Types of Test Activities

• Testing can be broken up into **four** general types of activities
  1. Test Design
  2. Test Automation
  3. Test Execution
  4. Test Evaluation

• Each type of activity requires different **skills**, background knowledge, education and training
Test Design

Design test values to satisfy coverage criteria or other engineering goal

- This is the most technical job in software testing
- Requires knowledge of:
  - Discrete math
  - Programming
  - Testing
- Requires much of a traditional CS degree
- This is intellectually stimulating, rewarding, and challenging
- Test design is analogous to software architecture on the development side
- Using people who are not qualified to design tests is a sure way to get ineffective tests
Test Automation

Embed test values into executable scripts

- This is slightly less technical
- Requires knowledge of **programming**
  - Fairly straightforward programming – small pieces and simple algorithms
- Requires very little theory
- Very **boring** for test designers
- Programming is out of reach for many **domain experts**
- Who is responsible for determining and embedding the **expected outputs**?
  - **Test designers** may not always know the expected outputs
  - **Test evaluators** need to get involved early to help with this
Test Execution

Run tests on the software and record the results

- This is easy – and trivial if the tests are well automated
- Requires basic computer skills
  - Interns
  - Employees with no technical background
- Asking qualified test designers to execute tests is a sure way to convince them to look for a development job
- If, for example, GUI tests are not well automated, this requires a lot of manual labor
  - Test executors have to be very careful and meticulous with bookkeeping
Test Evaluation

Evaluate results of testing, report to developers

• This is much **harder** than it may seem
• Requires **knowledge** of:
  – Domain
  – Testing
• Usually requires almost **no traditional CS**
  – A background in the **domain** of the software is essential
  – An **empirical background** is very helpful (biology, psychology, …)
  – A **logic background** is very helpful (law, philosophy, math, …)
• This is **intellectually** stimulating, rewarding, and challenging
  – But not to typical CS majors – they want to solve problems and build things
### Types of Test Activities – Summary

<table>
<thead>
<tr>
<th></th>
<th>Test design</th>
<th>Design test values to satisfy coverage criteria or other engineering goal</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Requires technical knowledge of discrete math, programming and testing</td>
</tr>
<tr>
<td>2.</td>
<td>Test Automation</td>
<td>Embed test values into executable scripts</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Requires knowledge of scripting</td>
</tr>
<tr>
<td>3.</td>
<td>Test execution</td>
<td>Run tests on the software and record the results</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Requires very little knowledge</td>
</tr>
<tr>
<td>4.</td>
<td>Test evaluation</td>
<td>Evaluate results of testing, report to developers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Requires domain knowledge</td>
</tr>
</tbody>
</table>

- These four general test activities are quite different
- It is a poor use of resources to use people inappropriately

**But most test organizations use the same people for ALL FOUR activities!!**
Types of Activities in the Book

Most of this book is on test design
Other activities are well covered elsewhere
Important Terms
Validation & Verification (IEEE)

- **Validation**: The process of evaluating software at the end of software development to ensure compliance with intended usage
  - Are we building the RIGHT product?

- **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
  - Are we building the product RIGHT?

IV&V stands for “independent verification and validation”
Static and Dynamic Testing

• **Static Testing**: Testing without executing the program
  – This include software inspections and some forms of analyses
  – Very effective at finding certain kinds of problems – especially “potential” faults, that is, problems that could lead to faults when the program is modified

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

- **Software Fault**: A static defect in the software (i.e., defect, bug)

- **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
A Different View: Error, Faults, Failures
Fault & Failure Model

Three conditions necessary for a failure to be observed

1. **Execution/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

**PIE model**
Testing & Debugging

- **Testing**: Finding inputs that cause the software to fail

- **Debugging**: The process of finding a fault given a failure
Test Case

- **Test Case Values/Test Input/Test Data**: 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
  - Related Term: **Test Oracles**
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.
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**
   
   \[(\neg X \lor \neg Y) \land A \land B\]

3. **Input Domain Characterization**
   
   - \(A: \{0, 1, >1\}\)
   - \(B: \{600, 700, 800\}\)
   - \(C: \{\text{swe, cs, isa, infs}\}\)

4. **Syntactic Structures**
   
   ```
   \text{if } (x > y) \text{ then } \\
   \quad z = x - y; \\
   \text{else } \text{ then } \\
   \quad z = 2 \times x;
   ```
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 …
This graph contains:

