lecture 13: automated testing

csc302h
winter 2014

announcements

- midterm in tutorial hour today

location	groups
SS1074	• The Brogrammers • Missing Brackets • Solutions
GB248 (lecture room)	• doge++ • Seven - 2 • Double Double
RW229	• THE Group • Fantasix • DOGE

Automated Testing

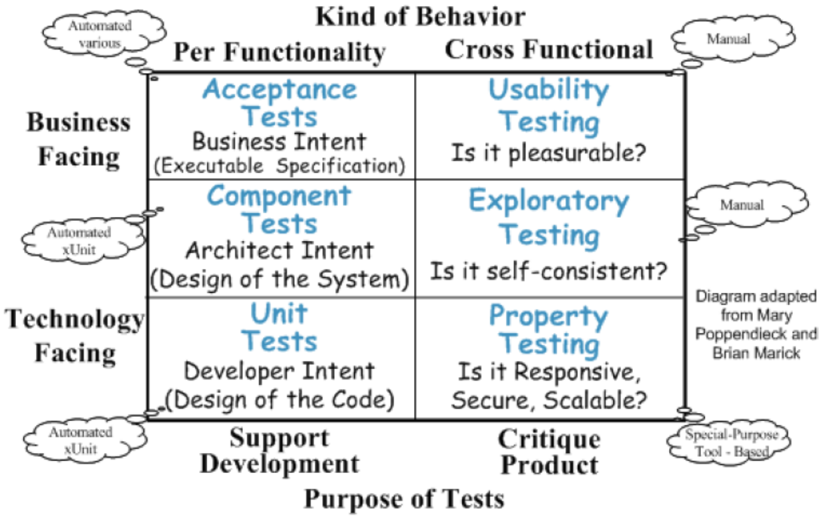
Automated testing

JUnit and family

Testing GUI-based software

Testing Object-Oriented Systems

When to stop testing



Kind of Behavior

	Per Functionality	Cross Functional
Business Facing	Acceptance Tests Business Intent (Executable Specification)	Usability Testing Is it pleasurable?
	Component Tests Architect Intent (Design of the System)	Exploratory Testing Is it self-consistent?
Technology Facing	Unit Tests Developer Intent (Design of the Code)	Property Testing Is it Responsive, Secure, Scalable?
	Support Development	Critique Product

Purpose of Tests

Diagram adapted from Mary Poppendieck and Brian Marick

Automated Testing

Source: Adapted from Liskov & Guttag, 2000, pp239-242

Where possible, automate your testing:

- tests can be repeated whenever the code is modified (“regression testing”)
- takes the tedium out of extensive testing
- makes more extensive testing possible

Will need:

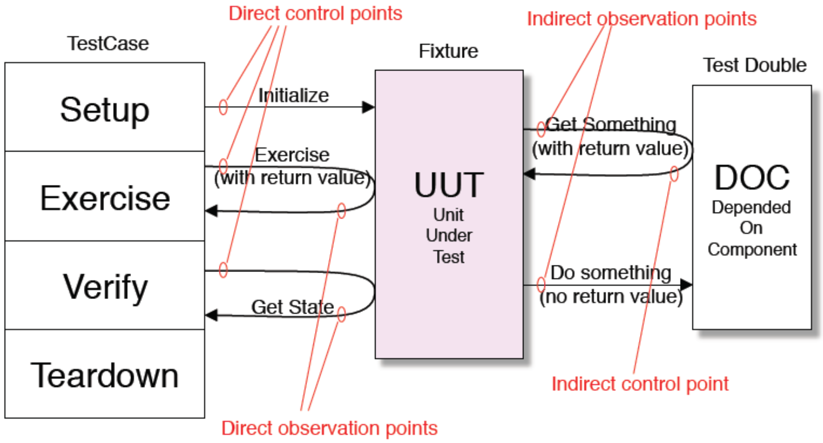
- test drivers - automate the process of running a test set
  - sets up the environment
  - makes a series of calls to the Unit-Under-Test (UUT)
  - saves results and checks they were right
  - generates a summary for the developers

May need:

- test stubs - simulate part of the program called by the unit-under-test
  - checks whether the UUT set up the environment correctly
  - checks whether the UUT passed sensible input parameters to the stub
  - passes back some return values to the UUT (according to the test case)
  - (stubs could be interactive - ask the user to supply return values)

