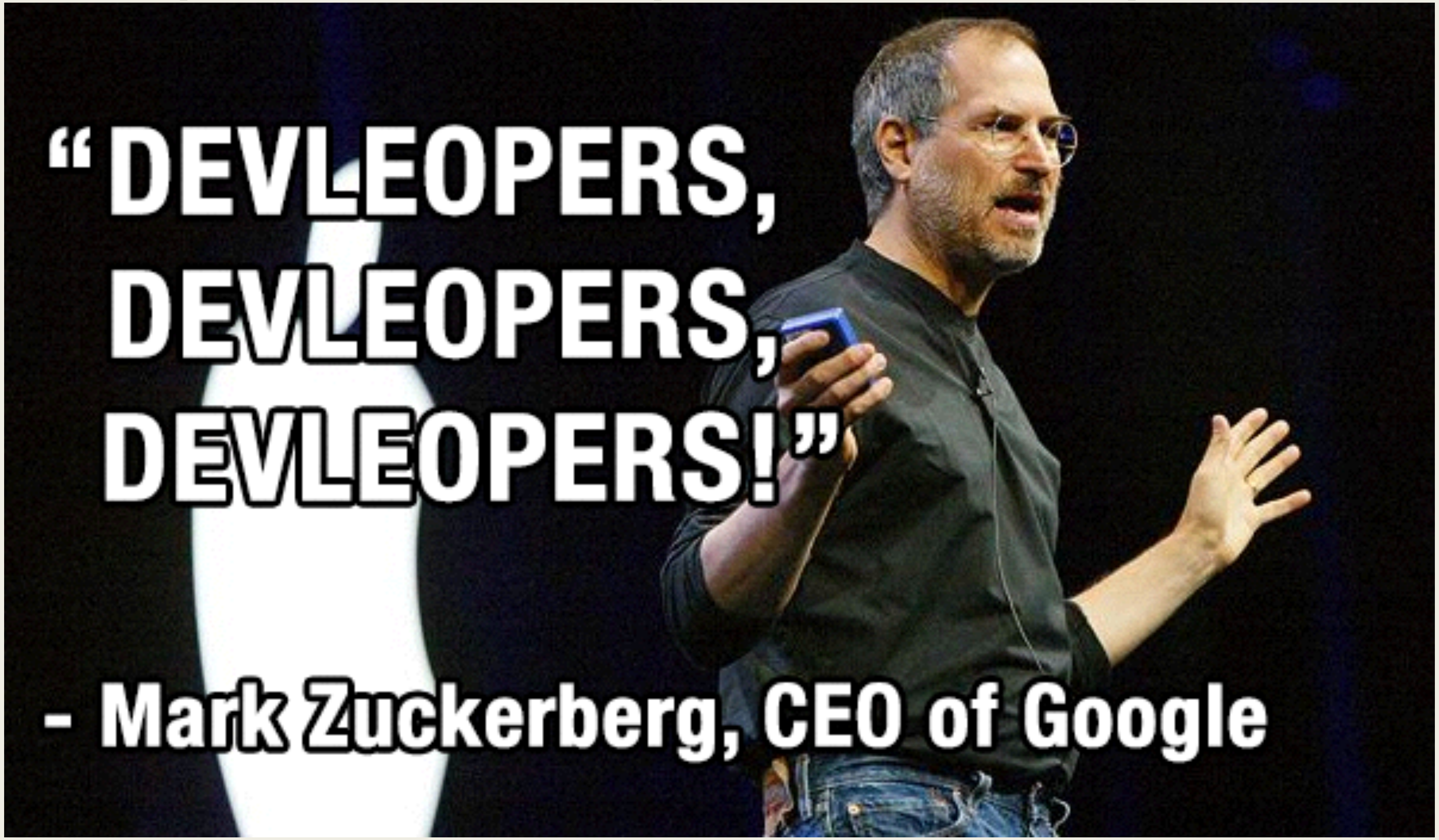
Formal Verification Is Slow and Inflexible

- Used by:
  - *Some embedded programs*
  - *Some cryptography libraries*
  - *At least one kernel, seL4*
  - *Places where accuracy is absolutely paramount*

And assumes you formally verified correctly!

- You may have a bug in your formal verification software
- You may not have specified it correctly - (remember, verification is not the same as validation)
- Does not take any subjective factors into account (e.g. usability of program, scalability, etc.)

Compilers! Compilers! Compilers!