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

**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

Driver 2 Configuration

New Configuration

Driver 2

Ignition = off

Ignition = off

(to Modified)

Ignition = off

New Configuration

Driver 1

New Configuration

Driver 2

Modified Configuration

[Ignition = off] | Button2

[Ignition = off] | Button1

[Ignition = on] | seatBack ()

[Ignition = on] | seatBottom ()

[Ignition = on] | lumbar ()

[Ignition = on] | sideMirrors ()

[Ignition = on] | Reset AND Button2

[Ignition = on] | Reset AND Button1

© Ammann & Offutt
2. Logical Expressions

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

Transitions

Program Decision Statements

Software Specifications
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>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td></td>
<td></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  \( F (\text{int } X, \text{int } Y) \)
  - Possible values  \( X: \{ <0, 0, 1, 2, >2 \}, \ Y: \{ 10, 20, 30 \} \)
  - Tests
    - \( F (-5, 10), F (0, 20), F (1, 30), F (2, 10), F (5, 20) \)

- **Combinatorial testing**
  - Pairwise testing  \( \rightarrow \) **n-way testing**
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

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

```plaintext
if (x > y)
    Δif (x >= y)
    z = x - y;
    Δz = x + y;
    Δz = x - m;
else
    Δelse
    z = 2 * x;
```
Source of Structures

- These structures can be extracted from lots of software artifacts
  - **Graphs** can be extracted from UML use cases, finite state machines, source code, ...
  - **Logical expressions** can be extracted from decisions in program source, guards on transitions, conditionals in use cases, ...

- **Model-based testing** derives tests from a model that describes some aspects of the system under test
  - The model usually describes part of the **behavior**
  - The **source** is usually *not* considered a model
Coverage Overview

Four Structures for Modeling Software

- Graphs
- Logic
- Input Space
- Syntax

Applied to

- Source
- FSMs
- Specs
- DNF

Applied to

- Source
- Specs
- Design
- Use cases

Applied to

- 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 \( C1 \) subsumes \( C2 \) if and only if every set of test cases that satisfies criterion \( C1 \) also satisfies \( C2 \)

  - 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?
A Difference View: Types of Testing

One possible classification is based on the following four classifiers:

C1: Source of test generation.

C2: Lifecycle phase in which testing takes place

C3: Goal of a specific testing activity

C4: Characteristics of the artifact under test
## C1: Source of test generation

<table>
<thead>
<tr>
<th>Artifact</th>
<th>Technique</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements (informal)</td>
<td>Black-box</td>
<td>Ad-hoc testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Boundary value analysis</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Category partition</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Classification trees</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cause-effect graphs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Equivalence partitioning</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Partition testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Predicate testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Random testing</td>
</tr>
<tr>
<td>Code</td>
<td>White-box</td>
<td>Adequacy assessment</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Coverage testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Data-flow testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Domain testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mutation testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Path testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Structural testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Test minimization using coverage</td>
</tr>
<tr>
<td>Requirements and code</td>
<td>Black-box and</td>
<td></td>
</tr>
<tr>
<td></td>
<td>White-box</td>
<td></td>
</tr>
<tr>
<td>Formal model:</td>
<td>Model-based</td>
<td>Statechart testing</td>
</tr>
<tr>
<td>Graphical or mathematical</td>
<td>Specification</td>
<td>FSM testing</td>
</tr>
<tr>
<td>specification</td>
<td></td>
<td>Pairwise testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Syntax testing</td>
</tr>
<tr>
<td>Component interface</td>
<td>Interface testing</td>
<td>Interface mutation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pairwise testing</td>
</tr>
</tbody>
</table>
### C2: Lifecycle phase in which testing takes place

<table>
<thead>
<tr>
<th>Phase</th>
<th>Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coding</td>
<td>Unit testing</td>
</tr>
<tr>
<td>Integration</td>
<td>Integration testing</td>
</tr>
<tr>
<td>System integration</td>
<td>System testing</td>
</tr>
<tr>
<td>Maintenance</td>
<td>Regression testing</td>
</tr>
<tr>
<td>Post system, pre-release</td>
<td>Beta-testing</td>
</tr>
</tbody>
</table>
## C3: Goal of specific testing activity

