
Software Testing: Course Organisation

Ajitha Rajan



Course Administration: Books

- Main text: Pezzè & Young, Software Testing and Analysis: Process, Principles and Techniques, Wiley, 2007.
- Paul Ammann and Jeff Offutt, Introduction to Software Testing, Cambridge University Press, Cambridge, UK, ISBN 0-52188-038-1, 2008.
- G.J. Myers, The Art of Software Testing, Second Edition, John Wiley & Sons, New York, 1976.
- B. Marick, The Craft of Software Testing, Prentice Hall, 1995
- C Kaner, J. Bach, B. Pettichord, Lessons Learned in Software Testing, Wiley, 2001.

Material covered via readings, presentations, web resources and practical experience.

Course Administration

- **Course TA:**
Chao Peng <chao.peng@ed.ac.uk> and
Sefa Akca <S.Akca@sms.ed.ac.uk>
- Tutors:
 - Ajitha Rajan arajan@staffmail.ed.ac.uk
 - Vanya Yaneva s0835905@sms.ed.ac.uk
- **Course Web page:** Link to the course should appear on your **Learn** page.
- Useful: <http://www.cs.uoregon.edu/~michal/book/index.html>

- Useful: <http://www.testingeducation.org>
- Alternate: http://www.youtube.com/watch?v=ILkT_HV9DVU Open Lecture by James Bach on Software Testing - where he takes a different perspective on the task.
- Useful Context: <http://www.computer.org/web/swebok> see the Testing section of the Software Engineering Body of Knowledge

Course Assessment

- **One practical worth 25% of the final mark** — Practical will involve working in groups of 2.

Issued: 25th February. **Deadline** - 29th March.

- **One examination worth 75%**. This will be an **open-book** examination.
- **Tutorials** — are not assessed but doing them will make it much easier to do the examination and practicals

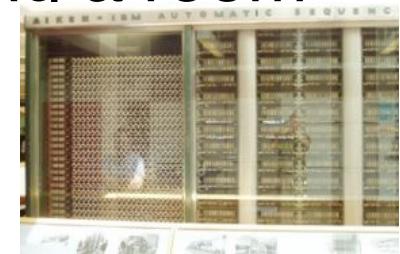
Tutorials

- There are four tutorials available on the course.
 - Tutorial 1 - Week 3 (week starting January 28th)
 - Tutorial 2 - Week 5 (week starting Feb 11th)
 - Tutorial 3 - Week 7 (week starting March 4th)
 - Tutorial 4 - Week 8 (week starting March 11th)
- You must read through and prepare for the tutorial in advance. Some of the tasks in the tutorials require you to work with another person, these can be done during the tutorial.

Software Testing – Why ?

What is a computer bug?

- In 1947 Harvard University was operating a room-sized computer called the Mark II.
 - mechanical relays
 - glowing vacuum tubes
 - technicians program the computer by reconfiguring it
 - Technicians had to change the occasional vacuum tube.
- A moth flew into the computer and was zapped by the high voltage when it landed on a relay.
- Hence, the first computer bug!
 - I am not making this up :-)



Bugs a.k.a. ...

- Defect
- Fault
- Problem
- Error
- Incident
- Anomaly
- Variance
- Failure
- Inconsistency
- Product Anomaly
- Product Incidence
- Feature :-)

Bug Free Software

- Software is in the news for the wrong reason
 - Security breach, Mars Lander lost, hackers getting credit card information, etc.
- Why can't software engineers develop software that just works?
 - As software gets more features and supports more platforms it becomes increasingly difficult to make it create bug-free.

Discussion ...

- Do you think bug free software is unattainable?
 - Are there technical barriers that make this impossible?
 - Is it just a question of time before we can do this?
 - Are we missing technology or processes?

Sources of Problems

- **Requirements Definition:** Erroneous, incomplete, inconsistent requirements.
- **Design:** Fundamental design flaws in the software.
- **Implementation:** Mistakes in chip fabrication, wiring, programming faults, malicious code.
- **Support Systems:** Poor programming languages, faulty compilers and debuggers, misleading development tools.

Sources of Problems (Cont'd)

- **Inadequate Testing of Software:**
Incomplete testing, poor verification, mistakes in debugging.
- **Evolution:** Sloppy redevelopment or maintenance, introduction of new flaws in attempts to fix old flaws, incremental escalation to inordinate complexity.

Adverse Effects of Faulty Software

- **Communications:** Loss or corruption of communication media, non delivery of data.
- **Space Applications:** Lost lives, launch delays.
- **Defense and Warfare:** Misidentification of friend or foe.

Adverse Effects of Faulty Software (Cont'd)

- **Transportation**: Deaths, delays, sudden acceleration, inability to brake.
- **Safety-critical Applications**: Death, injuries.
- **Electric Power**: Death, injuries, power outages, long-term health hazards (radiation).

Adverse Effects of Faulty Software (Cont'd)

- **Money Management:** Fraud, violation of privacy, shutdown of stock exchanges and banks, negative interest rates.
- **Control of Elections:** Wrong results (intentional or non-intentional).
- **Control of Jails:** Technology-aided escape attempts and successes, accidental release of inmates, failures in software controlled locks.
- **Law Enforcement:** False arrests and imprisonments.

Bug in Space Code

- Project Mercury's FORTRAN code had the following fault:
 DO I=1.10 instead of ... DO I=1,10
- The fault was discovered in an analysis of why the software did not seem to generate results that were sufficiently accurate.
- The erroneous 1.10 would cause the loop to be executed exactly once!

Military Aviation Problems

- An F-18 crashed because of a missing exception condition:
if ... then ... without the else clause that was thought could not possibly arise.
- In simulation, an F-16 program bug caused the virtual plane to flip over whenever it crossed the equator, as a result of a missing minus sign to indicate south latitude.

Year Ambiguities

- In 1992, Mary Bandar received an invitation to attend a kindergarten in Winona, Minnesota, along with others born in '88.
- Mary was 104 years old at the time.

Dates, Times, and Integers

- The number $32,768 = 2^{15}$ has caused all sorts of grief from the overflowing of 16-bit words.
- A Washington D.C. hospital computer system collapsed on September 19, 1989, 2^{15} days after January 1, 1900, forcing a lengthy period of manual operation.

Shaky Math

- In the US, five nuclear power plants were shut down in 1979 because of a program fault in a simulation program used to design nuclear reactor to withstand earthquakes.
- This program fault was, unfortunately, discovered after the power plants were built!

Shaky Math (Cont'd)

- Apparently, the arithmetic sum of a set of numbers was taken, instead of the sum of the absolute values.
- The five reactors would probably not have survived an earthquake that was as strong as the strongest earthquake ever recorded in the area.

Therac-25 Radiation “Therapy”

- In Texas, 1986, a man received between 16,500-25,000 rads in less than 1 sec, over an area of about 1 cm.
- He lost his left arm, and died of complications 5 months later.
- In Texas, 1986, a man received at least 4,000 rads in the right temporal lobe of his brain.
- The patient eventually died as a result of the overdose.

AT&T Bug: Hello? ... Hello?

- In mid-December 1989, AT&T installed new software in 114 electronic switching systems.
- On January 15, 1990, 5 million calls were blocked during a 9 hour period nationwide.

AT&T Bug (Cont'd)

- The bug was traced to a C program that contained a break statement within an switch clause nested within a loop.
- The switch clause was part of a loop. Initially, the loop contained only if clauses with break statements to exit the loop.
- When the control logic became complicated, a switch clause was added to improve the readability of the code ...

Bank Generosity

- A Norwegian bank ATM consistently dispersed 10 times the amount required.
- Many people joyously joined the queues as the word spread.

Bank Generosity (Cont'd)

- A software flaw caused a UK bank to duplicate every transfer payment request for half an hour. The bank lost 2 billion British pounds!
- The bank eventually recovered the funds but lost half a million pounds in potential interest.

Bug in BoNY Software

- The Bank of New York (BoNY) had a \$32 billion overdraft as the result of a 16-bit integer counter that went unchecked.
- BoNY was unable to process the incoming credits from security transfers, while the NY Federal Reserve automatically debited BoNY's cash account.

Discussion ...

- Have you heard of other software bugs?
 - In the media?
 - From personal experience?
- Does this embarrass you as a future software engineer?

Specification

“if you can’t say it, you can’t do it”

- You have to know what your product is before you can say if it has a bug.
- A *specification* defines the product being created and includes:
 - Functional requirements that describes the features the product will support. E.g., on a word processor
 - Save, print, check spelling, change font, ...
 - Non-functional requirements are constraints on the product. E.g,
 - Security, reliability, user friendliness, platform, ...

A software bug occurs when at least one of these rules is true

- The software does not do something that the specification says it should do.
- The software does something that the specification says it should not do.
- The software does something that the specification does not mention.
- The software does not do something that the product specification does not mention but should.
- The software is difficult to understand, hard to use, slow ...

Most bugs are not because of mistakes in the code ...

- Specification ($\sim= 55\%$)
- Design ($\sim= 25\%$)
- Code ($\sim= 15\%$)
- Other ($\sim= 5\%$)

Relative cost of bugs

“bugs found later cost more to fix”

- Cost to fix a bug increases exponentially (10^x)
 - i.e., it increases tenfold as time increases
- E.g., a bug found during specification costs \$1 to fix.
 - ... if found in design cost is \$10
 - ... if found in code cost is \$100
 - ... if found in released software cost is \$1000

Goal of a software tester

- ... to *find* bugs
- ... as *early* in the software development processes as possible
- ... and make sure they get *fixed*.

- **Advice:** Be careful not to get caught in the dangerous spiral of unattainable perfection.

You now know ...

