Computer Science
UNIVERSITY OF TORONTO

lecture 12: testing strategies

csc302h

winter 2014

(short) a2 interviews

Time	Team	Interviewer
10:10 a.m.	The Brogrammers	Andrew
10:10 a.m.	doge++	Mashiyat
10:10 a.m.	THE Group	Matt
--- short break ---		
10:17 a.m.	Missing Brackets	Andrew
10:17 a.m.	Seven - 2	Mashiyat
10:17 a.m.	Fantasix	Matt
--- short break ---		
10:24 a.m.	Solutions	Andrew
10:24 a.m.	Double Double	Mashiyat
10:24 a.m.	DOGE	Matt

recap from last time

introduction to testing

- multiple causes for defects: missing requirements, spec. error, bad design, bad algos, bad developers!
- defects (may) lead to failures, or, sometimes, go unnoticed.
- defect detection strategies & effectiveness:
  - formal design inspections & testing – 95%
  - agile informal review & regression – 90%
  - different costs, both useful depending on context

recap from last time (2)

- removing defects earlier is cheaper, sometimes by orders of magnitude!
- characteristics of good tests:
  - power: bug exists, test will find it
  - validity: no false-positives
  - non-redundancy: provides new information
  - repeatability: easy to re-run
  - etc. (don't memorize, but refer back when coming up with test plan)

recap from last time (3)

- type of test, to what it applies, & what it is testing:
  - unit test: unit of code, tested separately, generally applies to single use case or part of
  - integration: many (or all) units together, tests that code meets design specs.
  - functional test: coverage of all inputs (inc. edge/ corner cases), tests functional req's.
  - performance: tests (one of the) quality requirements
  - acceptance: customer goals
  - installation: user environment (optional depending on context)
  - …

testing strategies

Testing Strategies

Structural Coverage Strategies (White box testing):

- Statement Coverage
- Branch Coverage
- Condition Coverage
- Data Path Coverage

Function Coverage Strategies (Black box testing):

- Use Cases as Test Cases
- Testing with good and bad data

Stress Testing

- Quick Test
- Interference Testing

A radical alternative: Exploratory Testing

Developer Testing

Write the test cases first

- minimize the time to defect discovery
- forces you to think carefully about the requirements first
- exposes requirements problems early
- supports a "daily smoke test"

But: Limitations of Developer Testing

- Emphasis on clean tests (vs. dirty tests)
  - immature organisations have 1 dirty : 5 clean
  - mature organisations have 5 dirty : 1 clean
- Developers overestimate test coverage
- Developers tend to focus on statement coverage rather than …

Summary:

- Test-case first strategy is extremely valuable
- Test-case first strategy is not enough

Structured Basis Testing

Source: Adapted from McConnell 2004, p506-508

The minimal set of tests to cover every branch

How many tests?

start with 1 for the straight path

add 1 for each of these keywords: if, while, repeat, for, and, or

add 1 for each branch of a case statement

Example

```
int midval (int x, y, z) {
/* effects: returns median
   value of the three inputs
*/
if (x > y) {
   if (x > z) return x
   else return z }
else {
   if (y > z) return y
   else return z } }
```

Count 1 + 3 'if' s = 4 test cases

Now choose the cases to exercise the 4 paths:

e.g. x=3, y=2, z=1

x=3, y=2, z=4

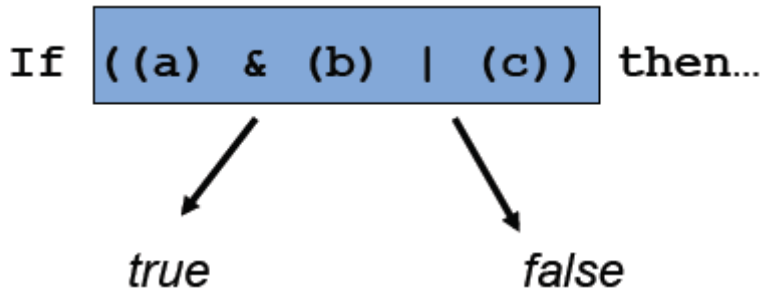
x=2, y=3, z=2

x=2, y=3, z=4

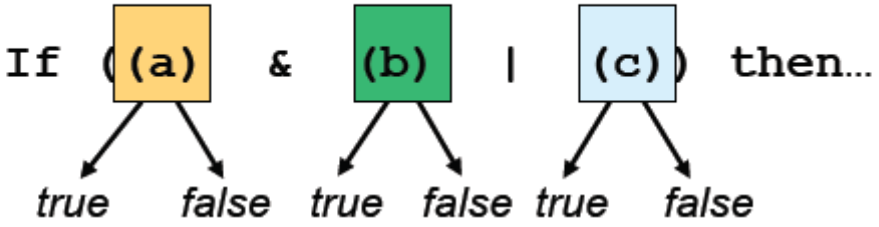
Complex Conditions

Source: Adapted from Christopher Ackermann's slides

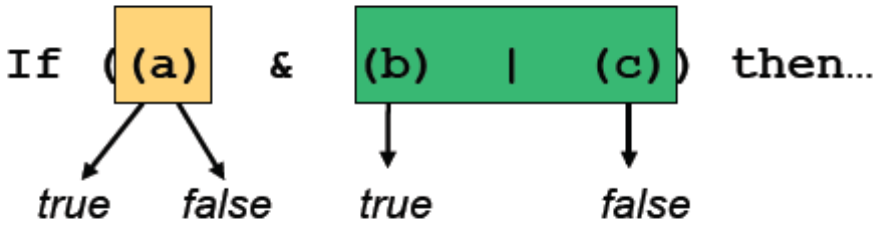
Branch Coverage:



Condition Coverage:



But can you show that each part has an independent effect on the outcome?



Dataflow testing

Source: Adapted from McConnell 2004, p506-508

Things that happen to data:

- Defined - data is initialized but not yet used
- Used - data is used in a computation
- Killed - space is released
- Entered - working copy created on entry to a method
- Exited - working copy removed on exit from a method

Normal life:

Defined once, Used a number of times, then Killed

Potential Defects:

- D-D: variable is defined twice
- D-Ex, D-K: variable defined but not used
- En-K: destroying a local variable that wasn't defined?
- En-U: for local variable, used before it's initialized
- K-K: unnecessary killing - can hang the machine?
- K-U: using data after it has been destroyed
- U-D: redefining a variable after is has been used

Testing all D-U paths

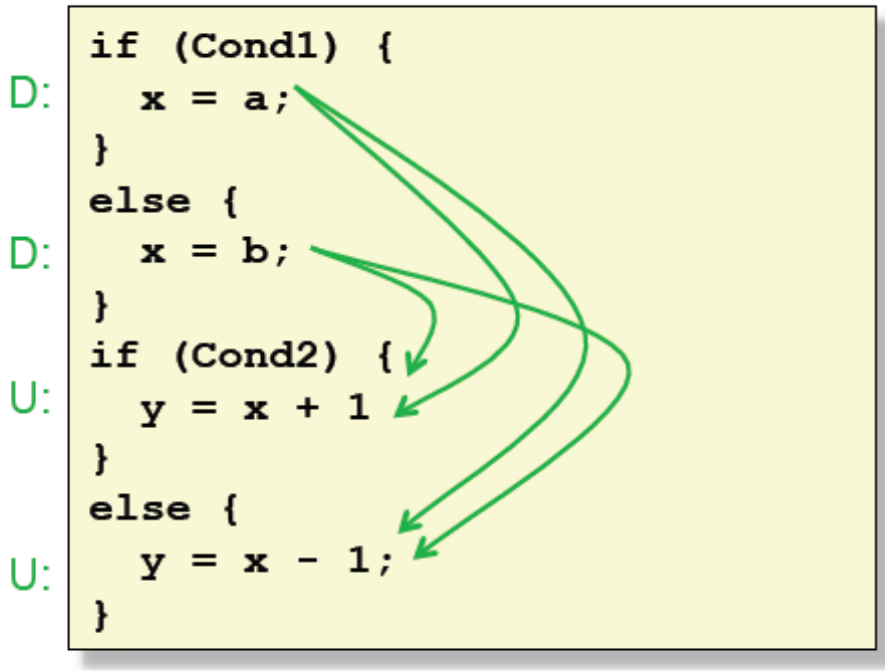
Source: Adapted from McConnell 2004, p506-508

The minimal set of tests to cover every D-U path

How many tests?

1 test for each path from each definition to each use of the variable

Example



Structured Basis Testing:

2 test cases is sufficient

Case 1: Cond1=true, Cond2=true

Case 2: Cond1=false, Cond2=false

All DU testing:

Need 4 test cases

Boundary Checking

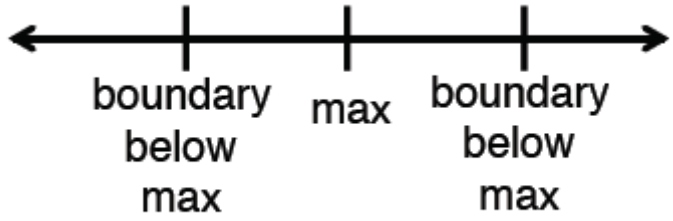
Source: Adapted from McConnell 2004, p506-508

Boundary Analysis

Every boundary needs 3 tests:

Example:

```
if (x < MAX) {
  … }
```



Add a test case for 3 values of x: MAX+1, MAX-1 and MAX

Compound Boundaries

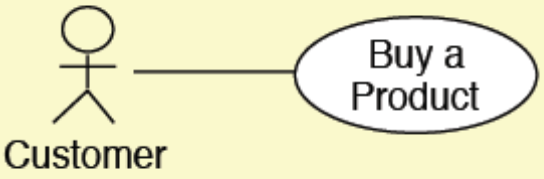
When several variables have combined boundaries

```
for (i=0; i<Num; i++) {
  if (a[i] < LIMIT) {
    y = y+a[i];
  }
}
```

Test when lots of array entries are close to LIMIT?