<table>
<thead>
<tr>
<th>Goal</th>
<th>Technique</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advertised features</td>
<td>Functional testing</td>
<td>Capture/plaback</td>
</tr>
<tr>
<td>Security</td>
<td>Security testing</td>
<td>Event sequence graphs</td>
</tr>
<tr>
<td>Invalid inputs</td>
<td>Robustness testing</td>
<td>Complete Interaction Sequence</td>
</tr>
<tr>
<td>Vulnerabilities</td>
<td>Vulnerability testing</td>
<td>Transactional-flow</td>
</tr>
<tr>
<td>Errors in GUI</td>
<td>GUI testing</td>
<td></td>
</tr>
<tr>
<td>Operational correctness</td>
<td>Operational testing</td>
<td></td>
</tr>
<tr>
<td>Reliability assessment</td>
<td>Reliability testing</td>
<td></td>
</tr>
<tr>
<td>Resistance to penetration</td>
<td>Penetration testing</td>
<td></td>
</tr>
<tr>
<td>System performance</td>
<td>Performance testing</td>
<td>Stress testing</td>
</tr>
<tr>
<td>Customer acceptability</td>
<td>Acceptance testing</td>
<td></td>
</tr>
<tr>
<td>Business compatibility</td>
<td>Compatibility testing</td>
<td>Interface testing</td>
</tr>
<tr>
<td>Peripherals compatibility</td>
<td>Configuration testing</td>
<td>Installation testing</td>
</tr>
</tbody>
</table>
## C4: Artifact under test

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application component</td>
<td>Component testing</td>
</tr>
<tr>
<td>Client and server</td>
<td>Client-server testing</td>
</tr>
<tr>
<td>Compiler</td>
<td>Compiler testing</td>
</tr>
<tr>
<td>Design</td>
<td>Design testing</td>
</tr>
<tr>
<td>Code</td>
<td>Code testing</td>
</tr>
<tr>
<td>Database system</td>
<td>Transaction-flow testing</td>
</tr>
<tr>
<td>OO software</td>
<td>OO testing</td>
</tr>
<tr>
<td>Operating system</td>
<td>Operating system testing</td>
</tr>
<tr>
<td>Real-time software</td>
<td>Real-time testing</td>
</tr>
<tr>
<td>Requirements</td>
<td>Requirement testing</td>
</tr>
<tr>
<td>Software</td>
<td>Software testing</td>
</tr>
<tr>
<td>Web service</td>
<td>Web service testing</td>
</tr>
</tbody>
</table>
Regression Testing

Jianjun Zhao
Evolving Software

- Software System evolve throughout their lifetime
- Maintenance tasks include
  - Bug Fixing
  - Addition of new features
  - Refactoring
- Maintenance (including Regression Testing) is Expensive - 40 to 70% of total cost
- Testing of evolving software – Regression testing
Regression Testing

• Definition: “The process of re-testing software that has been modified” (Textbook)

• Most software today has very little new development
  – Correcting, perfecting, adapting, or preventing problems with existing software
  – Composing new programs from existing components
  – Applying existing software to new environment

• Because of the deep interconnections among software components, changes in one method can cause problems in methods that seem to be unrelated

• Not surprisingly, most of our testing effort is regression testing

• Large regression test suites accumulate as programs (and software components) age
Regression Testing: Idea

- **Idea**
  - When you find a bug,
  - Write a test that exhibits the bug,
  - And always run that test when the code changes,
  - So that the bug doesn’t reappear

- **Without regression testing, it is surprising how often old bugs reoccur**
Regression Testing (Cont.)

● Regression testing ensures forward progress
  – We never go back to old bugs

● Regression testing can be manual or automatic
  – Ideally, run regressions after every change
  – To detect problems as quickly as possible

● But, regression testing is expensive
  – Limits how often it can be run in practice
  – Reducing cost is a long-standing research problem
Automation and Tool Support

- Too many tests to be run by hand
- Tests must be run and evaluated quickly
  - often overnight,
- Testers do not have time to view the results by inspection

Types of tools:
- Capture / Replay – Capture values entered into a GUI and replay those values on new versions
- Version control – Keeps track of collections of tests, expected results, where the tests came from, the criterion used, and their past effectiveness
- Scripting software – Manages the process of obtaining test inputs, executing the software, obtaining the outputs, comparing the results, and generating test reports
Managing Tests in Regression Test Suite