- ... what is a bug
- ... the relationship between specification and bugs
- ... the cost of a bug relative to when it is found
- ... the unattainable goal of perfect software
- ... the goal of the software tester
- ... valuable attributes of a software tester

Software Testing Overview

Four Questions

- ▶ Does my software work?

Four Questions

- ▶ Does my software work?
- ▶ Does my software meet its specification?

Four Questions

- ▶ Does my software work?
- ▶ Does my software meet its specification?
- ▶ I've changed something, does it still work?

Four Questions

- ▶ Does my software work?
- ▶ Does my software meet its specification?
- ▶ I've changed something, does it still work?
- ▶ How can I become a better programmer?

The Answer

Testing

Can we test?

"Program testing can be used to show the presence of bugs, but never to show their absence!" Edsger Dijkstra.

This is true, but it is no reason to give up on testing. All software has bugs. Anything you do to reduce the number of bugs is a good thing.

Test Driven Development

Later on we will look at *test driven development* (TDD) which is a programming discipline where you write the tests before you write the code.

What is a Test?

- ▶ A test is simply some inputs and some expected outputs.

This simple description hides a lot of complexity, though.

- ▶ How do I run a test? How do I give inputs to a real-time system or a GUI?
- ▶ How do I know what my code is supposed to do, so that I can work out what the expected outputs are?

Aspects of Testing

1. Test Design
2. Test Automation
3. Test Execution
4. Test Evaluation

It is very important that test execution should be as automated as possible. It should be part of your Makefile. Some systems even automatically run tests when you check in code.

Test Design

- ▶ Writing good tests is hard.
- ▶ It requires knowledge of your problem, and
- ▶ Knowledge of common errors.
- ▶ Often, a test designer is a separate position in a company.

Test Design

- ▶ Adversarial view of test design:
How do I break software?

Test Design

- ▶ Adversarial view of test design:
How do I break software?
- ▶ Constructive view of test design:
How do I design software tests that improve the software process?

Test Automation

- ▶ Designing good tests is hard.
- ▶ If you don't make the execution of the tests an automated process, then people will never run them.
- ▶ There are many automated systems, but you can roll your own via scripting languages.
- ▶ The xUnit framework has support in most languages for the automated running of tests.
- ▶ It should be as simple as `make tests`.

Test Automation

- ▶ There are tools for automatically testing web systems.
- ▶ There are tools for testing GUIs.
 - ▶ If you design your software correctly you should decouple as much of the GUI behaviour from the rest of the program as you can. This will not only make your program easier to port to other GUIs, but also it will make it easier to test.
- ▶ Don't forget to include test automation in your compilation process.
- ▶ Consider integrating automated testing into your version management system.

Test Execution

You need to think of test execution as separate activity. You have to remember to run the tests. In a large organization this might require some planning.

- ▶ Easy if testing is automated.
- ▶ Hard for some domains e.g. GUI.
- ▶ Very hard in distributed or real time environments.

Test Evaluation

- ▶ My software does not pass some of the tests. Is this good or bad?
- ▶ My software passes all my tests. Can I go home now? Or do I have to design more tests?

Important Terminology and Concepts

- ▶ **Validation:** The process of evaluation 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

Important Terminology and Concepts

- ▶ **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 behaviour.

Understanding the difference will help you fix faults. Write your code so it is testable.

Fault/Error/Failure Example

```
int count_spaces(char* str) {  
    int length, i, count;  
    count = 0;  
    length = strlen(str);  
    for(i=1; i<length; i++) {  
        if(str[i] == ' ') { count++; }  
    }  
    return(count);  
}
```

- ▶ Software Fault: `i=1` should be `i=0`.
- ▶ Software Error: some point in the program where you incorrectly count the number of spaces.
- ▶ Failure inputs and outputs that make the fault happen. For example `count_spaces("H H H")`; would not cause the failure while `count_spaces(" H")`; does.

Fault/Error/Failure

- ▶ Fault/Error/Failure is an important tool for thinking about how to test something (not just software).
- ▶ I am trying to correct **faults** that cause **errors** that cause **failures**.
- ▶ How do I design test cases that give **failures** that are caused by **errors** that are due to **faults** in the code.

Unit Testing

- ▶ xUnit testing is a framework where individual functions and methods are tested.
- ▶ It is not particularly well suited to integration testing or regression testing.
- ▶ The best way to write testable code is to write the tests as you develop the code.
- ▶ Writing the test cases after the code takes more time and effort than writing the test during the code. It is like good documentation; you'll always find something else to do if you leave until after you've written the code.

This will be covered in more detail in the next lecture.

Tools for Unit Test — JUnit

Ajitha Rajan



JUnit

JUnit is a framework for writing tests

- Written by Erich Gamma (Design Patterns) and Kent Beck (eXtreme Programming)
- JUnit uses *Java's reflection capabilities* (Java programs can examine their own code) and (as of version 4) *annotations*
- JUnit allows us to:
 - define and execute tests and test suites
 - Use test as an effective means of specification
 - write code and use the tests to support refactoring
 - integrate revised code into a build
- JUnit is available on several IDEs, e.g. BlueJ, JBuilder, and Eclipse have JUnit integration to some extent.

JUnit's Terminology

- A **test runner** is software that runs tests and reports results.

Many implementations: standalone GUI, command line, integrated into IDE

- A **test suite** is a collection of test cases.
- A **test case** tests the response of a single method to a particular set of inputs.
- A **unit test** is a test of the smallest element of code you can sensibly test, usually a single class.

JUnit's Terminology

- A **test fixture** is the environment in which a test is run. A new fixture is set up before each test case is executed, and torn down afterwards.

Example: if you are testing a database client, the fixture might place the database server in a standard initial state, ready for the client to connect.

- An **integration test** is a test of how well classes work together.

JUnit provides some limited support for integration tests.

- *Proper* unit testing would involve **mock objects** – fake versions of the other classes with which the class under test interacts.

JUnit does not help with this. It is worth knowing about, but not always necessary.

Structure of a JUnit (4) test class

We want to test a class named Triangle

- This is the unit test for the Triangle class; it defines objects used by one or more tests.

```
public class TriangleTest{  
}
```

- This is the default constructor.

```
public TriangleTest(){ }
```

Structure of a JUnit (4) test class

- `@Before public void init()`

Creates a test fixture by creating and initialising objects and values.
- `@After public void cleanUp()`

Releases any system resources used by the test fixture. Java usually does this for free, but files, network connections etc. might not get tidied up automatically.
- `@Test public void noBadTriangles(), @Test public void scaleneOk(), etc.`

These methods contain tests for the `Triangle` constructor and its `isScalene()` method.

Making Tests: Assert

- Within a test,
 - Call the method being tested and get the actual result.
 - *assert* a property that should hold of the test result.
 - Each *assert* is a challenge on the test result.
- If the property fails to hold then assert fails, and throws an `AssertionFailedError`:
 - JUnit catches these Errors, records the results of the test and displays them.

Making Tests: Assert

- static void assertTrue(boolean *test*)

static void assertTrue(String *message*, boolean *test*)

Throws an `AssertionFailedError` if the test fails. The optional *message* is included in the Error.

- static void assertFalse(boolean *test*)

static void assertFalse(String *message*, boolean *test*)

Throws an `AssertionFailedError` if the test succeeds.

Aside: Throwable

- `java.lang.Error`: a problem that an application would not normally try to handle — does not need to be declared in *throws* clause.
e.g. command line application given bad parameters by user.
- `java.lang.Exception`: a problem that the application might reasonably cope with — needs to be declared in *throws* clause.
e.g. network connection timed out during connect attempt.
- `java.lang.RuntimeException`: application might cope with it, but rarely — does not need to be declared in *throws* clause.
e.g. I/O buffer overflow.

Triangle class

For the sake of example, we will create and test a trivial Triangle class:

- The constructor creates a Triangle object, where only the lengths of the sides are recorded and the private variable p is the longest side.
- The `isScalene` method returns true if the triangle is scalene.
- The `isEquilateral` method returns true if the triangle is equilateral.
- We can write the test methods before the code. This has advantages in separating coding from testing.

But Eclipse helps more if you create the class under test first: Creates test stubs (methods with empty bodies) for all methods and constructors.

Notes on creating tests

- **Size:** Often the amount of (very routine) test code will exceed the size of the code for small systems.
- **Complexity:** Testing complex code can be a complex business and the tests can get quite complex.
- **Effort:** The effort taken in creating test code is repaid in reduced development time, most particularly when we go on to use the test subject in anger (i.e. real code).
- **Behaviour:** Creating a test often helps clarify our ideas on how a method should behave (particularly in exceptional circumstances).

A JUnit 4 test for Triangle

```
package st;

more imports are necessary R import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

no need to inherit from TestCase R public class TestTriangle {

    private Triangle t;

    Use annotations... R     @Before public void setUp() throws Exception {
        t = new Triangle(3, 4, 5);
    }

    ...rather than special names R     @Test public void scaleneOk() {
        assertTrue(t.isScalene());
    }
}
```

The Triangle class itself

- Is JUnit too much for small programs?
- Not if you think it will reduce errors.
- Tests on this scale of program often turn up errors or omissions – construct the tests working from the specification
- Sometimes you can omit tests for some particularly straightforward parts of the system

Assert methods II

- `assertEquals(expected, actual)`

`assertEquals(String message, expected, actual)`

This method is heavily overloaded: *expected* and *actual* must be both objects or both of the same primitive type. For objects, uses your `equals` method, if you have defined it properly, as `public boolean equals(Object o)` — otherwise it uses `==`

- `assertSame(Object expected, Object actual)`

`assertSame(String message, Object expected, Object actual)`

Asserts that two objects refer to the same object (using `==`)

- `assertNotSame(Object expected, Object actual)`

`assertNotSame(String message, Object expected, Object actual)`

Asserts that two objects do not refer to the same object

Assert methods III

- `assertNull(Object object)`
`assertNull(String message, Object object)`
Asserts that the object is null
- `assertNotNull(Object object)`
`assertNotNull(String message, Object object)`
Asserts that the object is not null
- `fail()`
`fail(String message)`
Causes the test to fail and throw an `AssertionFailedError` — Useful as a result of a complex test, when the other assert methods are not quite what you want

The assert statement in Java

- Earlier versions of JUnit had an `assert` method instead of an `assertTrue` method — The name had to be changed when Java 1.4 introduced the `assert` statement
- There are two forms of the `assert` statement:
 - `assert boolean_condition ;`
 - `assert boolean_condition : error_message ;`

Both forms throw an `AssertionFailedError` if the `boolean_condition` is false. The second form, with an explicit `error_message`, is seldom necessary.