A compiler can tell you…

- If your syntax is right
- If you have uncaught exceptions (in Java)
- If you have dead code (in Java)
- The ruby and javac compilers aren’t the best ones out there, but it can give you lots of information
- See the rustc and elm compilers for some really good static analysis

Bug Finders

- Static tools that focus on finding defects without executing code
- Full of false positives!
- Can be pointers to unclear or incorrect code

Example

```
def doStuff x
   return 5
   if x == 0
      x = 1
   else
      x = 3
   end
   x = 6
end
```

Useless conditional and other statement

- Has dead code - will not continue after “return”

Example

```
def yay_math
   x = 0.1
   y = 0.2
   z = x + y
   if z == 0.3
      puts “math works!”
   else
      puts “math is arbitrary!”
   end
end
```

`yay_math outputs “math is arbitrary!”`

Direct Equality Comparison of Floating-Point Values

- Floating-point values are approximations
- Always check to see if values are within epsilon of each other, e.g.
  - `if (z.abs - 3.0) < 0.01)`
- Or use BigDecimal (works in Ruby or Java)

BigDecimal Usage in Ruby

```
irb(main):020:0> require 'bigdecimal'
=> true
irb(main):022:0> x = BigDecimal::new("0.1")
=> 0.1e0
irb(main):023:0> y = BigDecimal::new("0.2")
=> 0.2e0
irb(main):024:0> z = x + y
=> 0.3e0
```

Optimization Example

```
def calculate
   Math.sqrt(90)
end
```

Return value will always be the same

- Just put the calculated value instead of calculating each time

```
def calculate
  9.486832980505138
end
```

All of these issues can be found without running code

- Some are performance defects
- Some are functional defects (causing incorrect behavior)
- Some are just confusing code – which can cause even more problems!

Linters

- Poorly written code can cause problems
- Multiple people writing code in different styles can cause issues

Imagine reading this (VALID!) code…

```
def DOSOMETHING y; x = y; bARGLE = 9; if x <
100; puts x; else; puts "MEOOOOOW"; end; end
```

Linters allow an entire team to use consistent spacing, tabs, variable naming, etc.

- Very, very common!
- I can’t remember ever working on professional code that did not have a style guide
- It’s been (well) over a decade since I worked on code which did not have an automated tool to check it

Mutation Testing

- A way of testing our tests!
- Modifies the code of the method…
- … then checks to see if our test suite still passes.
- If it does, that shows potential holes in our testing!

Mutation Testing Example

```
def yggdrasil x
  return 0 if x.nil?
  @tree += x
  if x >= 0
    1
  else
    -1
  end
end

# Tests (pseudocode)
assert_equal 1, yggdrasil(1)
assert_equal -1, yggdrasil(-1)
assert_equal 1, yggdrasil(0)
assert_equal 1, yggdrasil(9999999999999999999)
assert_equal 0, yggdrasil(nil)
```

100% code coverage!

```
def yggdrasil x
  return 0 if x.nil?
  @tree += x
  if x >= 0
    1
  else
    -1
  end
end

# Tests (pseudocode)
assert_equal 1, yggdrasil(1)
assert_equal -1, yggdrasil(-1)
assert_equal 1, yggdrasil(0)
assert_equal 1, yggdrasil(9999999999999999999)
assert_equal 0, yggdrasil(nil)
```

Mutation

```
def yggdrasil x
  return 0 if x.nil?
  @tree += x
  if x < 0
    1
  else
    -1
  end
end

# Tests (pseudocode) - FAILURES!  GOOD!
assert_equal 1, yggdrasil(1)
assert_equal -1, yggdrasil(-1)
assert_equal 1, yggdrasil(0)
assert_equal 1, yggdrasil(9999999999999999999)
assert_equal 0, yggdrasil(nil)
```

Mutation

```
def yggdrasil x
  return 9 if x.nil?
  @tree += x
  if x >= 0
    1
  else
    -1
  end
end

# Tests (pseudocode) - FAILURE!  GOOD!
assert_equal 1, yggdrasil(1)
assert_equal -1, yggdrasil(-1)
assert_equal 1, yggdrasil(0)
assert_equal 1, yggdrasil(9999999999999999999)
assert_equal 0, yggdrasil(nil)
```

Mutation Testing Example

```
def yggdrasil x
  return 0 if x.nil?
  # @tree += x
  if x >= 0
    1
  else
    -1
  end
end

# Tests (pseudocode) - NO FAILURES!
# Either hole in our testing OR superfluous code
assert_equal 1, yggdrasil(1)
assert_equal -1, yggdrasil(-1)
assert_equal 1, yggdrasil(0)
assert_equal 1, yggdrasil(9999999999999999999)
assert_equal 0, yggdrasil(nil)
```