- Test suites accumulate new tests over time
- Test suites are usually run in a fixed, short, period of time
  - Often overnight, sometimes longer, sometimes shorter
- At some point, the number of tests can become unmanageable
  - We cannot finish running the tests in the time allotted
- We can always add more computer hardware
- But is it worth it?
- How many of these tests are really worth running?
Policies for Updating Test Suites

- Which tests to keep can be based on several policies
  - Add a new test for every problem report
  - Ensure that a coverage criterion is always satisfied

- Sometimes harder to choose tests to remove
  - Remove tests that do not contribute to satisfying coverage
  - Remove tests that have never found a fault (risky!)
  - Remove tests that have found the same fault as other tests (also risky!)

- Reordering strategies
  - If a suite of \( N \) tests satisfies a coverage criterion, the tests can often be reordered so that the first \( N-x \) tests satisfies the criterion – so the remaining tests can be removed
When a Regression Test Fails

- Regression tests are evaluated based on whether the result on the new program $P$ is equivalent to the result on the previous version $P-1$
  - If they differ, the test is considered to have failed

- Regression test failures represent three possibilities:
  - The software has a fault – *Must fix the faults*
  - The test values are no longer valid on the new version – *Must delete or modify the test*
  - The expected output is no longer valid – *Must update the test*

- Sometimes hard to decide which!!
Evolving Tests Over Time

• Changes to external interfaces can sometimes cause all tests to fail
  – Modern capture / replay tools will not be fooled by trivial changes like color, format, and placement
  – Automated scripts can be changed automatically via global changes in an editor or by another script

• Adding one test does not cost much – but over time the cost of these small additions start to pile up
Choosing Which Regression Tests to Run

Change Impact Analysis: How does a change impact rest of the software system?

- When a small change is made in the software, what portions of the software can be impacted by that change?

- More directly, which tests need to be re-run?
  - Conservative approach: Run all tests
  - Cheap approach: Run only tests whose test requirements relate to the statements that were changed
  - Realistic approach: Consider how the changes propagate through the software

- Clearly, tests that never reach the modified statements do not need to be run

- Lots of clever algorithms to perform CIA have been invented
  - Few if any available in commercial tools

© Ammann & Offutt 12
Rationales for Selecting a Test to Re-Run

• Inclusive: A selection technique is *inclusive if it includes tests* that are “*modification revealing*”
  – Unsafe techniques have less than 100% inclusiveness

• Precise: A selection technique is *precise if it omits regression* tests that are not modification revealing

• Efficient: A selection technique is *efficient if deciding what tests to omit is cheaper than running the omitted tests*
  – This can depend on how much automation is available
Summary of Regression Testing

- We spend far more time on regression testing than on testing new software

- If tests are based on covering criteria, all problems are much simpler
  - We know why each test was created
  - We can make rationale decisions about whether to run each test
  - We know when to delete the test
  - We know when to modify the test

- Automating regression testing will save much more than it will cost
Are Current Regression Testing Practices Sufficient?

- With a regression test suite with a 100% structural coverage, can we be able to find all regression faults?
public class E {
    void simple (int i) {
        int x = i;
        if (x > 5) {
            x = (5/(x-5));
        }
        x = x - 1;
        if (x == 0)
            print(x);
        else
            print(10/x);
    }
}...

Note: Example adapted from http://www.cc.gatech.edu/aristotle/slides/harrold.TAICPART2006.pdf
public class E {
    void simple (int i) {
        int x = i;
        if (x > 5) {// x>=5
            x = (5/(x-5));
        }
        x = x - 1;
        if (x == 0)
            print(x);
        else
            print(10/x);
    }...
}
public class E {
    void simple (int i) {
        int x = i;
        if (x > 5) {// x>=5
            x = (5/(x-5));
        }
        x = x - 1;
        if (x == 0)
            print(x);
        else
            print(10/x);
    }
}

Test Inputs: i=0

Note: Example adapted from
Example

```java
public class E {
    void simple (int i) {
        int x = i;
        if (x > 5) {//x>=5
            x = (5/(x-5));
        }
        x = x - 1;
        if (x == 0)
            print(x);
        else
            print(10/x);
    }
}
```

Test Inputs: i=0, 6

Note: Example adapted from