Lecture 20: Black Box & Exploratory Testing

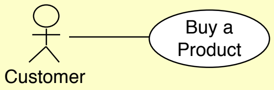
Use Cases as Test Cases

Quicktests

Exploratory Testing

When to stop testing

Generating Tests from Use Cases



Buy a Product

Precondition: Customer has successfully logged in

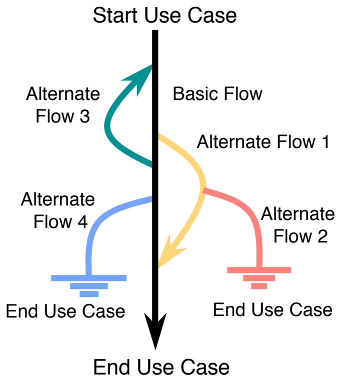
Main Success Scenario:
Customer browses catalog and selects items to buy
Customer goes to check out
Customer fills in shipping information (address, next-day or 3-day delivery)
System presents full pricing information
Customer fills in credit card information
System authorizes purchase
System confirms sale immediately
System sends confirming email to customer

Postcondition: Payment was received in full, customer has received confirmation

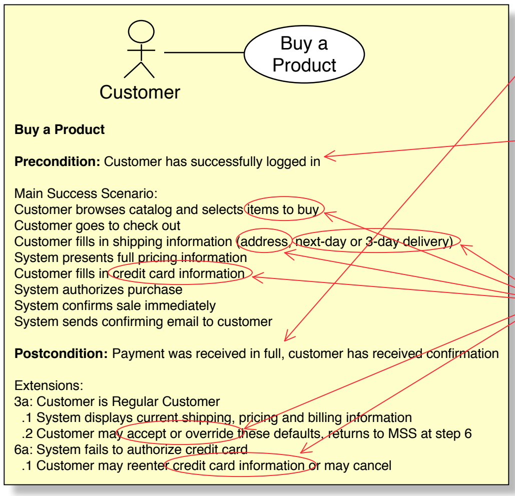
Extensions:
3a: Customer is Regular Customer
.1 System displays current shipping, pricing and billing information
.2 Customer may accept or override these defaults, returns to MSS at step 6
6a: System fails to authorize credit card
.1 Customer may reenter credit card information or may cancel

1 Test the Basic Flow

2 Test the Alternate Flows



Generating Tests from Use Cases



Buy a Product

Precondition: Customer has successfully logged in

Main Success Scenario:
Customer browses catalog and selects items to buy
Customer goes to check out
Customer fills in shipping information (address, next-day or 3-day delivery)
System presents full pricing information
Customer fills in credit card information
System authorizes purchase
System confirms sale immediately
System sends confirming email to customer

Postcondition: Payment was received in full, customer has received confirmation

Extensions:
3a: Customer is Regular Customer
.1 System displays current shipping, pricing and billing information
.2 Customer may accept or override these defaults, returns to MSS at step 6
6a: System fails to authorize credit card
.1 Customer may reenter credit card information or may cancel

3 Test the Postconditions

Are they met on all paths through the use case?

Are all postconditions met?

4 Break the Preconditions

What happens if this is not met?

In what ways might it not be met?

5 Identify options for each variable

select combinations of options for each test case

Classes of input variables

values that trigger alternative flows

e.g. invalid credit card

e.g. regular customer

trigger different error messages

e.g. text too long for field

e.g. email address with no “@”

inputs that cause changes in the appearance of the UI

e.g. a prompt for additional information

inputs that causes different options in dropdown menus

e.g. US/Canada triggers menu of states/provinces

cases in a business rule

e.g. No next day delivery after 6pm

border conditions

if password must be min 6 characters,

test password of 5,6,7 characters

Check the default values

e.g. when cardholder’s name is filled automatically

Override the default values

e.g. when the user enters different name

Enter data in different formats

e.g. phone numbers:

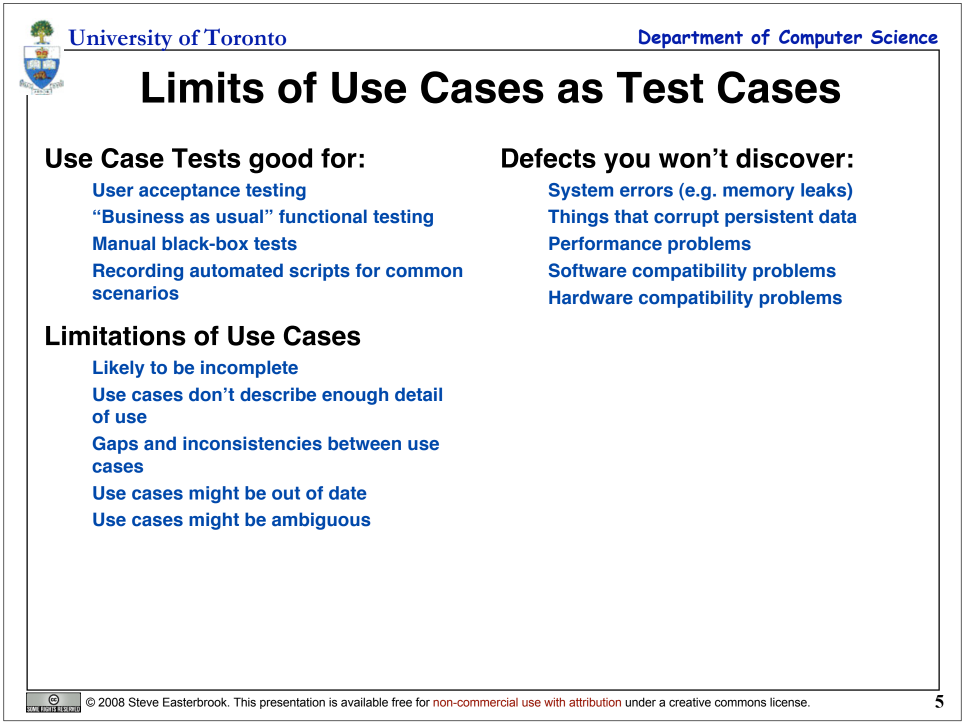
(416) 555 1234

416-555-1234

416 555 1234

Test country-specific assumptions

e.g. date order: 5/25/08 vs. 25/5/08



Limits of Use Cases as Test Cases

Use Case Tests good for:

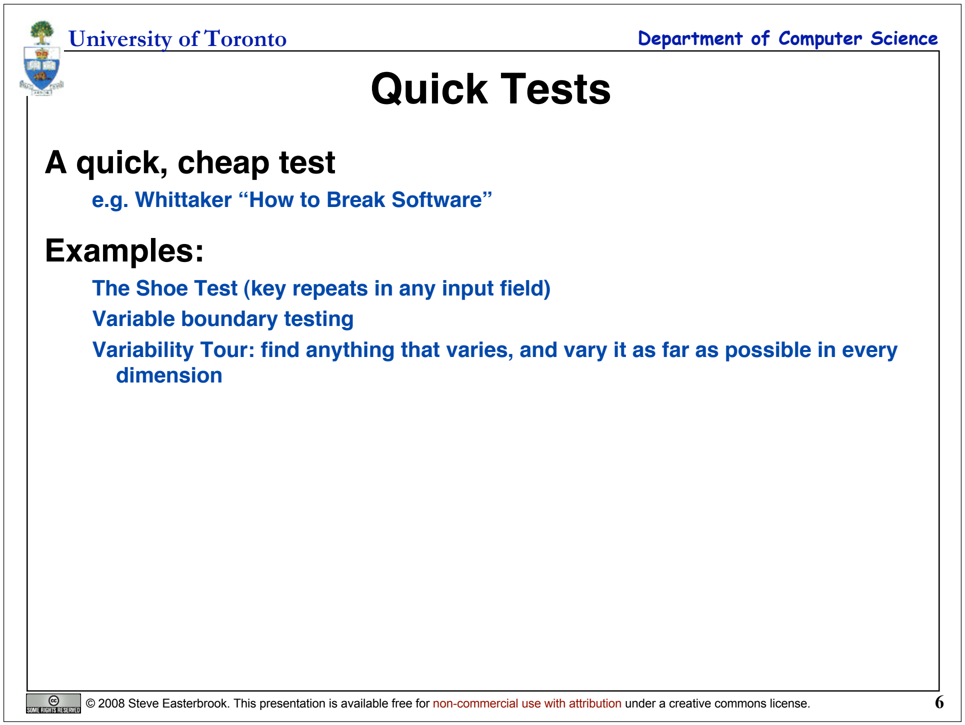
- User acceptance testing
- “Business as usual” functional testing
- Manual black-box tests
- Recording automated scripts for common scenarios

