Course Review

Ajitha Rajan
Software Faults, Errors & Failures

- **Software Fault**: A static defect in the software

- **Software Failure**: External, incorrect behavior with respect to the requirements or other description of the expected behavior

- **Software Error**: An incorrect internal state that is the manifestation of some fault
A tester’s goal is to eliminate faults as early as possible

- Improve quality
- Reduce cost
- Preserve customer satisfaction
Functional testing
Functional testing

- **Functional testing**: Deriving test cases from program specifications
  - *Functional* refers to the source of information used in test case design, not to what is tested
- **Also known as**:  
  - specification-based testing (from specifications)
  - black-box testing (no view of the code)
- **Functional specification** = description of intended program behavior
  - either formal or informal
Systematic vs Random Testing

- **Random (uniform):**
  - Pick possible inputs uniformly
  - Avoids designer bias
    - A real problem: The test designer can make the same logical mistakes and bad assumptions as the program designer (especially if they are the same person)
  - But treats all inputs as equally valuable

- **Systematic (non-uniform):**
  - Try to select inputs that are especially valuable
  - Usually by choosing representatives of classes that are apt to fail often or not at all

- Functional testing is systematic testing
Functional testing: exploiting the specification

- Functional testing uses the specification (formal or informal) to partition the input space
  - E.g., specification of “roots” program suggests division between cases with zero, one, and two real roots

- Test each category, and boundaries between categories
  - No guarantees, but experience suggests failures often lie at the boundaries (as in the “roots” program)
Combinatorial testing
Combinatorial testing: Basic idea

- Identify distinct attributes that can be varied
  - In the data, environment, or configuration
  - Example: browser could be “IE” or “Firefox”, operating system could be “Vista”, “XP”, or “OSX”

- Systematically generate combinations to be tested
  - Example: IE on Vista, IE on XP, Firefox on Vista, Firefox on OSX, …

- Rationale: Test cases should be varied and include possible “corner cases”
Key ideas in combinatorial approaches

- **Category-partition testing**
  - separate (manual) identification of values that characterize the input space from (automatic) generation of combinations for test cases

- **Pairwise testing**
  - systematically test interactions among attributes of the program input space with a relatively small number of test cases

- **Catalog-based testing**
  - aggregate and synthesize the experience of test designers in a particular organization or application domain, to aid in identifying attribute values
Category partition (manual steps)

1. Decompose the specification into independently testable features
   - for each feature identify
     • parameters
     • environment elements
   - for each parameter and environment element identify elementary characteristics (categories)

2. Identify relevant values
   - for each characteristic (category) identify (classes of) values
     • normal values
     • boundary values
     • special values
     • error values

3. Introduce constraints
Example: Display Control

No constraints reduce the total number of combinations

432 (3x4x3x4x3) test cases

if we consider all combinations

<table>
<thead>
<tr>
<th>Display Mode</th>
<th>Language</th>
<th>Fonts</th>
<th>Color</th>
<th>Screen size</th>
</tr>
</thead>
<tbody>
<tr>
<td>full-graphics</td>
<td>English</td>
<td>Minimal</td>
<td>Monochrome</td>
<td>Hand-held</td>
</tr>
<tr>
<td>text-only</td>
<td>French</td>
<td>Standard</td>
<td>Color-map</td>
<td>Laptop</td>
</tr>
<tr>
<td>limited-bandwidth</td>
<td>Spanish</td>
<td>Document-loaded</td>
<td>16-bit</td>
<td>Full-size</td>
</tr>
<tr>
<td></td>
<td>Portuguese</td>
<td></td>
<td>True-color</td>
<td></td>
</tr>
</tbody>
</table>
## Pairwise combinations: 17 test cases