The assert statement in Java

When to use an assert statement:

- Use it to document a condition that you ‘*know*’ to be true
- Use assert false; in code that you ‘*know*’ cannot be reached (such as a default case in a switch statement)
- Do not use assert to check whether parameters have legal values, or other places where throwing an Exception is more appropriate
- **Can be dangerous:** customers are not impressed by a library bombing out with an assertion failure.

JUnit in Eclipse

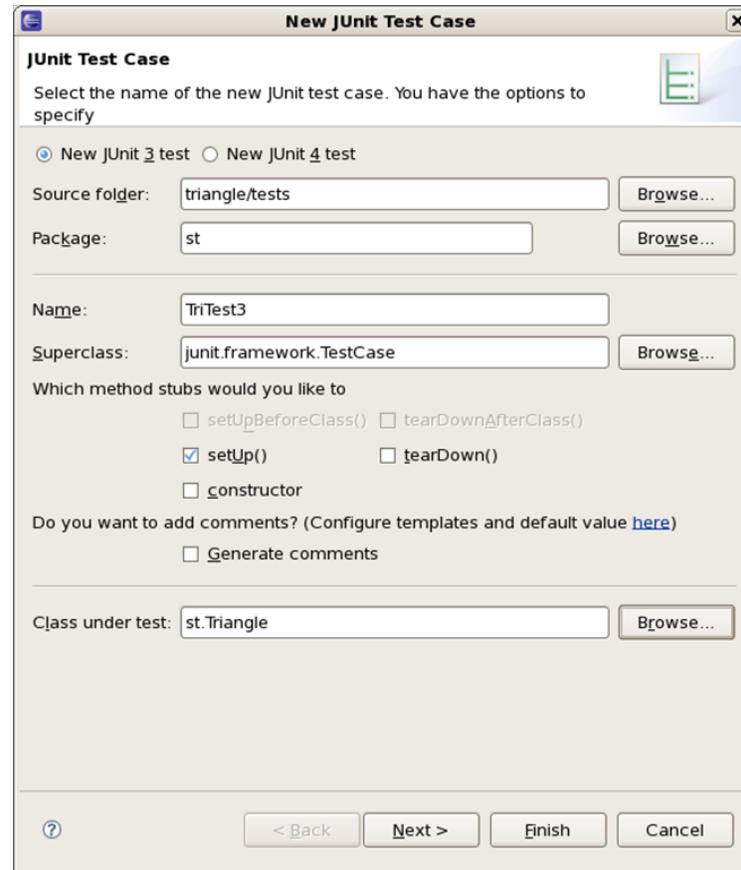
To create a test class, select
File → New → JUnit Test Case
and enter the name of your test case