Limitations of Use Cases

- Likely to be incomplete
- Use cases don’t describe enough detail of use
- Gaps and inconsistencies between use cases
- Use cases might be out of date
- Use cases might be ambiguous

Defects you won’t discover:

- System errors (e.g. memory leaks)
- Things that corrupt persistent data
- Performance problems
- Software compatibility problems
- Hardware compatibility problems



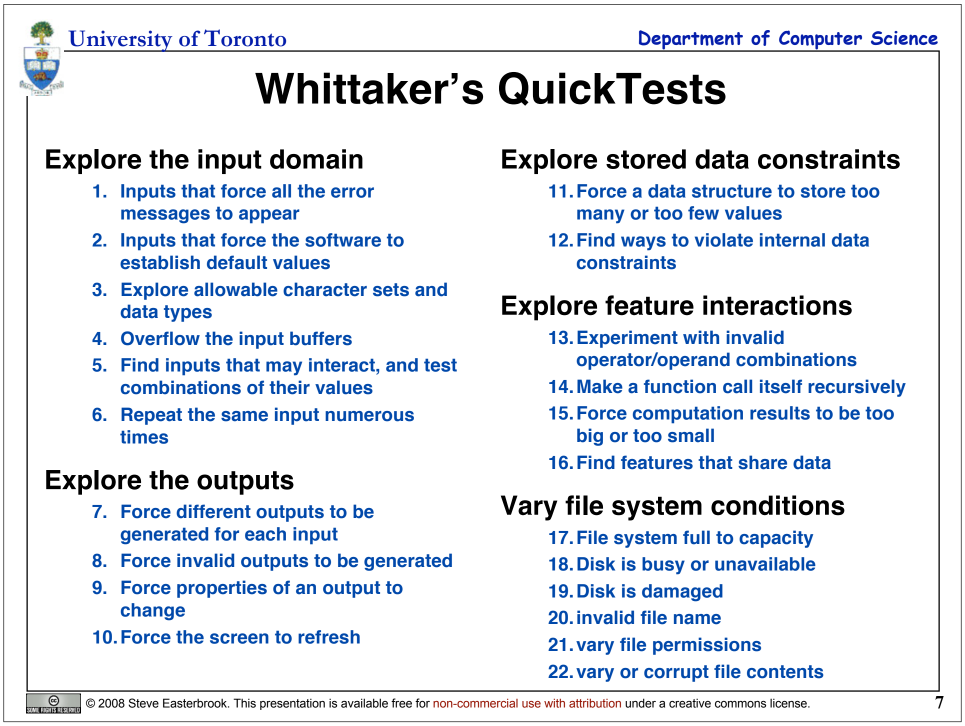
Quick Tests

A quick, cheap test

- e.g. Whittaker “How to Break Software”

Examples:

- The Shoe Test (key repeats in any input field)
- Variable boundary testing
- Variability Tour: find anything that varies, and vary it as far as possible in every dimension



Whittaker's QuickTests

Explore the input domain

1. Inputs that force all the error messages to appear
2. Inputs that force the software to establish default values
3. Explore allowable character sets and data types
4. Overflow the input buffers
5. Find inputs that may interact, and test combinations of their values
6. Repeat the same input numerous times

Explore the outputs

7. Force different outputs to be generated for each input
8. Force invalid outputs to be generated
9. Force properties of an output to change
10. Force the screen to refresh

