A Talk in 3 Parts

1. Why do we test?

2. What should we do during testing?

3. How do we get to this future of testing?

We are in the middle of a revolution in how software is tested

Research is finally meeting practice
Here! Test This!

My first “professional” job

A stack of computer printouts—and no documentation

Cost of Testing

You’re going to spend about half of your development budget on testing, whether you want to or not.

- In real-world usage, testing is the principle post-design activity
- Restricting early testing usually increases cost
- Extensive hardware-software integration requires more testing
Part 1: Why Test?

If you don’t know why you’re conducting a test, it won’t be very helpful.

- Written test objectives and requirements are rare
- How much testing is enough?
- Common objective – spend the budget …

Why Test?

If you don’t start planning for the test at the time the functional requirements are formed, you’ll never know why you’re conducting the test.

- “The software shall be easily maintainable”
- Threshold reliability requirements?
- Requirements definition teams should include testers!
Cost of Not Testing

Program Managers often say: “Testing is too expensive.”

- Not testing is even more expensive
- Planning for testing after development is prohibitively expensive
- A test station for circuit boards costs half a million dollars …
- Software test tools cost less than $10,000 !!!

Caveat: Impact of New Tools and Techniques

They’re teaching a new way of plowing over at the Grange tonight - you going?

Naw - I already don’t plow as good as I know how...

“Knowing is not enough, we must apply. Willing is not enough, we must do.”

Goethe
Part 2: What?

But ... what should we do?

Testing in the 21st Century

- We are going through a time of change
- Software Defines Behavior
  - network routers
  - financial networks
  - telephone switching networks
  - other infrastructure
- Embedded Control Applications
  - airplanes
  - spaceships
  - air traffic control systems
  - watches
  - ovens
  - remote controllers
  - PDAs
  - memory seats
  - DVD players
  - garage door openers
  - cell phones
- Safety critical, real-time software
- And of course ... web apps must be highly reliable!
Important Terms
Validation & Verification

- **Validation**: The process of evaluating software at the end of software development to ensure compliance with intended usage

- **Verification**: The process of determining whether the products of a given phase of the software development process fulfill the requirements established during the previous phase

IV&V stands for “independent verification and validation”

Test Engineer & Test Managers

- **Test Engineer**: An IT professional who is in charge of one or more technical test activities
  - designing test inputs
  - producing test values
  - running test scripts
  - analyzing results
  - reporting results to developers and managers

- **Test Manager**: In charge of one or more test engineers
  - sets test policies and processes
  - interacts with other managers on the project
  - otherwise helps the engineers do their work
Test Engineer Activities

Static and Dynamic Testing

- **Static Testing**: Testing without executing the program.
  - This include software inspections and some forms of analyses.

- **Dynamic Testing**: Testing by executing the program with real inputs
Software Faults, Errors & Failures

- **Software Fault**: A static defect in the software
- **Software Error**: An incorrect internal state that is the manifestation of some fault
- **Software Failure**: External, incorrect behavior with respect to the requirements or other description of the expected behavior

**Faults in software are design mistakes and will always exist**

Testing & Debugging

- **Testing**: Finding inputs that cause the software to fail
- **Debugging**: The process of finding a fault given a failure
Fault & Failure Model

Three conditions necessary for a failure to be observed

1. **Reachability**: The location or locations in the program that contain the fault must be reached

2. **Infection**: The state of the program must be incorrect

3. **Propagation**: The infected state must propagate to cause some output of the program to be incorrect

Test Case

- **Test Case Values**: The values that directly satisfy one test requirement

- **Expected Results**: The result that will be produced when executing the test if the program satisfies its intended behavior
Observability and Controllability

- **Software Observability**: How easy it is to observe the behavior of a program in terms of its outputs, effects on the environment and other hardware and software components
  - Software that affects hardware devices, databases, or remote files have low observability

- **Software Controllability**: How easy it is to provide a program with the needed inputs, in terms of values, operations, and behaviors
  - Easy to control software with inputs from keyboards
  - Inputs from hardware sensors or distributed software is harder
  - Data abstraction reduces controllability and observability

Inputs to Affect Controllability and Observability

- **Prefix Values**: Any inputs necessary to put the software into the appropriate state to receive the test case values

- **Postfix Values**: Any inputs that need to be sent to the software after the test case values

- **Two types of postfix values**
  1. **Verification Values**: Values necessary to see the results of the test case values
  2. **Exit Commands**: Values needed to terminate the program or otherwise return it to a stable state

- **Executable Test Script**: A test case that is prepared in a form to be executed automatically on the test software and produce a report
Top-Down and Bottom-Up Testing

- **Top-Down Testing**: Test the main procedure, then go down through procedures it calls, and so on.

- **Bottom-Up Testing**: Test the leaves in the tree (procedures that make no calls), and move up to the root.
  - Each procedure is not tested until all of its children have been tested.

White-box and Black-box Testing

- **Black-box testing**: Deriving tests from external descriptions of the software, including specifications, requirements, and design.