Automated Testing Strategy

Source: Adapted from Meszaros 2007, p66



Direct control points

Indirect observation points

TestCase

Fixture

Test Double

Setup

Exercise

Verify

Teardown

Initialize

Exercise (with return value)

Get State

UUT Unit Under Test

Get Something (with return value)

Do something (no return value)

DOC Depended On Component

Indirect control point

Direct observation points

Principles of Automated Testing

Source: Adapted from Meszaros 2007, p39-48

Write the Test Cases First

Design for Testability

Use the Front Door First

- test via public interface
- avoid creating back door manipulation

Communicate Intent

- Tests as Documentation!
- Make it clear what each test does

Don’t Modify the UUT

- avoid test doubles
- avoid test-specific subclasses
- (unless absolutely necessary)

Keep tests Independent

- Use fresh fixtures
- Avoid shared fixtures

Isolate the UUT

Minimize Test Overlap

Check One Condition Per Test

Test Concerns Separately

Minimize Untestable code

- e.g. GUI components
- e.g. multi-threaded code
- etc

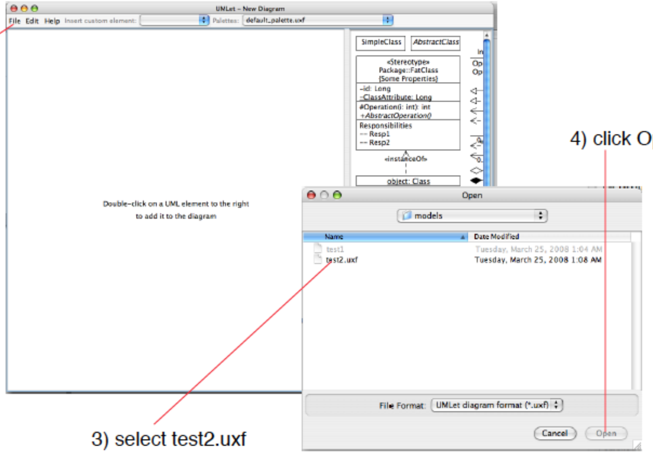
Keep test logic out of production code

- No test hooks!

Testing interactive software

1) Start the application (e.g. UMLet)

2) Click on File -> Open



Double-click on a UML element to the right to add it to the diagram

Name	Date Modified
test1	Tuesday, March 25, 2008 1:04 AM
test2.uxf	Tuesday, March 25, 2008 1:08 AM

File Format: UMLet diagram format (*.uxf)

4) click Open

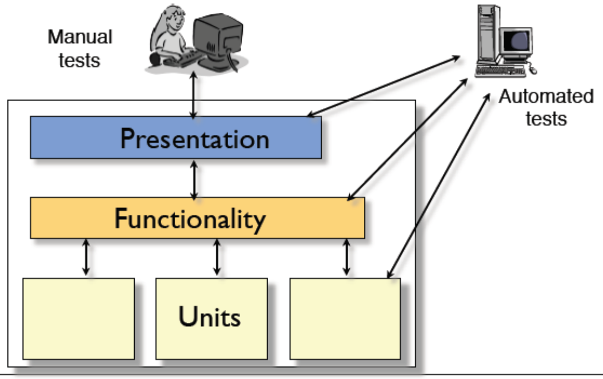
3) select test2.uxf

Automating the testing

Source: Adapted from Zeller 2006, p57

Challenges for automated testing:

- Synchronization - How do we know a window popped open that we can click in?
- Abstraction - How do we know it's the right window?
- Portability - What happens on a display with different resolution / size, etc



Testing the Presentation Layer

Source: Adapted from Zeller 2006, chapter 3

Script the mouse and keyboard events

- script can be recorded (e.g. "send_xevents @400,100")
- script is write-only and fragile

Script at the application function level

- E.g. Applescript: tell application "UMLet" to activate
- Robust against size and position changes
- Fragile against widget renamings, layout changes, etc.

Write an API for your application…

- Allow an automated test to create windows, interact with widgets, etc.

How to Test Object Oriented Code?

Encapsulation

- If the object hides it's internal state, how do we test it?
- Could add methods that expose internal state, only to be used in testing
- But: how do we know these extra methods are correct?