Explore stored data constraints

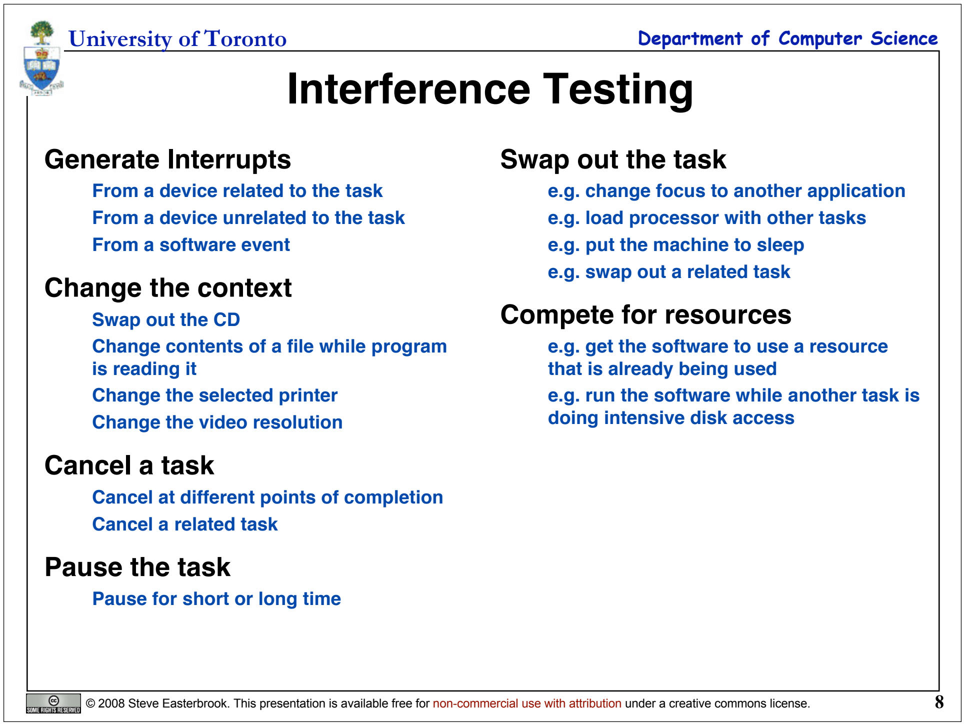
11. Force a data structure to store too many or too few values
12. Find ways to violate internal data constraints

Explore feature interactions

13. Experiment with invalid operator/operand combinations
14. Make a function call itself recursively
15. Force computation results to be too big or too small
16. Find features that share data

Vary file system conditions

17. File system full to capacity
18. Disk is busy or unavailable
19. Disk is damaged
20. invalid file name
21. vary file permissions
22. vary or corrupt file contents



Interference Testing

Generate Interrupts

From a device related to the task

From a device unrelated to the task

From a software event

Change the context

Swap out the CD

Change contents of a file while program is reading it

Change the selected printer

Change the video resolution

Cancel a task

Cancel at different points of completion

Cancel a related task

Pause the task

Pause for short or long time

Swap out the task

e.g. change focus to another application

e.g. load processor with other tasks

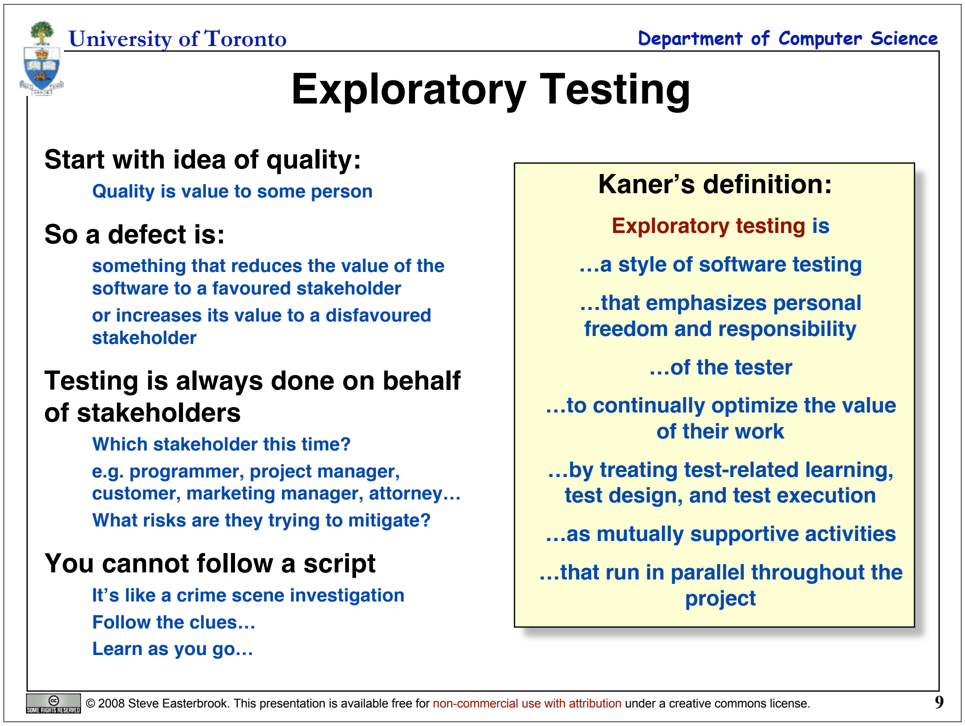
e.g. put the machine to sleep

e.g. swap out a related task

Compete for resources

e.g. get the software to use a resource that is already being used

e.g. run the software while another task is doing intensive disk access



Exploratory Testing

Start with idea of quality:

- Quality is value to some person

So a defect is:

- something that reduces the value of the software to a favoured stakeholder
- or increases its value to a disfavoured stakeholder

Testing is always done on behalf of stakeholders

- Which stakeholder this time?
- e.g. programmer, project manager, customer, marketing manager, attorney…
- What risks are they trying to mitigate?