<table>
<thead>
<tr>
<th>Language</th>
<th>Color</th>
<th>Display Mode</th>
<th>Fonts</th>
<th>Screen Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>English</td>
<td>Monochrome</td>
<td>Full-graphics</td>
<td>Minimal</td>
<td>Hand-held</td>
</tr>
<tr>
<td>English</td>
<td>Color-map</td>
<td>Text-only</td>
<td>Standard</td>
<td>Full-size</td>
</tr>
<tr>
<td>English</td>
<td>16-bit</td>
<td>Limited-bandwidth</td>
<td>-</td>
<td>Full-size</td>
</tr>
<tr>
<td>English</td>
<td>True-color</td>
<td>Text-only</td>
<td>Document-loaded</td>
<td>Laptop</td>
</tr>
<tr>
<td>French</td>
<td>Monochrome</td>
<td>Limited-bandwidth</td>
<td>Standard</td>
<td>Laptop</td>
</tr>
<tr>
<td>French</td>
<td>Color-map</td>
<td>Full-graphics</td>
<td>Document-loaded</td>
<td>Full-size</td>
</tr>
<tr>
<td>French</td>
<td>16-bit</td>
<td>Text-only</td>
<td>Minimal</td>
<td>-</td>
</tr>
<tr>
<td>French</td>
<td>True-color</td>
<td>-</td>
<td>-</td>
<td>Hand-held</td>
</tr>
<tr>
<td>Spanish</td>
<td>Monochrome</td>
<td>-</td>
<td>Document-loaded</td>
<td>Full-size</td>
</tr>
<tr>
<td>Spanish</td>
<td>Color-map</td>
<td>Limited-bandwidth</td>
<td>Minimal</td>
<td>Hand-held</td>
</tr>
<tr>
<td>Spanish</td>
<td>16-bit</td>
<td>Full-graphics</td>
<td>Standard</td>
<td>Laptop</td>
</tr>
<tr>
<td>Spanish</td>
<td>True-color</td>
<td>Text-only</td>
<td>-</td>
<td>Hand-held</td>
</tr>
<tr>
<td>Portuguese</td>
<td>-</td>
<td>-</td>
<td>Monochrome</td>
<td>Text-only</td>
</tr>
<tr>
<td>Portuguese</td>
<td>Color-map</td>
<td>-</td>
<td>Minimal</td>
<td>Laptop</td>
</tr>
<tr>
<td>Portuguese</td>
<td>16-bit</td>
<td>Limited-bandwidth</td>
<td>Document-loaded</td>
<td>Hand-held</td>
</tr>
<tr>
<td>Portuguese</td>
<td>True-color</td>
<td>Full-graphics</td>
<td>Minimal</td>
<td>Full-size</td>
</tr>
<tr>
<td>Portuguese</td>
<td>True-color</td>
<td>Limited-bandwidth</td>
<td>Standard</td>
<td>Hand-held</td>
</tr>
</tbody>
</table>
Next …

- Category-partition approach gives us …
  - Separation between (manual) identification of parameter characteristics and values and (automatic) generation of test cases that combine them
  - Constraints to reduce the number of combinations

- Pairwise (or n-way) testing gives us …
  - Much smaller test suites, even without constraints
    - (but we can still use constraints)

- We still need …
  - Help to make the manual step more systematic
Catalog based testing

- Deriving value classes requires human judgment
- Gathering experience in a systematic collection can:
  - speed up the test design process
  - routinize many decisions, better focusing human effort
  - accelerate training and reduce human error
- Catalogs capture the experience of test designers by listing important cases for each possible type of variable
  - Example: if the computation uses an integer variable a catalog might indicate the following relevant cases
    - The element immediately preceding the lower bound
    - The lower bound of the interval
    - A non-boundary element within the interval
    - The upper bound of the interval
    - The element immediately following the upper bound
Catalog based testing process

Step 1:
Analyze the initial specification to identify simple elements:
- Pre-conditions
- Post-conditions
- Definitions
- Variables
- Operations

Step 2:
Derive a first set of test case specifications from pre-conditions, post-conditions and definitions

Step 3:
Complete the set of test case specifications using test catalogs
Finite Models
public static String collapseNewlines(String argStr) {
    char last = argStr.charAt(0);
    StringBuffer argBuf = new StringBuffer();

    for (int cldx = 0; cldx < argStr.length(); cldx++) {
        char ch = argStr.charAt(cldx);
        if (ch != 'n' || last != 'n') {
            argBuf.append(ch);
            last = ch;
        }
    }
    return argBuf.toString();
}
Structural Testing
“Structural” testing