- **White-box testing**: Deriving tests from the source code internals of the software, specifically including branches, individual conditions, and statements.

This view is really out of date.
The more general question is: *from what level of abstraction to we derive tests?*
Changing Notions of Testing

- Old view of testing is of testing at specific software development **phases**
  - Unit, module, integration, system ...

- New view is in terms of **structures** and **criteria**
  - Graphs, logical expressions, syntax, input space

Old: Testing at Different Levels

- **Acceptance testing**: Is the software acceptable to the user?
- **System testing**: Test the overall functionality of the system
- **Integration testing**: Test how modules interact with each other
- **Module testing**: Test each class, file, module or component
- **Unit testing**: Test each unit (method) individually
Old: Find a Graph and Cover It

- Tailored to:
  - a particular software artifact
    - code, design, specifications
  - a particular phase of the lifecycle
    - requirements, specification, design, implementation

- This viewpoint obscures underlying similarities
- Graphs do not characterize all testing techniques well
- Four abstract models suffice

New: Test Coverage Criteria

A tester’s job is simple: Define a model of the software, then find ways to cover it

- Test Requirements: Specific things that must be satisfied or covered during testing
- Test Criterion: A collection of rules and a process that define test requirements

Testing researchers have defined dozens of criteria, but they are all really just a few criteria on four types of structures …
New: Criteria Based on Structures

Structures: Four ways to model software

1. Graphs

2. Logical Expressions

   (not X or not Y) and A and B

3. Input Domain Characterization

   A: \{0, 1, >1\}
   B: \{600, 700, 800\}
   C: \{swe, cs, isa, infs\}

4. Syntactic Structures

1. Graph Coverage – Structural

This graph may represent
- statements & branches
- methods & calls
- components & signals
- states and transitions

Path
"Cover every path"

- 12567
- 1257
- 13567
- 1357
- 1343567
- 134357 ...
1. Graph Coverage – Data Flow

This graph contains:
- **defs**: nodes & edges where variables get values
- **uses**: nodes & edges where values are accessed

### Defs & Uses Pairs
- (x, 1, (1,2)), (x, 1, (1,3))
- (y, 1, 4), (y, 1, 6)
- (a, 2, (5,6)), (a, 2, (5,7)), (a, 3, (5,6)), (a, 3, (5,7))
- (m, 4, 7), (m, 6, 7)

### All Uses

*Every def “reaches” every use*
- 1, 2, 5, 6, 7
- 1, 2, 5, 7
- 1, 3, 5, 6, 7
- 1, 3, 5, 7
- 1, 3, 4, 3, 5, 7

---

1. Graph - FSM Example

Memory Seats in a Lexus ES 300

- **Guard (safety constraint)**
- **Trigger (input)**

### Driver 1 Configuration
- [Ignition = off] | Button1
- [Ignition = off] | Button2

### Driver 2 Configuration
- [Ignition = off] | Button1
- [Ignition = on] | seatBack ()
- [Ignition = on] | seatBottom ()
- [Ignition = on] | lumbar ()
- [Ignition = on] | sideMirrors ()

### New Configuration Driver 1
- [Ignition = on] | Reset AND Button2

### New Configuration Driver 2
- [Ignition = on] | Reset AND Button2

### Modified Configuration
- [Ignition = on] | Reset AND Button1
2. Logical Expressions

\[(a > b) \text{ or } G \text{ ) and } (x < y)\]

- **Predicate Coverage**: Each predicate must be true and false
  - \((a > b) \text{ or } G \text{ ) and } (x < y)\) = True, False

- **Clause Coverage**: Each clause must be true and false
  - \(a > b\) = True, False
  - \(G\) = True, False
  - \(x < y\) = True, False

- **Combinatorial Coverage**: Various combinations of clauses
  - **Active Clause Coverage**: Each clause must determine the predicate’s result
2. Logic – Active Clause Coverage

\[(a > b) \text{ or } G \) and \((x < y)\]

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>F</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

With these values for \(G\) and \((x < y)\), \((a > b)\) determines the value of the predicate.

duplicate

3. Input Domain Characterization

- Describe the input domain of the software
  - Identify inputs, parameters, or other categorization
  - Partition each input into finite sets of representative values
  - Choose combinations of values

- **System level**
  - Number of students \[\{0, 1, >1\}\]
  - Level of course \[\{600, 700, 800\}\]
  - Major \[\{swe, cs, isa, infs\}\]

- **Unit level**
  - Parameters \(F\) (int \(X\), int \(Y\))
  - Possible values \(X: \{<0, 0, 1, 2, >2\}\), \(Y: \{10, 20, 30\}\)
  - Tests
    - \(F(-5, 10), F(0, 20), F(1, 30), F(2, 10), F(5, 20)\)
4. Syntactic Structures

- Based on a **grammar**, or other syntactic definition
- Primary example is **mutation testing**
  1. Induce small changes to the program: **mutants**
  2. Find tests that cause the mutant programs to fail: **killing mutants**
  3. Failure is defined as **different output** from the original program
  4. Check the **output** of useful tests on the original program