Inheritance

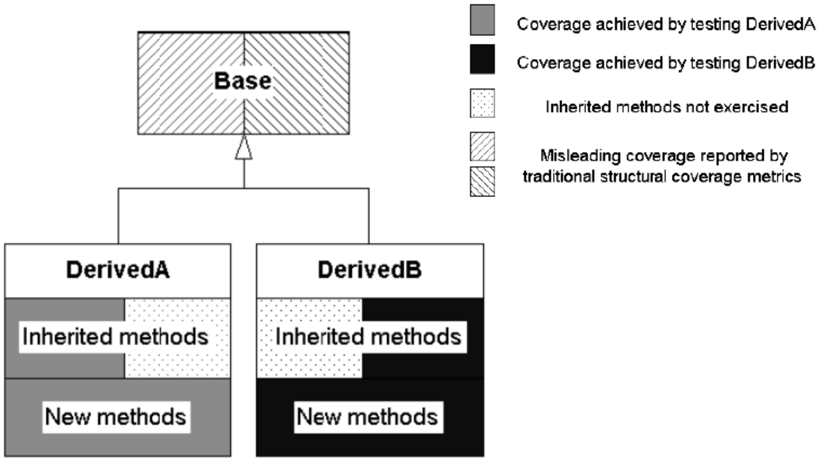
- When a subclass extends a well-tested class, what extra testing is needed?
  - e.g. Test just the overridden methods?
- But with dynamic binding, this is not sufficient
  - e.g. other methods can change behaviour because they call over-ridden methods

Polymorphism

- When class A calls class B, it might actually be interacting with any of B's subclasses…

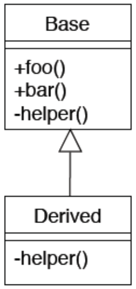
Inheritance Coverage

Source: Adapted from IPL 1999


Base
DerivedA
Inherited methods
New methods
DerivedB
Inherited methods
New methods
Coverage achieved by testing DerivedA
Coverage achieved by testing DerivedB
Inherited methods not exercised
Misleading coverage reported by traditional structural coverage metrics

Consider this program...

Source: Adapted from IPL 1999



Base

+foo()
+bar()
-helper()

Derived

-helper()

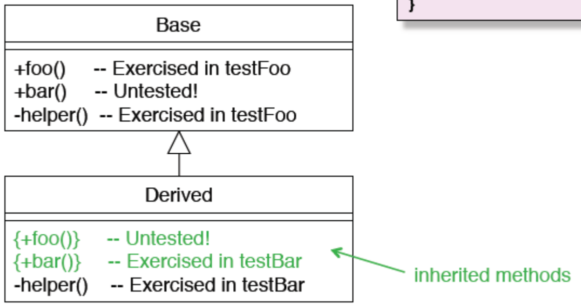
```
class Base {
  public void foo() {
    … helper(); …
  }
  public void bar() {
    … helper(); …
  }
  private helper() {…}
}

class Derived extends Base {
  private helper() {…}
}
```

Test Cases

Source: Adapted from IPL 1999

```
public void testFoo() {
  Base b = new Base();
  b.foo();
}
public void testBar() {
  Derived d = new Derived();
  d.bar();
}
```



Base

+foo() -- Exercised in testFoo
+bar() -- Untested!
-helper() -- Exercised in testFoo

Derived

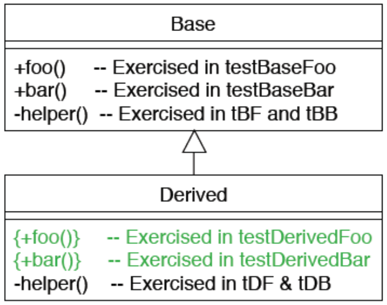
{+foo()} -- Untested!
{+bar()} -- Exercised in testBar
-helper() -- Exercised in testBar

inherited methods

Extend the test suite

Source: Adapted from IPL 1999

```
public void testBaseFoo() {
  Base b = new Base();
  b.foo();
}
public void testBaseBar() {
  Base b = new Base();
  b.bar();
}
public void testDerivedFoo() {
  Base d = new Derived();
  d.foo();
}
public void testDerivedBar() {
  Derived d = new Derived();
  d.bar();
}
```