- Judging test suite thoroughness based on the structure of the program itself
  - Also known as “white-box”, “glass-box”, or “code-based” testing
  - To distinguish from functional (requirements-based, “black-box” testing)
    - “Structural” testing is still testing product functionality against its specification. Only the measure of thoroughness has changed.
Structural testing *complements* functional testing

- Control flow testing includes cases that may not be identified from specifications alone
  - Typical case: implementation of a single item of the specification by multiple parts of the program
  - Example: hash table collision (invisible in interface spec)

- Test suites that satisfy control flow adequacy criteria could fail in revealing faults that can be caught with functional criteria
  - Typical case: missing path faults
Subsumption relation

THEORETICAL CRITERIA

Cyclomatic testing

Boundary interior testing

Compound condition testing

MC/DC testing

LCSAJ testing

Branch and condition testing

Branch testing

Basic condition testing

Statement testing

Loop boundary testing

(c) 2007 Mauro Pezzè & Michal Young
Summary

- We defined a number of adequacy criteria
  - NOT test design techniques!
- Different criteria address different classes of errors
- Full coverage is usually unattainable
  - Remember that attainability is an undecidable problem!
- ...and when attainable, “inversion” is usually hard
  - How do I find program inputs allowing to cover something buried deeply in the CFG?
  - Automated support (e.g., symbolic execution) may be necessary
- Therefore, rather than requiring full adequacy, the “degree of adequacy” of a test suite is estimated by coverage measures
  - May drive test improvement
Write tests that provide statement, branch, and basic condition coverage over the following code:

```c++
int search(string A[], int N, string what){
    int index = 0;
    if ((N == 1) && (A[0] == what)){
        return 0;
    } else if (N == 0){
        return -1;
    } else if (N > 1){
        while(index < N){
            if (A[index] == what)
                return index;
            else
                index++;
        }
    }
    return -1;
}
```
Write tests that provide statement, branch, and basic condition coverage over the following code:

```
index = 0
(N == 1) && (A[0] = what) 
return 0;
N == 0
False
True
return -1;
N > 1
False
return -1;
index < N
True
A[index] == what
True
return index;
```

```
False
index++;
```
Write tests that provide statement, branch, and basic condition coverage over the following code:

```
index=0
(N==1) &&
(A[0] = what)
return 0;
N==0
False
True
N>1
False
return -1;
index
< N
return -1;
A[index]
== what
True
return index;
index++;
return -1;
```

1: A[“Bob”, “Jane”], 2, “Jane”
2: A[“Bob”, “Jane”], 2, “Spot”
3: A[], 0, “Bob”
4. A[“Bob”], 1, “Bob”
5. A[“Bob”], 1, “Spot”
Dependence and Data Flow Models
Def-Use Pairs

\[
\text{... if (...) \{} \\
\text{\hspace{1em} x = ... ;} \\
\text{... \}} \\
\text{y = ... + x + ... ;}
\]
**Definition-Clear or Killing**

- **A**: def x
  
  - x = ...
  
  - q = ...
  
  - x = y; // B: kill x, def x
  
  - z = ...
  
  - y = f(x); // C: use x

Path A..C is not definition-clear

Path B..C is definition-clear

- **B**: x = y
  
  - B: kill x, def x
  
  - y = f(x); // C: use x

- **C**: y = f(x)
  
  - C: use x

**Use**: the value of x is extracted

**Definition**: x gets a new value, old value is killed

(c) 2007 Mauro Pezzè & Michal Young Ch 6, slide 8
Data flow testing
Terms

• DU pair: a pair of *definition* and *use* for some variable, such that at least one DU path exists from the definition to the use
  
  \[ x = \ldots \text{ is a *definition* of } x \]
  \[ = \ldots x \ldots \text{ is a *use* of } x \]

• DU path: a definition-clear path on the CFG starting from a definition to a use of a same variable
  - Definition clear: Value is not replaced on path
  - Note - loops could create infinite DU paths between a def and a use
Adequacy criteria

- All DU pairs: Each DU pair is exercised by at least one test case
- All DU paths: Each *simple* (non looping) DU path is exercised by at least one test case
- All definitions: For each definition, there is at least one test case which exercises a DU pair containing it
  - (Every computed value is used somewhere)