Package R

Test class R

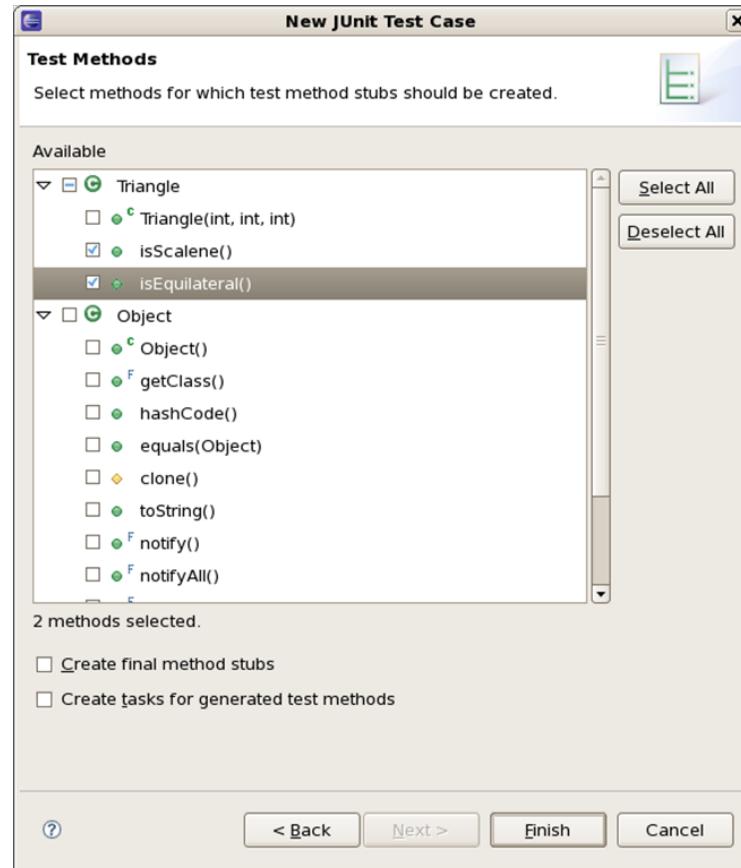
Decide what stubs you want to create R

Identify the class under test R

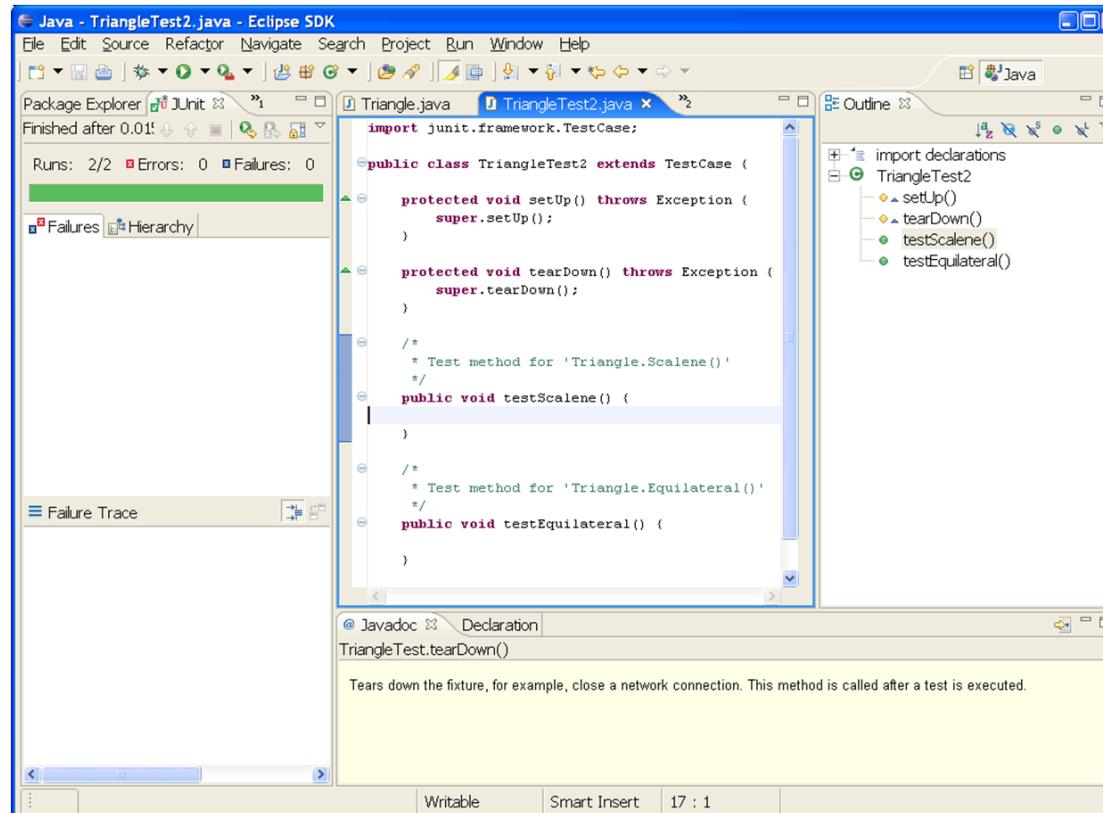


Creating a Test

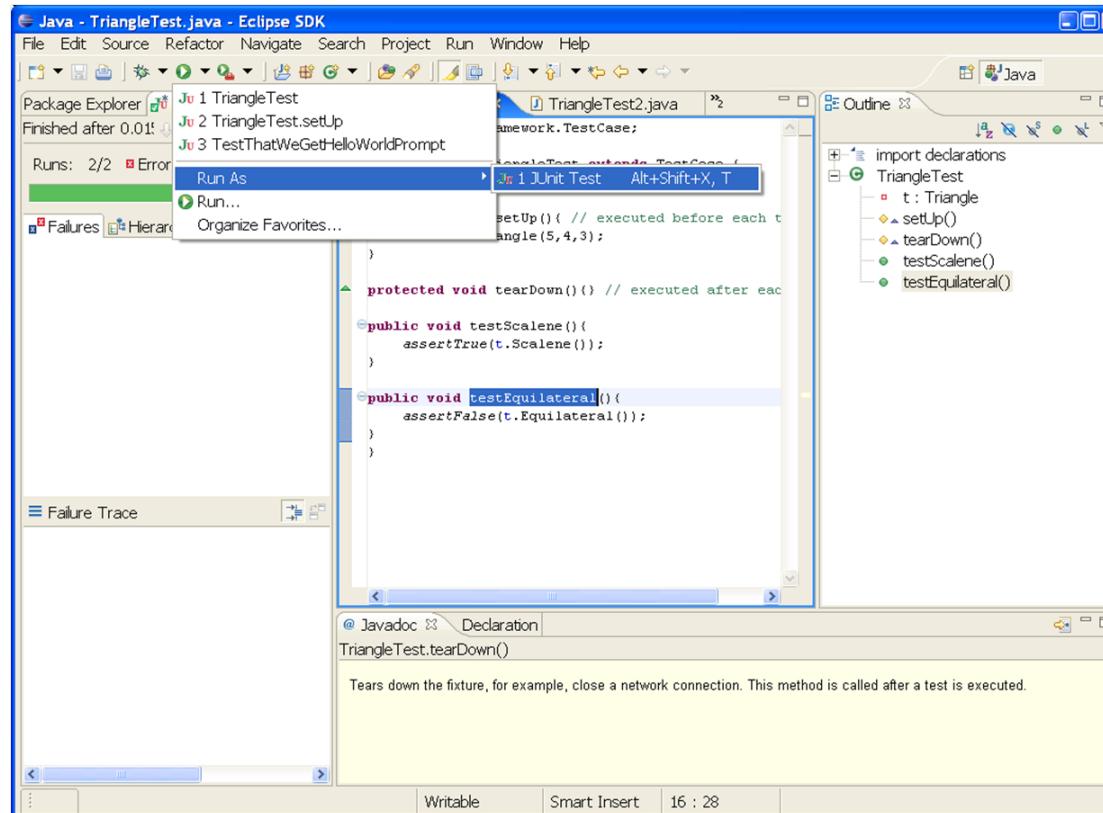
Decide what you want to test R



Template for New Test

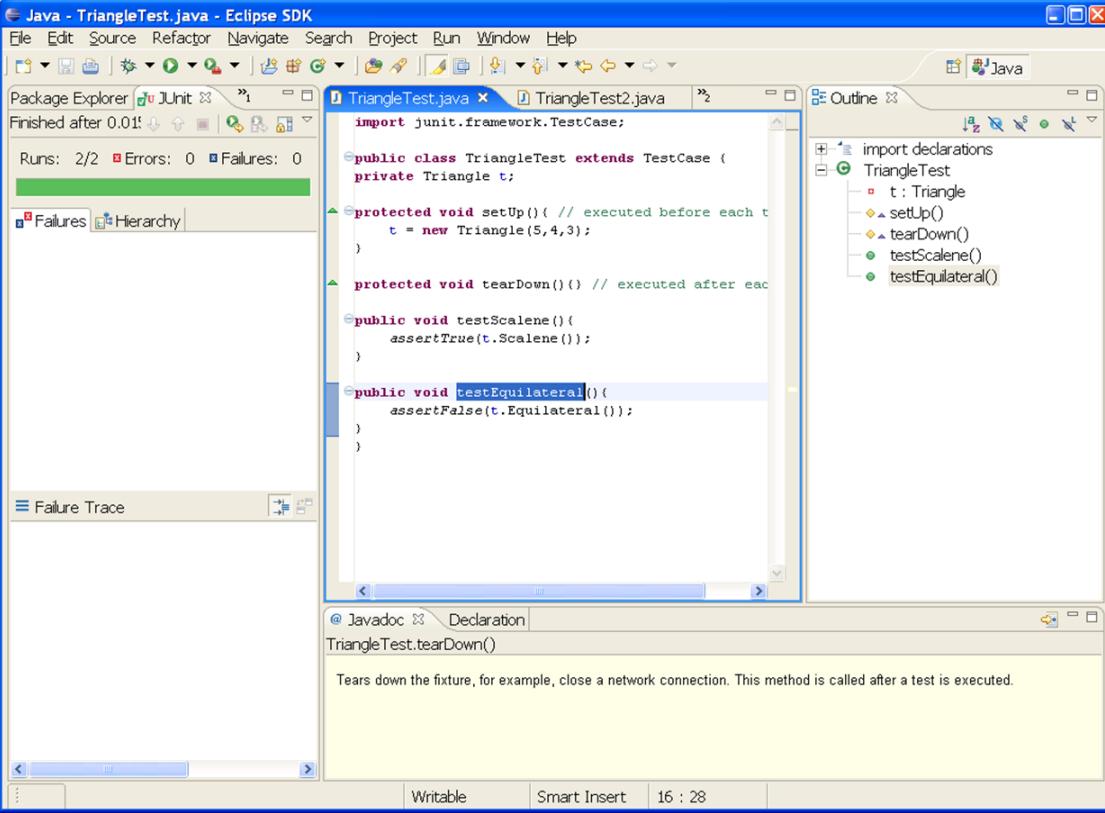


Running JUnit



Results

Results are here R



The screenshot shows the Eclipse IDE interface with the following details:

- Title Bar:** Java - TriangleTest.java - Eclipse SDK
- Package Explorer:** Shows a green bar indicating "Runs: 2/2 Errors: 0 Failures: 0".
- Central View:** Displays the code for `TriangleTest.java`. The `tearDown()` method is currently selected.
- Outline View:** On the right, it shows the class structure: `import declarations`, `TriangleTest` (containing `t : Triangle`, `setUp()`, `tearDown()`, `testScalene()`, and `testEquilateral()`).
- Bottom View:** Shows the `Javadoc` and `Declaration` tabs for the `tearDown()` method, with a description: "Tears down the fixture, for example, close a network connection. This method is called after a test is executed."

Issues with JUnit

JUnit has a model of calling methods and checking results against the expected result. **Issues** are:

- **State:** objects that have significant internal state (e.g. collections with some additional structure) are harder to test because it may take many method calls to get an object into a state you want to test. **Solutions:**
 - Write long tests that call some methods many times.
 - Add additional methods in the interface to allow observation of state (or make private variables public?)
 - Add additional methods in the interface that allow the internal state to be set to a particular value
 - “Heisenbugs” can be an issue in these cases (changing the observations changes what is observed).

Issues with JUnit

- Other effects, e.g. output can be hard to capture correctly.
- JUnit tests of GUIs are not particularly helpful (recording gestures might be helpful here?)

Positives

- Using JUnit encourages a ‘*testable*’ style, where the result of a calling a method is easy to check against the specification:
 - Controlled use of state
 - Additional observers of the state (testing interface)
 - Additional components in results that ease checking
- It is well integrated into a range of IDEs (e.g. Eclipse)
- Tests are easy to define and apply in these environments.
- JUnit encourages frequent testing during development — e.g. XP (eXtreme Programming) ‘*test as specification*’
- JUnit tends to shape code to be easily testable.
- JUnit supports a range of extensions that support structured testing (e.g. coverage analysis) – we will see some of these extensions later.

Another Framework for Testing

- Framework for Integrated Test (FIT), by Ward Cunningham (inventor of wiki)
- Allows closed loop between customers and developers:
 - Takes HTML tables of expected behaviour from customers or spec.
 - Turns those tables into test data: inputs, activities and assertions regarding expected results.
 - Runs the tests and produces tabular summaries of the test runs.
- Only a few years old, but lots of people seem to like it — various practitioners seem to think it is revolutionary.

Readings

Required Readings

- [JUnit Test Infected: Programmers Love Writing Tests](#)
an introduction to JUnit.
- [Using JUnit With Eclipse IDE](#)
an O'Reilly article
- [Unit Testing in Jazz Using JUnit](#)
an NCSU Open Lab article on using JUnit with Eclipse

Suggested Readings

- Michael Olan. 2003. [Unit testing: test early, test often](#). J. Comput. Small Coll. 19, 2 (December 2003), 319-328.

Resources

Getting started with Eclipse and JUnit

Activity: to start using JUnit within Eclipse review and try the example of defining tests for a Triangle class.

[\[link to Activity\]](#)

Video: this video tutorial shows how to create a new Eclipse project and start writing JUnit tests first.

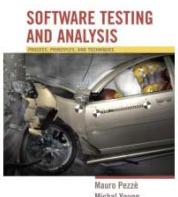
[\[link to Video\]](#)

Get testing!

Start up Eclipse and:

1. Create a new Java project
2. Add a new package, ‘‘st’’
3. Create st.Triangle; grab the source from the Junit lecture’s Activity in the resources
4. Create a new source folder called ‘‘tests’’ if you like (with a new ‘‘st’’ package)
5. Create a new JUnit test for st.Triangle
6. And get testing!

Functional testing

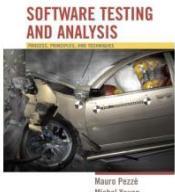


(c) 2007 Mauro Pezzè & Michal Young

Ch 10, slide 1

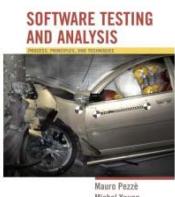
Learning objectives

- Understand the rationale for systematic (non-random) selection of test cases
 - Understand the basic concept of partition testing and its underlying assumptions
- Understand why functional test selection is a primary, base-line technique
 - Why we expect a specification-based partition to help select valuable test cases
- Distinguish functional testing from other systematic testing techniques



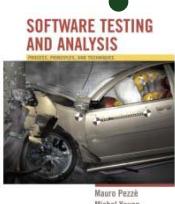
Functional testing

- 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

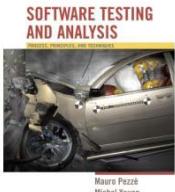


Why Not Random?