Base

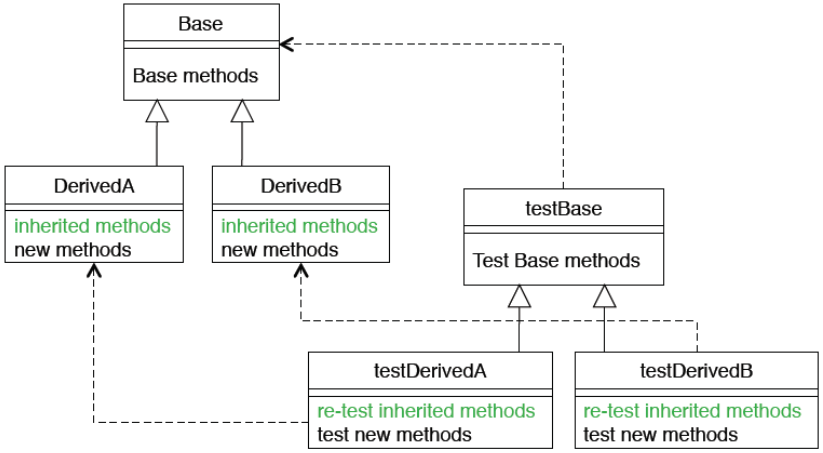
+foo() -- Exercised in testBaseFoo
+bar() -- Exercised in testBaseBar
-helper() -- Exercised in tBF and tBB

Derived

{+foo()} -- Exercised in testDerivedFoo
{+bar()} -- Exercised in testDerivedBar
-helper() -- Exercised in tDF & tDB