Corresponding coverage fractions can also be defined

(c) 2007 Mauro Pezzè & Michal Young
Model based testing
...to a finite state machine...
Testing Object Oriented Software

Chapter 15
Characteristics of OO Software

Typical OO software characteristics that impact testing

- State dependent behavior
- Encapsulation
- Inheritance
- Polymorphism and dynamic binding
- Abstract and generic classes
- Exception handling
Interclass Testing

- The first level of integration testing for object-oriented software
  - Focus on interactions between classes
- Bottom-up integration according to “depends” relation
  - A depends on B: Build and test B, then A
- Start from use/include hierarchy
  - Implementation-level parallel to logical “depends” relation
    - Class A makes method calls on class B
    - Class A objects include references to class B methods
      - but only if reference means “is part of”

(c) 2008 Mauro Pezzè & Michal Young
from a class diagram...
....to a hierarchy

Note: we may have to break loops and generate stubs
**Intra**class data flow testing

- Exercise sequences of methods
  - From setting or modifying a field value
  - To using that field value

- We need a control flow graph that encompasses more than a single method ...
The intraclass control flow graph

Control flow for each method
+ node for class
+ edges from node class to the start nodes of the methods from the end nodes of the methods to node class

=> control flow through sequences of method calls
Mutation Testing
Example of Mutation Operators I

- Constant replacement
- Scalar variable replacement
- Scalar variable for constant replacement
- Constant for scalar variable replacement
- Array reference for constant replacement
- Array reference for scalar variable replacement
- Constant for array reference replacement
- Scalar variable for array reference replacement
- Array reference for array reference replacement
- Source constant replacement
- Data statement alteration
- Comparable array name replacement
- Arithmetic operator replacement
- Relational operator replacement
- Logical connector replacement
- Absolute value insertion
- Unary operator insertion
- Statement deletion
- Return statement replacement
Regression Testing

Ajitha Rajan
Example

Version 1

- Feature A
- Feature B

Tests

Version 2

- Feature A
- Feature B
- Feature C

Old Tests + New Tests

Regression Tests for the next version
Regression Test Optimization

➔ Re-test All
➔ Regression Test Selection
➔ Regression Test Set Minimisation
➔ Regression Test Set Prioritisation
Integration and Component-based Software Testing
## What is integration testing?

<table>
<thead>
<tr>
<th></th>
<th>Module test</th>
<th>Integration test</th>
<th>System test</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Specification:</strong></td>
<td>Module interface</td>
<td>Interface specs, module breakdown</td>
<td>Requirements specification</td>
</tr>
<tr>
<td><strong>Visible structure:</strong></td>
<td>Coding details</td>
<td>Modular structure (software architecture)</td>
<td>— none —</td>
</tr>
<tr>
<td><strong>Scaffolding required:</strong></td>
<td>Some</td>
<td>Often extensive</td>
<td>Some</td>
</tr>
<tr>
<td><strong>Looking for faults in:</strong></td>
<td>Modules</td>
<td>Interactions, compatibility</td>
<td>System functionality</td>
</tr>
</tbody>
</table>
Write stubs of called or used modules at each step in construction.
Bottom Up ..

... but we must construct drivers for each module (as in unit testing) ..
System, Acceptance, and Regression Testing
System Testing

• Key characteristics:
  - Comprehensive (the whole system, the whole spec)
  - Based on specification of observable behavior
    Verification against a requirements specification, not validation, and not opinions
  - Independent of design and implementation

*Independence*: Avoid repeating software design errors in system test design
Global Properties

• Some system properties are inherently global
  - Performance, latency, reliability, ...
  - Early and incremental testing is still necessary, but provide only estimates

• A major focus of system testing
  - The only opportunity to verify global properties against actual system specifications
  - Especially to find unanticipated effects, e.g., an unexpected performance bottleneck
Context-Dependent Properties

- Beyond system-global: Some properties depend on the system context and use
  - Example: Performance properties depend on environment and configuration
  - Example: Privacy depends both on system and how it is used
    - Medical records system must protect against unauthorized use, and authorization must be provided only as needed
  - Example: Security depends on threat profiles
    - And threats change!