- Non-uniform distribution of faults
- *Example:* Java class “roots” applies quadratic equation
$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

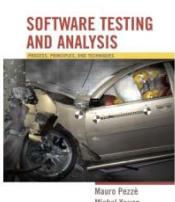
Incomplete implementation logic: Program does not properly handle the case in which $b^2 - 4ac = 0$ and $a=0$

Failing values are *sparse* in the input space – needles in a very big haystack. Random sampling is unlikely to choose $a=0.0$ and $b=0.0$

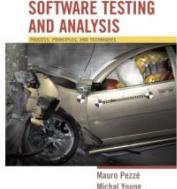
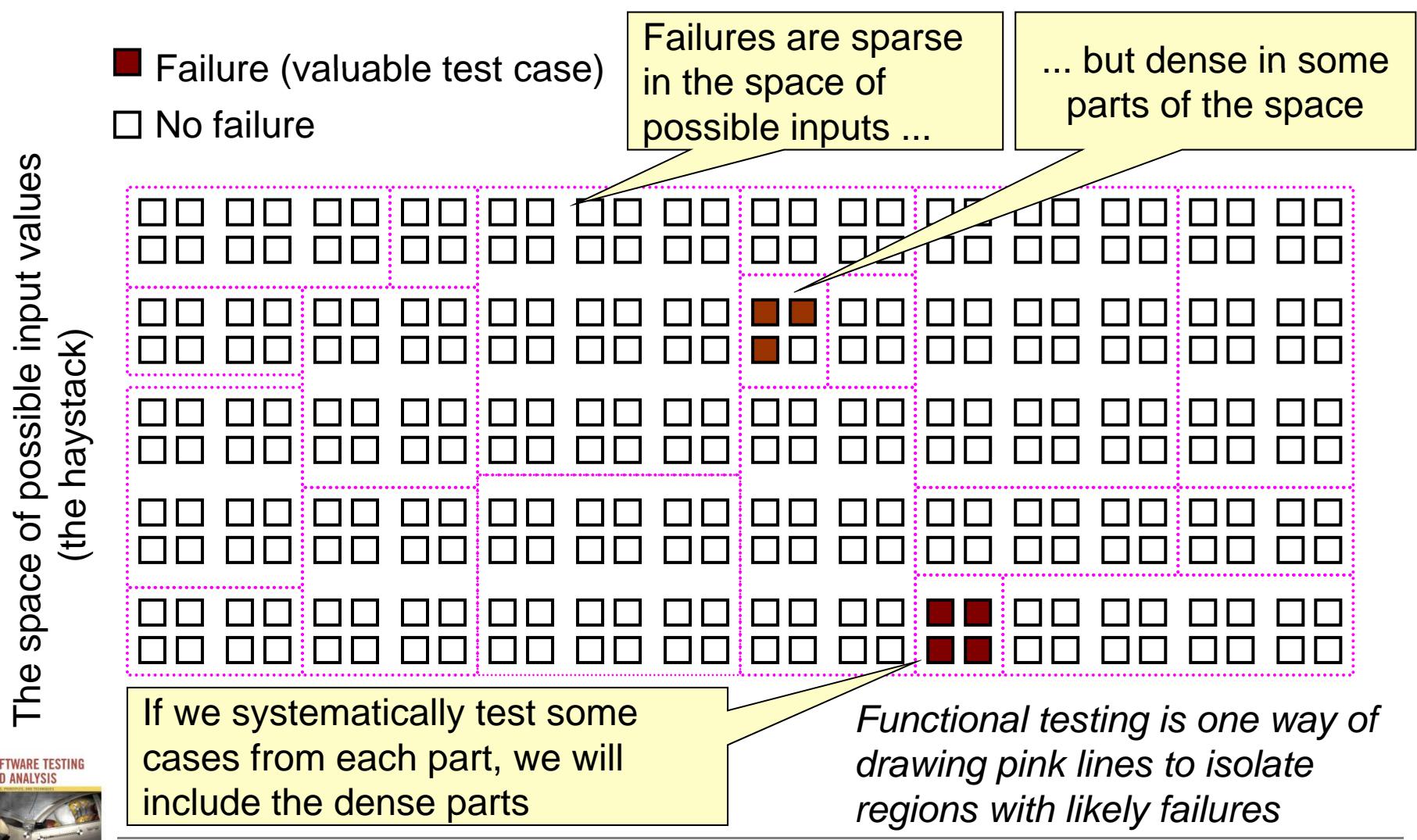


Consider the purpose of testing ...

- To estimate the proportion of needles to hay, sample randomly
 - Reliability estimation requires unbiased samples for valid statistics. *But that's not our goal!*
- To find needles and remove them from hay, look systematically (non-uniformly) for needles
 - Unless there are a *lot* of needles in the haystack, a random sample will not be effective at finding them
 - We need to use everything we know about needles, e.g., are they heavier than hay? Do they sift to the bottom?

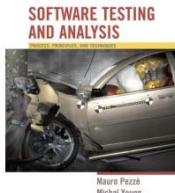


Systematic Partition Testing



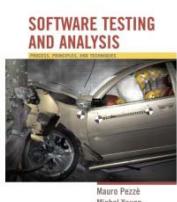
The partition principle

- Exploit some knowledge to choose samples that are more likely to include “special” or trouble-prone regions of the input space
 - Failures are sparse in the whole input space ...
 - ... but we may find regions in which they are dense
- (Quasi*)-Partition testing: separates the input space into classes whose union is the entire space
 - » *Quasi because: The classes may overlap
- Desirable case: Each fault leads to failures that are dense (easy to find) in some class of inputs
 - sampling each class in the quasi-partition selects at least one input that leads to a failure, revealing the fault
 - seldom guaranteed; we depend on experience-based heuristics



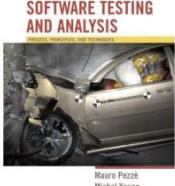
Functional testing: exploiting the specification

- 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)



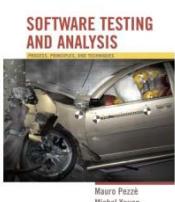
Why functional testing?

- The base-line technique for designing test cases
 - Timely
 - Often useful in refining specifications and assessing testability *before* code is written
 - Effective
 - finds some classes of fault (e.g., missing logic) that can elude other approaches
 - Widely applicable
 - to any description of program behavior serving as spec
 - at any level of granularity from module to system testing.
 - Economical
 - typically less expensive to design and execute than structural (code-based) test cases



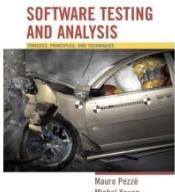
Early functional test design

- Program code is not necessary
 - Only a description of intended behavior is needed
 - Even incomplete and informal specifications can be used
 - Although precise, complete specifications lead to better test suites
- Early functional test design has side benefits
 - Often reveals ambiguities and inconsistency in spec
 - Useful for assessing testability
 - And improving test schedule and budget by improving spec
 - Useful explanation of specification
 - or in the extreme case (as in XP), test cases are the spec



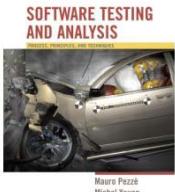
Functional versus Structural: Classes of faults

- Different testing strategies (functional, structural, fault-based, model-based) are most effective for different classes of faults
- Functional testing is best for *missing logic* faults
 - A common problem: Some program logic was simply forgotten
 - Structural (code-based) testing will never focus on code that isn't there!



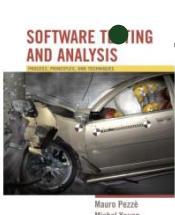
Functional vs structural test: granularity levels

- Functional test applies at all granularity levels:
 - Unit (from module interface spec)
 - Integration (from API or subsystem spec)
 - System (from system requirements spec)
 - Regression (from system requirements + bug history)
- Structural (code-based) test design applies to relatively small parts of a system:
 - Unit
 - Integration

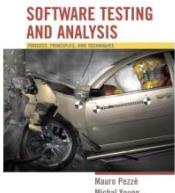
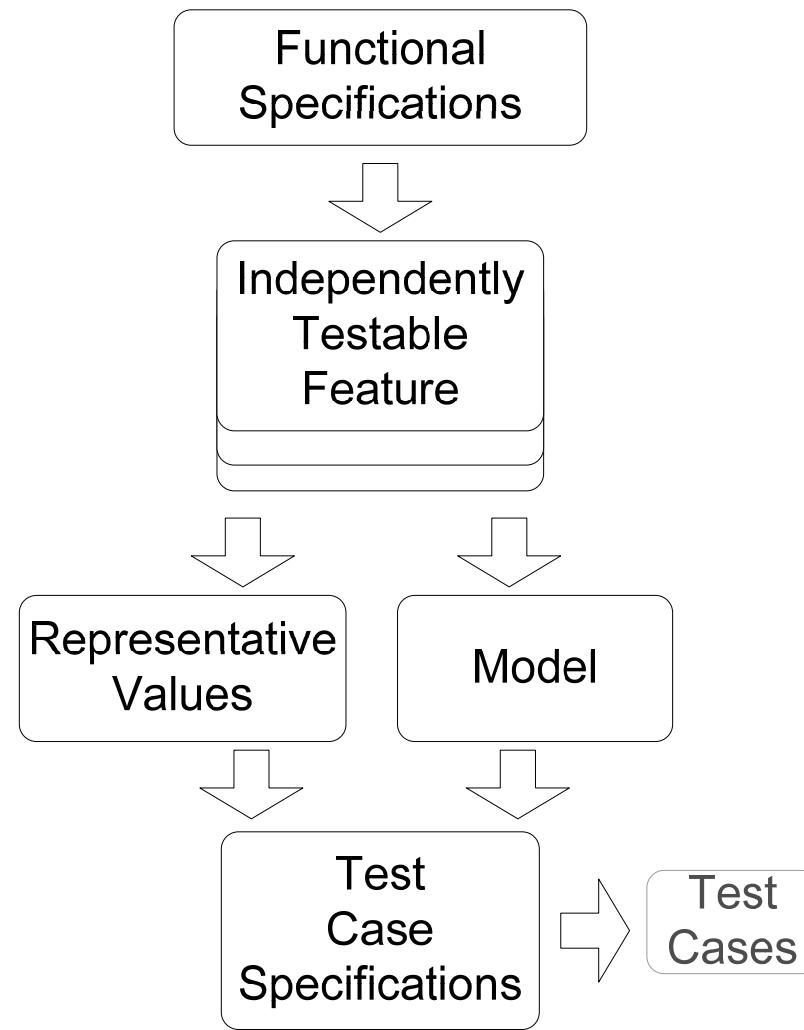


Steps: From specification to test cases

- 1. Decompose the specification
 - If the specification is large, break it into *independently testable features* to be considered in testing
- 2. Select representatives
 - Representative values of each input, or
 - Representative behaviors of a *model*
 - Often simple input/output transformations don't describe a system. We use models in program specification, in program design, and in test design
- 3. Form test specifications
 - Typically: combinations of input values, or model behaviors
- 4. Produce and execute actual tests



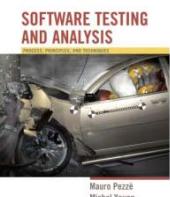
From specification to test cases



Simple example: Postal code lookup



- Input: ZIP code (5-digit US Postal code)
- Output: List of cities
- What are some representative values (or classes of value) to test?