Subclassing the Test Cases

Source: Adapted from IPL 1999



Base

Base methods

DerivedA

inherited methods
new methods

DerivedB

inherited methods
new methods

testBase

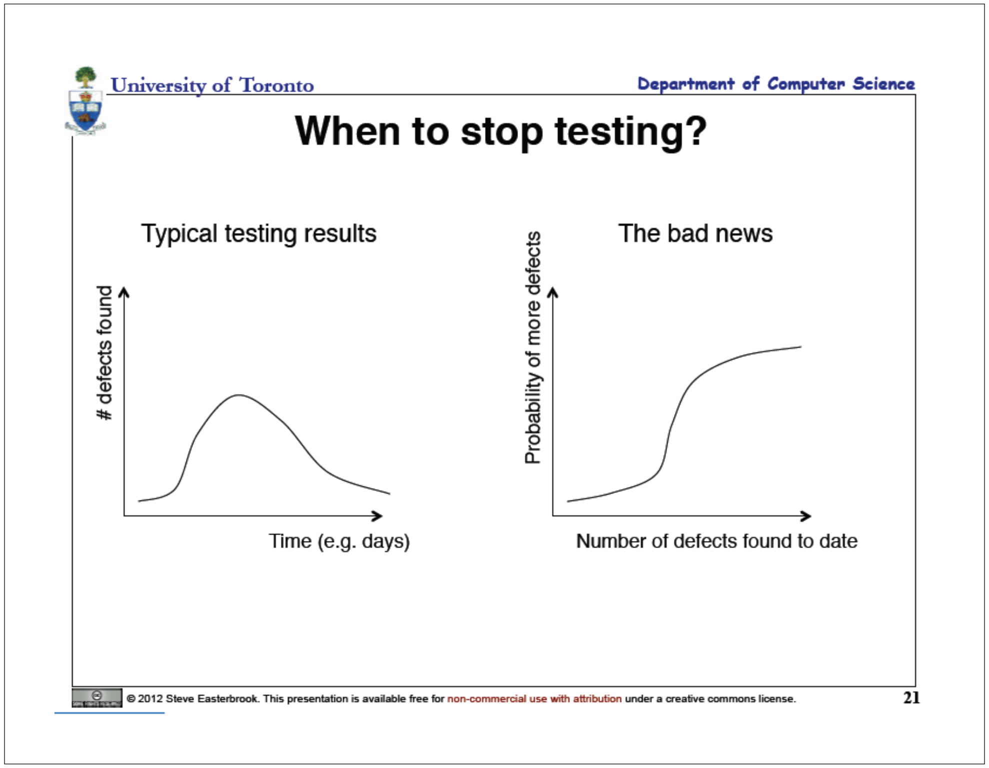
Test Base methods

testDerivedA

re-test inherited methods
test new methods

testDerivedB

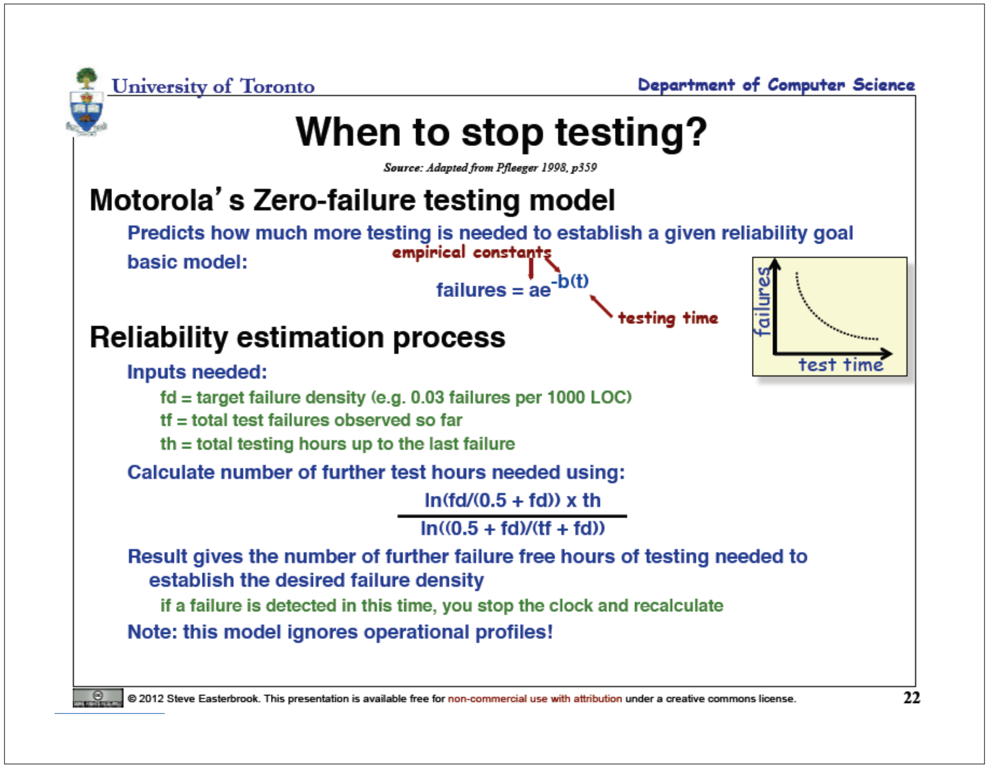
re-test inherited methods
test new methods



When to stop testing?

Typical testing results

The bad news



When to stop testing?

Source: Adapted from Pfleeger 1998, p359

Motorola's Zero-failure testing model

- Predicts how much more testing is needed to establish a given reliability goal
- basic model:

empirical constants

$$\text{failures} = ae^{-b(t)}$$

testing time

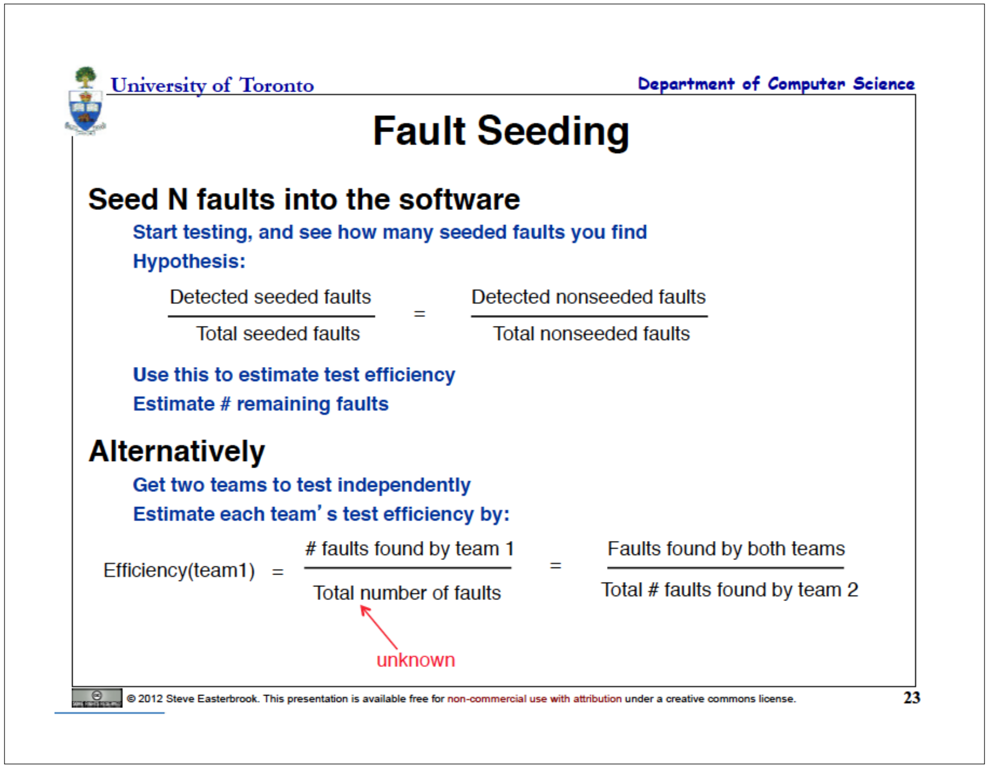
failures
test time

Reliability estimation process

- Inputs needed:
  - fd = target failure density (e.g. 0.03 failures per 1000 LOC)
  - tf = total test failures observed so far
  - th = total testing hours up to the last failure
- Calculate number of further test hours needed using:

$$\frac{\ln(fd/(0.5 + fd)) \times th}{\ln((0.5 + fd)/(tf + fd))}$$

- Result gives the number of further failure free hours of testing needed to establish the desired failure density
  - if a failure is detected in this time, you stop the clock and recalculate
- Note: this model ignores operational profiles!



Fault Seeding

Seed N faults into the software

- Start testing, and see how many seeded faults you find
- Hypothesis:

$$\frac{\text{Detected seeded faults}}{\text{Total seeded faults}} = \frac{\text{Detected nonseeded faults}}{\text{Total nonseeded faults}}$$

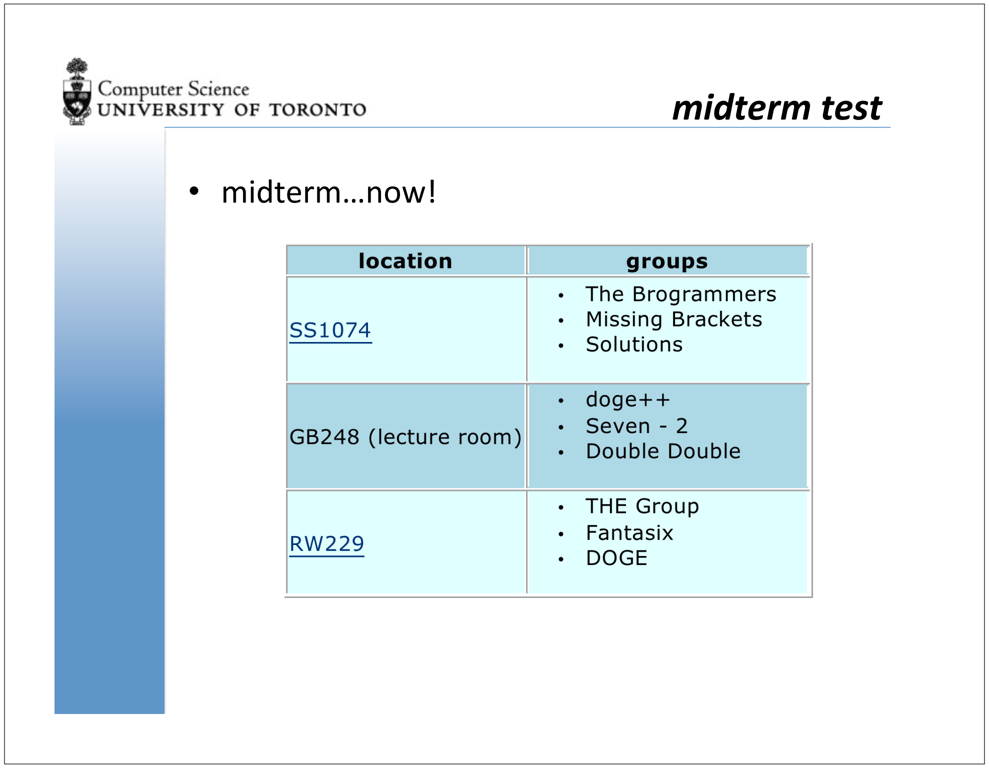
- Use this to estimate test efficiency
- Estimate # remaining faults

Alternatively

- Get two teams to test independently
- Estimate each team's test efficiency by:

$$\text{Efficiency(team1)} = \frac{\text{\# faults found by team 1}}{\text{Total number of faults}} = \frac{\text{Faults found by both teams}}{\text{Total \# faults found by team 2}}$$

unknown



Computer Science
UNIVERSITY OF TORONTO

midterm test

- midterm…now!

location	groups
SS1074	• The Brogrammers • Missing Brackets • Solutions
GB248 (lecture room)	• doge++ • Seven - 2 • Double Double
RW229	• THE Group • Fantasix • DOGE