You cannot follow a script

- It's like a crime scene investigation
- Follow the clues…
- Learn as you go…

Kaner's definition:

Exploratory testing is

…a style of software testing

…that emphasizes personal freedom and responsibility

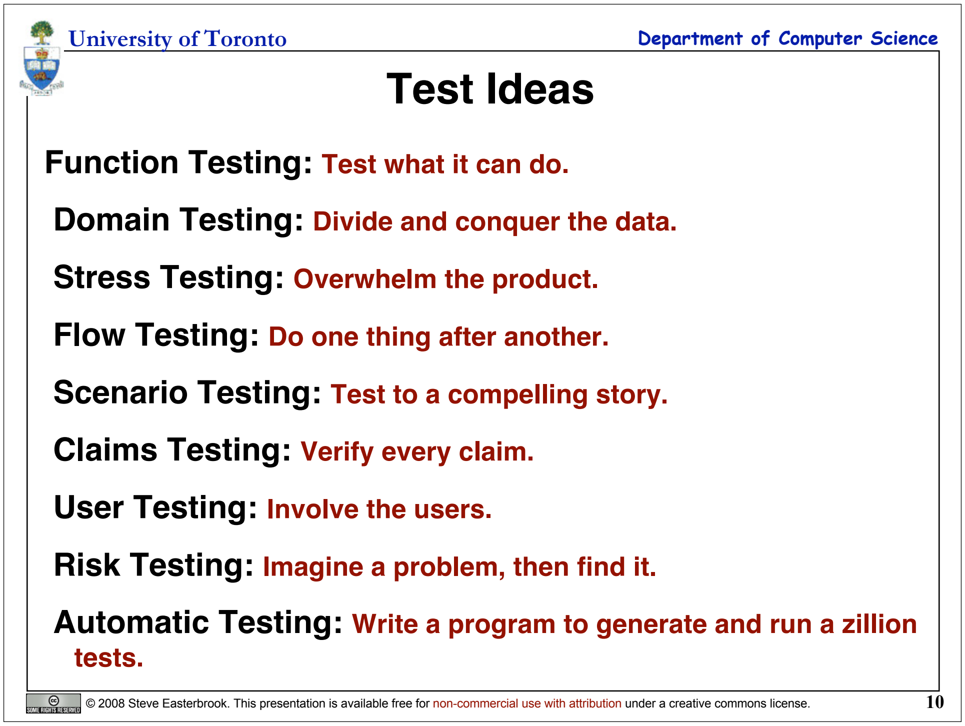
…of the tester

…to continually optimize the value of their work

…by treating test-related learning, test design, and test execution

…as mutually supportive activities

…that run in parallel throughout the project



Test Ideas

Function Testing: Test what it can do.

Domain Testing: Divide and conquer the data.

Stress Testing: Overwhelm the product.

Flow Testing: Do one thing after another.

Scenario Testing: Test to a compelling story.

Claims Testing: Verify every claim.

User Testing: Involve the users.

Risk Testing: Imagine a problem, then find it.

Automatic Testing: Write a program to generate and run a zillion tests.

When to stop testing?

***Source:** Adapted from Pfleeger 1998, p359*

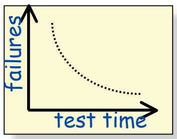
Motorola's Zero-failure testing model

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

empirical constants

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

testing time



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:

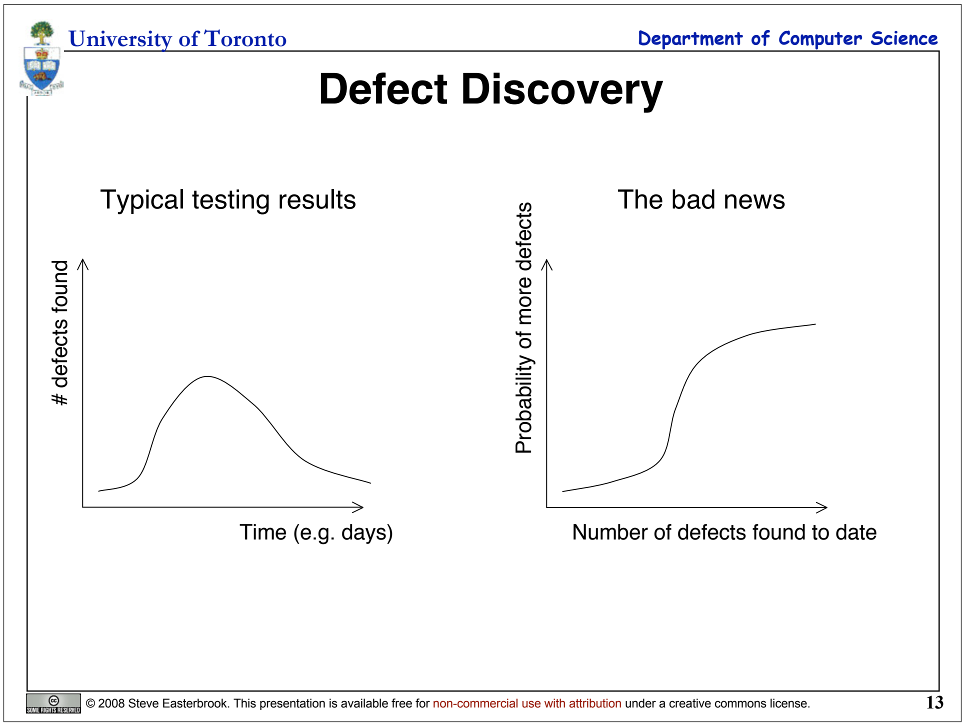
$$\frac{\text{Detected seeded faults}}{\text{Total seeded faults}} = \frac{\text{Detected nonseeded faults}}{\text{Total nonseeded faults}}$$

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

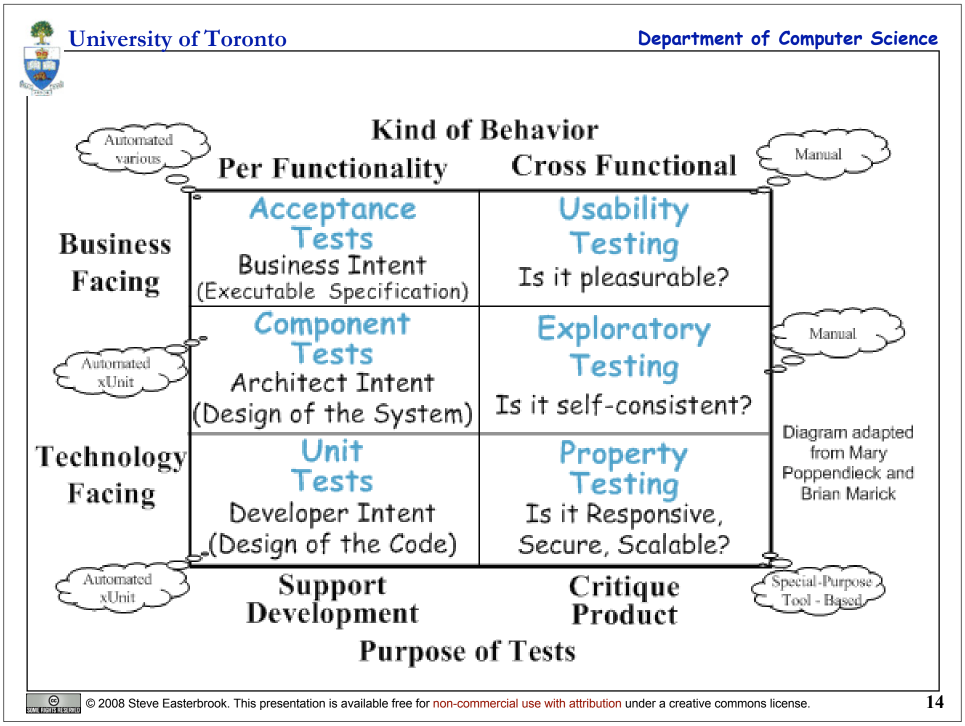
Alternatively

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

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



Defect Discovery



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