Example: Representative values

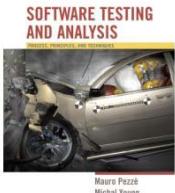
Simple example with
one input, one output



The screenshot shows the ZIP Code Lookup page from the United States Postal Service. At the top is the USPS logo. Below it is a cartoon character of a mail carrier holding a envelope. To the right of the character is the text "ZIP Code Lookup". Below the character are three buttons: "Search By Address >>", "Search By City >>", and "Search By Company >>". To the right of these buttons is a "Find" button. Underneath these buttons is the text "Find a list of cities that are in a ZIP Code." Below this is a field labeled "Required Fields" with a red asterisk next to it, followed by "ZIP Code" and an empty input field. At the bottom is a "Submit >" button.

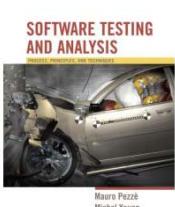
- Correct zip code
 - With 0, 1, or many cities
- Malformed zip code
 - Empty; 1-4 characters; 6 characters; very long
 - Non-digit characters
 - Non-character data

Note prevalence of boundary
values (0 cities, 6 characters)
and error cases

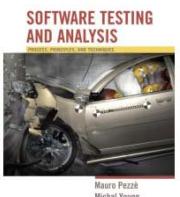


Summary

- Functional testing, i.e., generation of test cases from specifications is a valuable and flexible approach to software testing
 - Applicable from very early system specs right through module specifications
- (quasi-)Partition testing suggests dividing the input space into (quasi-)equivalent classes
 - Systematic testing is intentionally non-uniform to address special cases, error conditions, and other small places
 - Dividing a big haystack into small, hopefully uniform piles where the needles might be concentrated



Combinatorial testing

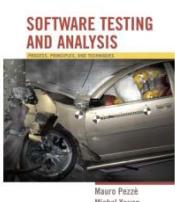


(c) 2007 Mauro Pezzè & Michal Young

Ch 11, slide 1

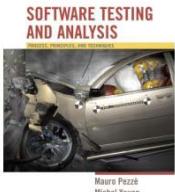
Learning objectives

- Understand rationale and basic approach for systematic combinatorial testing
- Learn how to apply some representative combinatorial approaches
 - Category-partition testing
 - Pairwise combination testing
 - Catalog-based testing
- Understand key differences and similarities among the approaches□□□
 - and application domains for which they are suited



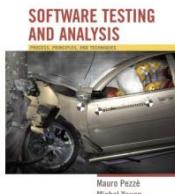
Combinatorial testing: Basic idea

- 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

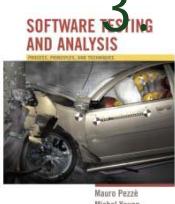
- Category-partition testing
 - separate (manual) identification of values that characterize the input space from (automatic) generation of combinations for test cases
- Pairwise testing
 - systematically test interactions among attributes of the program input space with a relatively small number of 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

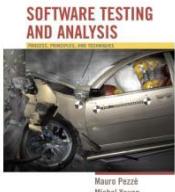
3 Introduce constraints



An informal specification: *check configuration*

Check Configuration

- Check the validity of a computer configuration
- The parameters of check-configuration are:
 - Model
 - Set of components



An informal specification: parameter *model*

Model

- A model identifies a specific product and determines a set of constraints on available components. Models are characterized by logical slots for components, which may or may not be implemented by physical slots on a bus. Slots may be required or optional. Required slots must be assigned with a suitable component to obtain a legal configuration, while optional slots may be left empty or filled depending on the customers' needs

Example:

The required “slots” of the Chipmunk C20 laptop computer include a screen, a processor, a hard disk, memory, and an operating system. (Of these, only the hard disk and memory are implemented using actual hardware slots on a bus.) The optional slots include external storage devices such as a CD/DVD writer.



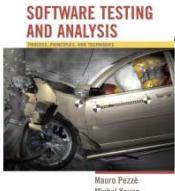
An informal specification of parameter *set of components*

Set of Components

- A set of *(slot, component)* pairs, corresponding to the required and optional slots of the model. A *component* is a choice that can be varied within a model, and which is not designed to be replaced by the end user. Available components and a default for each slot is determined by the model. The special value *empty* is allowed (and may be the default selection) for optional slots. In addition to being compatible or incompatible with a particular model and slot, individual components may be compatible or incompatible with each other.

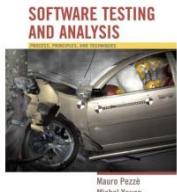
Example:

The default configuration of the Chipmunk C20 includes 20 gigabytes of hard disk; 30 and 40 gigabyte disks are also available. (Since the hard disk is a required slot, empty is not an allowed choice.) The default operating system is RodentOS 3.2, personal edition, but RodentOS 3.2 mobile server edition may also be selected. The mobile server edition requires at least 30 gigabytes of hard disk.



Step1: Identify independently testable units and categories

- Choosing categories
 - no hard-and-fast rules for choosing categories
 - not a trivial task!
- Categories reflect test designer's judgment
 - regarding which classes of values may be treated differently by an implementation
- Choosing categories well requires experience and knowledge
 - of the application domain and product architecture. The test designer must look under the surface of the specification and identify hidden characteristics



Step 1: Identify independently testable units

Parameter *Model*

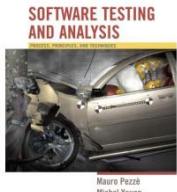
- Model number
- Number of required slots for selected model (#SMRS)
- Number of optional slots for selected model (#SMOS)

Parameter *Components*

- Correspondence of selection with model slots
- Number of required components with selection \neq empty
- Required component selection
- Number of optional components with selection \neq empty
- Optional component selection

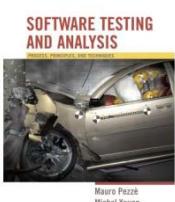
Environment element: *Product database*

- Number of models in database (#DBM)
- Number of components in database (#DBC)



Step 2: Identify relevant values

- Identify (list) representative classes of values for each of the categories
 - Ignore interactions among values for different categories (considered in the next step)
- Representative values may be identified by applying
 - Boundary value testing
 - select extreme values within a class
 - select values outside but as close as possible to the class
 - select interior (non-extreme) values of the class
 - Erroneous condition testing
 - select values outside the normal domain of the program



Step 2: Identify relevant values: Model

Model number

Malformed

Not in database

Valid

Number of required slots for selected model (#SMRS)

0

1

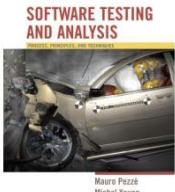
Many

Number of optional slots for selected model (#SMOS)

0

1

Many



Step 2: Identify relevant values: Component

Correspondence of selection with model slots

Omitted slots

Extra slots

Mismatched slots

Complete correspondence

Number of required components with non empty selection

0

< number required slots

= number required slots

Required component selection

Some defaults

All valid

≥ 1 incompatible with slots

≥ 1 incompatible with another selection

≥ 1 incompatible with model

≥ 1 not in database

Number of optional components with non empty selection

0

< #SMOS

= #SMOS

Optional component selection

Some defaults

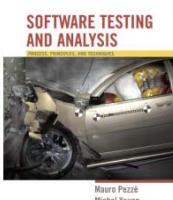
All valid

≥ 1 incompatible with slots

≥ 1 incompatible with another selection

≥ 1 incompatible with model

≥ 1 not in database



Step 2: Identify relevant values: Database

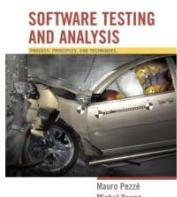
Number of models in database (#DBM)

- 0
- 1
- Many

Number of components in database (#DBC)

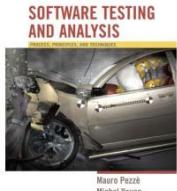
- 0
- 1
- Many

Note 0 and 1 are unusual (special) values. They might cause unanticipated behavior alone or in combination with particular values of other parameters.