Test when lots of entries are close to zero?

Generating Tests from Use Cases



Buy a Product

Precondition: Customer has successfully logged in

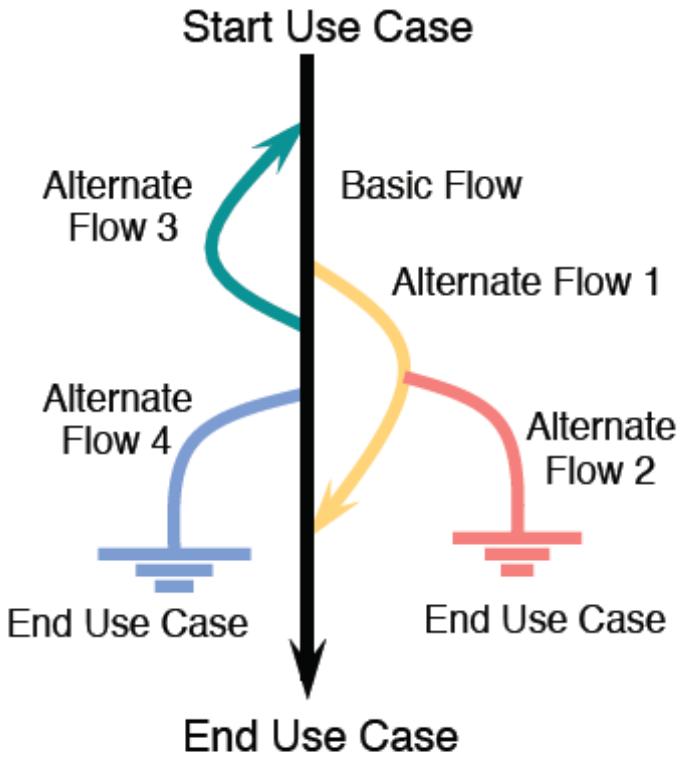
Main Success Scenario:
1. Customer browses catalog and selects items to buy
2. Customer goes to check out
3. Customer fills in shipping information (address, 1-day or 3-day)
4. System presents full pricing information
5. Customer fills in credit card information
6. System authorizes purchase
7. System confirms sale immediately
8. System sends confirming email to customer

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

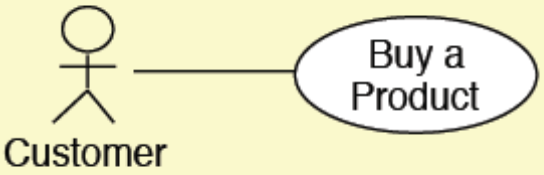
Extensions:
3a: Customer is Regular Customer:
.1 System displays current shipping, pricing and billing information
.2 Customer may accept or override defaults, cont MSS at step 6
6a: System fails to authorize credit card
.1 Customer may reenter credit card information or may cancel

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:
1. Customer browses catalog and selects items to buy
2. Customer goes to check out
3. Customer fills in shipping information (address, 1-day or 3-day)
4. System presents full pricing information
5. Customer fills in credit card information
6. System authorizes purchase
7. System confirms sale immediately
8. 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 defaults, cont 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 input choice

- select combinations of options for each test case

Data Classes

Source: Adapted from McConnell 2004, p506-508

Classes of Bad Data

- Too little data (or no data)
- Too much data
- The wrong kind of data (invalid data)
- The wrong size of data
- Uninitialized data

Classes of Good Data

- Nominal cases - middle of the road, expected values
- Minimum normal configuration
- Maximum normal configuration
- Compatibility with old data

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 cause 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: 3/15/12 vs. 15/3/12

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

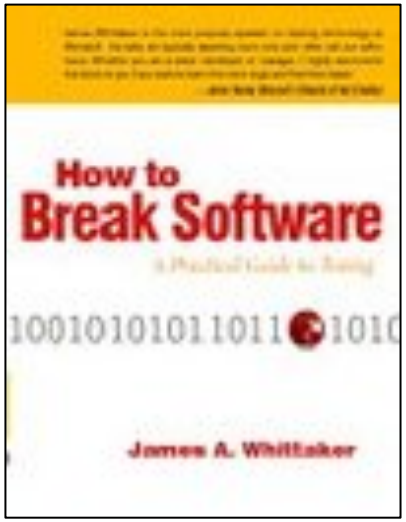
e.g. Whittaker "How to Break Software"

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

- Change focus to another application
- Load the processor with other tasks
- Put the machine to sleep
- Swap out a related task

Compete for resources

- Get the software to use a resource that is already being used
- 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

Things to Explore

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.