- Example program and mutants

```java
if (x > y)
    z = x - y;
else
    z = 2 * x;
```

```java
if (x > y)
    ∆if (x >= y)
    z = x - y;
    ∆z = x + y;
    ∆z = x - m;
else
    z = 2 * x;
```

---

**Coverage Overview**

**Four Structures for Modeling Software**

- **Graphs**
- **Logic**
- **Input Space**
- **Syntax**

- **Applied to**
  - **Source**
  - **Specs**
  - **Design**
  - **Use cases**
  - **Source**
  - **FSMs**
  - **Specs**
  - **DNF**
  - **Source**
  - **Models**
  - **Integ**
  - **Input**
Coverage

Given a set of test requirements $TR$ for coverage criterion $C$, a test set $T$ satisfies $C$ coverage if and only if for every test requirement $tr$ in $TR$, there is at least one test $t$ in $T$ such that $t$ satisfies $tr$

- **Infeasible test requirements**: test requirements that cannot be satisfied
  - No test case values exist that meet the test requirements
  - Dead code
  - Detection of infeasible test requirements is formally undecidable for most test criteria

- Thus, 100% coverage is impossible in practice

### Two Ways to Use Test Criteria

1. **Directly generate** test values to satisfy the criterion often assumed by the research community. Most obvious way to use criteria very hard without automated tools

2. Generate test values externally and measure against the criterion usually favored by industry
   - sometimes misleading
   - if tests do not reach 100% coverage, what does that mean?

   Test criteria are sometimes called **metrics**
Generators and Recognizers

- **Generator**: A procedure that automatically generates values to satisfy a criterion
- **Recognizer**: A procedure that decides whether a given set of test values satisfies a criterion

- Both problems are provably undecidable for most criteria
- It is possible to recognize whether test cases satisfy a criterion far more often than it is possible to generate tests that satisfy the criterion
- Coverage analysis tools are quite plentiful

Comparing Criteria with Subsumption

- **Criteria Subsumption**: A test criterion $C_1$ subsumes $C_2$ if and only if every set of test cases that satisfies criterion $C_1$ also satisfies $C_2$

- Must be true for every set of test cases
- **Example**: If a test set has covered every branch in a program (satisfied the branch criterion), then the test set is guaranteed to also have covered every statement
Test Coverage Criteria

- Traditional software testing is expensive and labor-intensive
- Formal coverage criteria are used to decide which test inputs to use
- More likely that the tester will find problems
- Greater assurance that the software is of high quality and reliability
- A goal or stopping rule for testing
- Criteria makes testing more efficient and effective

But how do we start to apply these ideas in practice?

Part 3: How?

Now we know what and why ...

How do we get there?
Testing Levels Based on Test Process Maturity

- **Level 0**: There's no difference between testing and debugging
- **Level 1**: The purpose of testing is to show correctness
- **Level 2**: The purpose of testing is to show that the software doesn't work
- **Level 3**: The purpose of testing is not to prove anything specific, but to reduce the risk of using the software
- **Level 4**: Testing is a mental discipline that helps all IT professionals develop higher quality software

Level 0 Thinking

- Testing is the same as debugging
- Does not distinguish between incorrect behavior and mistakes in the program
- Does not help develop software that is **reliable** or **safe**

This is what we teach undergraduate CS majors
Level 1 Thinking

- **Purpose is to show correctness**
- Correctness is **impossible** to achieve
- What do we know if no failures?
  - Good software or bad tests?
- Test engineers have no:
  - Strict goal
  - Real stopping rule
  - Formal test technique
  - Test managers are **powerless**

  This is what hardware engineers often expect

Level 2 Thinking

- **Purpose is to show failures**
- Looking for failures is a **negative** activity
- Puts testers and developers into an **adversarial** relationship
- What if there are no failures?

  This describes most software companies. How can we move to a **team approach**??
Level 3 Thinking

- Testing can only show the presence of failures
- Whenever we use software, we incur some risk
- Risk may be small and consequences unimportant
- Risk may be great and the consequences catastrophic
- Testers and developers work together to reduce risk

This describes a few “enlightened” software companies

Level 4 Thinking

A mental discipline that increases quality

- Testing is only one way to increase quality
- Test engineers can become technical leaders of the project
- Primary responsibility to measure and improve software quality
- Their expertise should help the developers

This is the way “traditional” engineering works
Summary

- **More testing saves money**
  - Planning for testing saves lots of money

- **Testing is no longer an “art form”**
  - **Engineers** have a tool box of test criteria

- **When testers become engineers, the product gets better**
  - The developers get better

Open Questions

- **Which criteria work best on embedded, highly reliable software?**
  - Which software structure to use?

- **How can we best automate this testing with robust tools?**
  - Deriving the software structure
  - Constructing the test requirements
  - Creating values from test requirements
  - Creating full test scripts
  - Solution to the “mapping problem”

- **Empirical validation**
- **Technology transition**
- **Application to new domains**