Step 3: Introduce constraints

- A combination of values for each category corresponds to a test case specification
 - in the example we have 314.928 test cases
 - most of which are impossible!
 - example
zero slots and *at least one incompatible slot*
- Introduce constraints to
 - rule out impossible combinations
 - reduce the size of the test suite if too large



Step 3: error constraint

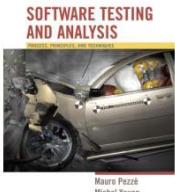
[error] indicates a value class that

- corresponds to a erroneous values
- need be tried only once

Example

Model number: Malformed and Not in database
error value classes

- No need to test all possible combinations of errors
- One test is enough (we assume that handling an error case bypasses other program logic)



Example - Step 3: *error* constraint

Model number

Malformed	[error]
Not in database	[error]
Valid	

Correspondence of selection with model slots

Omitted slots	[error]
Extra slots	[error]
Mismatched slots	[error]
Complete correspondence	

Number of required comp. with non empty selection

0	[error]
< number of required slots	[error]

Required comp. selection

≥ 1 not in database	[error]
--------------------------	---------

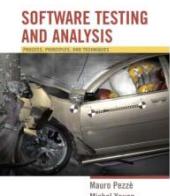
Number of models in database (#DBM)

0	[error]
---	---------

Number of components in database (#DBC)

0	[error]
---	---------

Error constraints
reduce test suite
from 314.928 to
2.711 test cases



Step 3: *property* constraints

constraint [property] [if-property] rule out invalid combinations of values

[property] groups values of a single parameter to identify subsets of values with common properties

[if-property] bounds the choices of values for a category that can be combined with a particular value selected for a different category

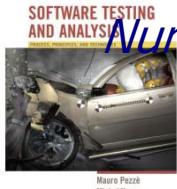
Example

combine

*Number of required comp. with non empty selection = number required slots
[if RSMANY]*

only with

Number of required slots for selected model (#SMRS) = Many [Many]



Example - Step 3: property constraints

Number of required slots for selected model (#SMRS)

1	[property RSNE]
Many	[property RSNE] [property RSMANY]

Number of optional slots for selected model (#SMOS)

1	[property OSNE]
Many	[property OSNE] [property OSMANY]

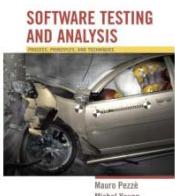
Number of required comp. with non empty selection

0	[if RSNE] [error]
< number required slots	[if RSNE] [error]
= number required slots	[if RSMANY]

Number of optional comp. with non empty selection

< number required slots	[if OSNE]
= number required slots	[if OSMANY]

from 2.711 to 908
test cases



Step 3 (cont): *single* constraints

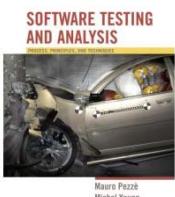
[single] indicates a value class that test designers choose to test only once to reduce the number of test cases

Example

value some default for required component selection and optional component selection may be tested only once despite not being an erroneous condition

note -

single and error have the same effect but differ in rationale. Keeping them distinct is important for documentation and regression testing



Example - Step 3: *single* constraints

Number of required slots for selected model (#SMRS)

- 0 [single]
- 1 [property RSNE] [single]

Number of optional slots for selected model (#SMOS)

- 0 [single]
- 1 [single] [property OSNE]

Required component selection

- Some default [single]

Optional component selection

- Some default [single]

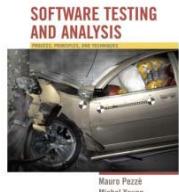
Number of models in database (#DBM)

- 1 [single]

Number of components in database (#DBC)

- 1 [single]

from 908 to
69 test cases



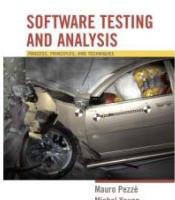
Check configuration – Summary

Parameter Model

- Model number
 - Malformed [error]
 - Not in database [error]
 - Valid
- Number of required slots for selected model (#SMRS)
 - 0 [single]
 - 1 [property RSNE] [single]
 - Many [property RSNE] [property RSMANY]
- Number of optional slots for selected model (#SMOS)
 - 0 [single]
 - 1 [property OSNE] [single]
 - Many [property OSNE] [property OSMANY]

Environment Product data base

- Number of models in database (#DBM)
 - 0 [error]
 - 1 [single]
 - Many
- Number of components in database (#DBC)
 - 0 [error]
 - 1 [single]
 - Many

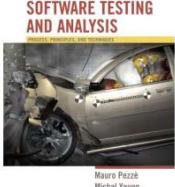


Parameter Component

- Correspondence of selection with model slots
 - Omitted slots [error]
 - Extra slots [error]
 - Mismatched slots [error]
 - Complete correspondence
- # of required components (selection ≠ empty)
 - 0 [if RSNE] [error]
 - < number required slots [if RSNE] [error]
 - = number required slots [if RSMANY]
- Required component selection
 - Some defaults [single]
 - All valid
 - ≥ 1 incompatible with slots
 - ≥ 1 incompatible with another selection
 - ≥ 1 incompatible with model
 - ≥ 1 not in database [error]
- # of optional components (selection ≠ empty)
 - 0 [if OSNE]
 - < #SMOS [if OSMANY]
 - = #SMOS
- Optional component selection
 - Some defaults [single]
 - All valid
 - ≥ 1 incompatible with slots
 - ≥ 1 incompatible with another selection
 - ≥ 1 incompatible with model
 - ≥ 1 not in database [error]

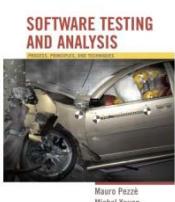
Next ...

- Category partition testing gave us
 - Systematic approach: Identify characteristics and values (the creative step), generate combinations (the mechanical step)
- But ...
 - Test suite size grows very rapidly with number of categories. Can we use a non-exhaustive approach?
- Pairwise (and n-way) combinatorial testing do
 - Combine values systematically but not exhaustively
 - Rationale: Most unplanned interactions are among just two or a few parameters or parameter characteristics



Pairwise combinatorial testing

- Category partition works well when intuitive constraints reduce the number of combinations to a small amount of test cases
 - Without many constraints, the number of combinations may be unmanageable □□□
- Pairwise combination (instead of exhaustive)
 - Generate combinations that efficiently cover all pairs (triples,...) of classes
 - Rationale: most failures are triggered by single values or combinations of a few values. Covering pairs (triples,...) reduces the number of test cases, but reveals most faults



Example: Display Control

No constraints reduce the total number of combinations
□□□432 (3x4x3x4x3) test cases
if we consider all combinations

Display Mode	Language	Fonts	Color	Screen size
full-graphics	English	Minimal	Monochrome	Hand-held
text-only	French	Standard	Color-map	Laptop
limited-bandwidth	Spanish	Document-loaded	16-bit	Full-size
	Portuguese		True-color	

Pairwise combinations: 17 test cases

Language	Color	Display Mode	Fonts	Screen Size
English	Monochrome	Full-graphics	Minimal	Hand-held
English	Color-map	Text-only	Standard	Full-size
English	16-bit	Limited-bandwidth	-	Full-size
English	True-color	Text-only	Document-loaded	Laptop
French	Monochrome	Limited-bandwidth	Standard	Laptop
French	Color-map	Full-graphics	Document-loaded	Full-size
French	16-bit	Text-only	Minimal	-
French	True-color	-	-	Hand-held
Spanish	Monochrome	-	Document-loaded	Full-size
Spanish	Color-map	Limited-bandwidth	Minimal	Hand-held
Spanish	16-bit	Full-graphics	Standard	Laptop
Spanish	True-color	Text-only	-	Hand-held
Portuguese	-	-	Monochrome	Text-only
Portuguese	Color-map	-	Minimal	Laptop
Portuguese	16-bit	Limited-bandwidth	Document-loaded	Hand-held
Portuguese	True-color	Full-graphics	Minimal	Full-size
Portuguese	True-color	Limited-bandwidth	Standard	Hand-held

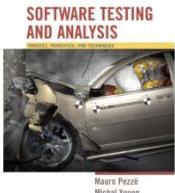


Adding constraints

- Simple constraints

example: color monochrome not compatible with screen laptop and full size

can be handled by considering the case in separate tables



Example: Monochrome only with hand-held

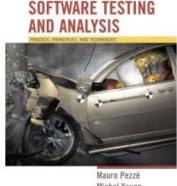
Display Mode	Language	Fonts	Color	Screen size
full-graphics	English	Minimal	Monochrome	Hand-held
text-only	French	Standard	Color-map	
limited-bandwidth	Spanish	Document-loaded	16-bit	
	Portuguese		True-color	

Display Mode	Language	Fonts	Color	Screen size
full-graphics	English	Minimal		
text-only	French	Standard	Color-map	Laptop
limited-bandwidth	Spanish	Document-loaded	16-bit	Full-size
	Portuguese		True-color	



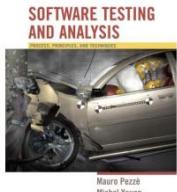
Next ...