- Testing is just one part of the approach
22.3 Acceptance testing
Estimating Dependability

- Measuring quality, not searching for faults
  - Fundamentally different goal than systematic testing
- Quantitative dependability goals are statistical
  - Reliability
  - Availability
  - Mean time to failure
  - ...
- Requires valid statistical samples from operational profile
  - Fundamentally different from systematic testing
V-model

Requirements Analysis

System Design

Object Design

Coding

System Testing

Unit Testing

Acceptance Testing

is validated by

build system

validate system
eXtreme Programming (XP)

http://www.extremeprogramming.org/map/project.html
HOW DOES TDD HELP
TDD CYCLE

- **Write Test Code**
  - Guarantees that every functional code is testable
  - Provides a specification for the functional code
  - Helps to think about design
  - Ensure the functional code is tangible

- **Write Functional Code**
  - Fulfill the requirement (test code)
  - Write the simplest solution that works
  - Leave Improvements for a later step
  - The code written is only designed to pass the test
    - no further (and therefore untested code is not created).

- **Refactor**
  - Clean-up the code (test and functional)
  - Make sure the code expresses intent
  - Remove code smells
  - Re-think the design
  - Delete unnecessary code
Principle of TDD (In Practice)

Start

Write a Test

Run the Test

Write (just enough)
Dev Code to compile

Run the Test

Write (just enough)
Dev Code to pass

Run the Test

Refactoring

TDD

Red

See it fail
because there's
no dev code

Green

See it fail
because no logic
is implemented

Refactor

See the
test pass
Security testing vs “regular” testing

- “Regular” testing aims to ensure that the program meets customer requirements in terms of features and functionality.
- Tests “normal” use cases
  - Test with regards to common expected usage patterns.
- Security testing aims to ensure that program fulfills security requirements.
  - Often non-functional.
  - More interested in misuse cases
    - Attackers taking advantage of “weird” corner cases.
Common security testing approaches

Often difficult to craft e.g. unit tests from non-functional requirements

Two common approaches:

- Test for known vulnerability types
- Attempt directed or random search of program state space to uncover the “weird corner cases”

In today’s lecture:

- Penetration testing (briefly)
- Fuzz testing or “fuzzing”
- Concolic testing
Fuzz testing architecture

- Fuzzer generates inputs to SUT
- Dispatcher responsible for running SUT with input from fuzzer
- Assessor examines behavior of SUT to detect failures (i.e. signs of triggered bugs)
Fuzzing components: Input generation

Simplest method: Completely random

- Won’t work well in practice – Input deviates too much from expected format, rejected early in processing.

Two common methods:

- Mutation based fuzzing
- Generation based fuzzing
Fuzzing outlook

- Mutation-based fuzzing can typically only find the “low-hanging fruit” – shallow bugs that are easy to find
- Generation-based fuzzers almost invariably gives better coverage, but requires much more manual effort

- Current research in fuzzing attempts to combine the “fire and forget” nature of mutation-based fuzzing and the coverage of generation-based.
  - **Evolutionary fuzzing** combines mutation with genetic algorithms to try to “learn” the input format automatically. Recent successful example is “American Fuzzy Lop” (AFL)
  - **Whitebox fuzzing** generates test cases based on the control-flow structure of the SUT. Our next topic…
Concolic testing

Idea: Combine concrete and symbolic execution

- Concolic execution (CONCrete and symbOLIC)

Concolic execution workflow:

1. Execute the program for real on some input, and record path taken.
2. Encode path as query to SMT solver and negate one branch condition.
3. Ask the solver to find new satisfying input that will give a different path.

Reported bugs are always accompanied by an input that triggers the bug (generated by SMT solver).

⇒ Complete – Reported bugs are always real bugs.
Greybox fuzzing

- Probability of hitting a “deep” level of the code decreases exponentially with the “depth” of the code for mutation based fuzzing.

- Similarly, the time required for solving an SMT query is high, and increases exponentially with the depth of the path constraint.

- Black-box fuzzing is too “dumb” and whitebox fuzzing may be “too smart”
  - Idea of greybox fuzzing is to find a sweet spot in between.

```c
if(condition1)
  if(condition2)
    if(condition3)
      if(condition4)
        bug();
```