- 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:

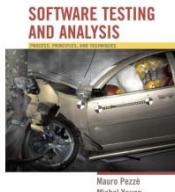
- Pre-conditions
- Post-conditions
- Definitions
- Variables
- Operations

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

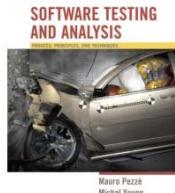


An informal specification: cgi_decode

Function *cgi_decode* translates a cgi-encoded string to a plain ASCII string, reversing the encoding applied by the common gateway interface (CGI) of most web servers

CGI translates spaces to +, and translates most other non-alphanumeric characters to hexadecimal escape sequences

cgi_decode maps + to spaces, %xy (where x and y are hexadecimal digits) to the corresponding ASCII character, and other alphanumeric characters to themselves



An informal specification: input/output

[INPUT] encoded: string of characters (the input CGI sequence)

can contain:

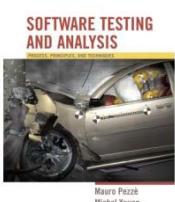
- alphanumeric characters
 - the character +
 - the substring %xy, where x and y are hexadecimal digits
- is terminated by a null character

[OUTPUT] decoded: string of characters (the plain ASCII characters corresponding to the input CGI sequence)

- alphanumeric characters copied into output (in corresponding positions)
- blank for each + character in the input
- single ASCII character with value xy for each substring %xy

[OUTPUT] return value cgi_decode returns

- 0 for success
- 1 if the input is malformed



Step 1: Identify simple elements

Pre-conditions: conditions on inputs that must be true before the execution

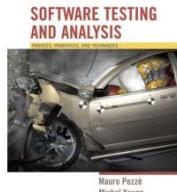
- validated preconditions: checked by the system
- assumed preconditions: assumed by the system

Post-conditions: results of the execution

Variables: elements used for the computation

Operations: main operations on variables and inputs

Definitions: abbreviations



Step 1: cgi_decode (pre and post)

PRE 1 (Assumed) input string encoded null-terminated string of chars

PRE 2 (Validated) input string encoded sequence of CGI items

POST 1 if encoded contains alphanumeric characters, they are copied to the output string

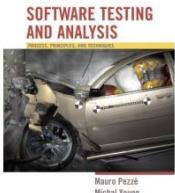
POST 2 if encoded contains characters +, they are replaced in the output string by ASCII SPACE characters

POST 3 if encoded contains CGI hexadecimals, they are replaced by the corresponding ASCII characters

POST 4 if encoded is processed correctly, it returns 0

POST 5 if encoded contains a wrong CGI hexadecimal (a substring xy, where either x or y are absent or are not hexadecimal digits, cgi_decode returns 1

POST 6 if encoded contains any illegal character, it returns 1



Step 1: cgi_decode (var, def, op.)

VAR 1 encoded: a string of ASCII characters

VAR 2 decoded: a string of ASCII characters

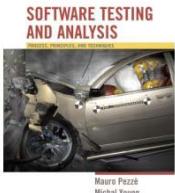
VAR 3 return value: a boolean

DEF 1 hexadecimal characters, in range ['0' .. '9', 'A' .. 'F', 'a' .. 'f']

DEF 2 sequences %xy, where x and y are hexadecimal characters

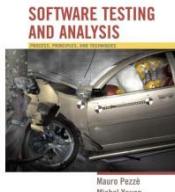
DEF 3 CGI items as alphanumeric character, or '+', or CGI hexadecimal

OP 1 Scan encoded



Step 2: Derive initial set of test case specs

- Validated preconditions:
 - simple precondition (expression without operators)
 - 2 classes of inputs:
 - inputs that satisfy the precondition
 - inputs that do not satisfy the precondition
 - compound precondition (with AND or OR):
 - apply modified condition/decision (MC/DC) criterion
 - Assumed precondition:
 - apply MC/DC only to “OR preconditions”
 - Postconditions and Definitions :
 - if given as conditional expressions, consider conditions as if they were validated preconditions



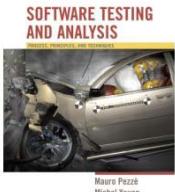
Step 2: cgi_decode (tests from Pre)

PRE 2 (Validated) the input string encoded is a sequence of CGI items

- *TC-PRE2-1: encoded is a sequence of CGI items*
- *TC-PRE2-2: encoded is not a sequence of CGI items*

POST 1 if encoded contains alphanumeric characters, they are copied in the output string in the corresponding position

- *TC-POST1-1: encoded contains alphanumeric characters*
- *TC-POST1-2: encoded does not contain alphanumeric characters*
- POST 2 if encoded contains characters +, they are replaced in the output string by ASCII SPACE characters
 - *TC-POST2-1: encoded contains character +*
 - *TC-POST2-2: encoded does not contain character +*



Step 2: cgi_decode (tests from Post)

POST 3 if `encoded` contains CGI hexadecimals, they are replaced by the corresponding ASCII characters

- *TC-POST3-1 Encoded: contains CGI hexadecimals*
- *TC-POST3-2 Encoded: does not contain a CGI hexadecimal*

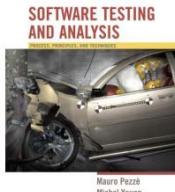
POST 4 if `encoded` is processed correctly, it returns 0

POST 5 if `encoded` contains a wrong CGI hexadecimal (a substring `xy`, where either x or y are absent or are not hexadecimal digits, `cgi_decode` returns 1

- *TC-POST5-1 Encoded: contains erroneous CGI hexadecimals*

POST 6 if `encoded` contains any illegal character, it returns 1

- *TC-POST6-1 Encoded: contains illegal characters*



Step 2: cgi_decode (tests from Var)

VAR 1 encoded: a string of ASCII characters

VAR 2 decoded: a string of ASCII characters

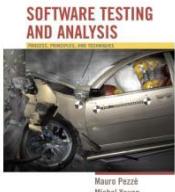
VAR 3 return value: a boolean

DEF 1 hexadecimal characters, in range ['0' .. '9', 'A' .. 'F', 'a' .. 'f']

DEF 2 sequences %xy, where x and y are hexadecimal characters

DEF 3 CGI items as alphanumeric character, or '+', or CGI hexadecimal

OP 1 Scan encoded



Step 3: Apply the catalog

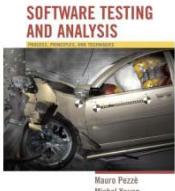
- Scan the catalog sequentially
- For each element of the catalog
 - scan the specifications
 - apply the catalog entry
- Delete redundant test cases
- Catalog:
 - List of kinds of elements that can occur in a specification
 - Each catalog entry is associated with a list of generic test case specifications

Example:

catalog entry Boolean

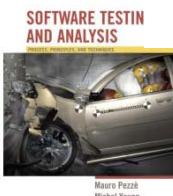
two test case specifications: true, false

Label in/out indicate if applicable only to input, output, both



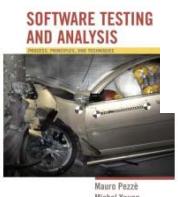
A simple catalog (part I)

- Boolean
 - True in/out
 - False in/out
- Enumeration
 - Each enumerated value in/out
 - Some value outside the enumerated set in
- Range L ... U
 - L-1 in
 - L in/out
 - A value between L and U in/out
 - U in/out
 - U+1 in
- Numeric Constant C
 - C in/out
 - C -1 in
 - C+1 in
 - Any other constant compatible with C in



A simple catalog (part II)

- Non-Numeric Constant C
 - C in/out
 - Any other constant compatible with C in
 - Some other compatible value in
- Sequence
 - Empty in/out
 - A single element in/out
 - More than one element in/out
 - Maximum length (if bounded) or very long in/out
 - Longer than maximum length (if bounded) in
 - Incorrectly terminated in
- Scan with action on elements P
 - P occurs at beginning of sequence in
 - P occurs in interior of sequence in
 - P occurs at end of sequence in
 - PP occurs contiguously in
 - P does not occur in sequence in
 - pP where p is a proper prefix of P in
 - Proper prefix p occurs at end of sequence in



Example - Step 3: Catalog entry *boolean*

- Boolean
 - True in/out
 - False in/out

applies to *return value*
generates 2 test cases already covered by
TC-PRE2-1 and TC-PRE2-2

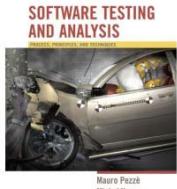
Example - Step 3: entry *enumeration*

- Enumeration
 - Each enumerated value in/out
 - Some value outside the enumerated set in

applies to

- ### *- CGI item (DEF 3)*

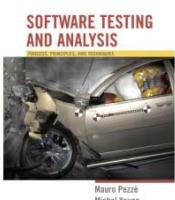
included in TC-POST1-1, TC-POST1-2, TC-POST2-1, TC-POST2-2, TC-POST3-1, TC-POST3-2



Example - Step 3: entry enumeration

applies also to improper CGI hexadecimals

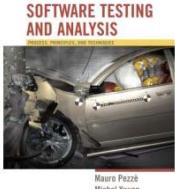
- New test case specifications
 - TC-POST5-2 *encoded terminated with %x, where x is a hexadecimal digit*
 - TC-POST5-3 *encoded contains %ky, where k is not a hexadecimal digit and y is a hexadecimal digit*
 - TC-POST5-4 *encoded contains %xk, where x is a hexadecimal digit and k is not*
- Old test case specifications can be eliminated if they are less specific than the newly generated cases
 - TC-POST3-1 *encoded contains CGI hexadecimals*
 - TC-POST5-1 *encoded contains erroneous CGI hexadecimals*



Example - Step 3: entry *range*

Applies to variables defined on a finite range

- hexadecimal digit
 - characters / and :
(before 0 and after 9 in the ASCII table)
 - values 0 and 9 (bounds),
 - one value between 0 and 9
 - @, G, A, F, one value between A and F
 - }, g, a, f, one value between a and f
 - 30 new test cases (15 for each character)
- Alphanumeric char (DEF 3):
 - 5 new test cases

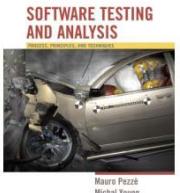


Example - Step 3: entries *numeric* and *non-numeric constant*

Numeric Constant does not apply

Non-Numeric Constant applies to

- + and %, in DEF 3 and DEF 2:
- 6 new Test Cases (all redundant)

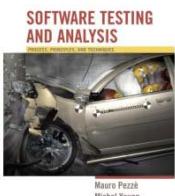


Step 3: entry sequence

apply to

encoded (VAR 1), *decoded* (VAR 2), and *cgi-item* (DEF 2)

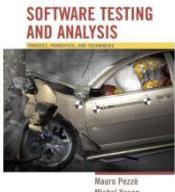
- 6 new Test Cases for each variable
- Only 6 are non-redundant:
 - *encoded*
 - empty sequence
 - sequence of length one
 - long sequence
 - *cgi-item*
 - % terminated sequence (subsequence with one char)
 - % initiated sequence
 - sequence including %xyz, with x, y, and z hexadecimals



Step 3: entry *scan*

applies to *Scan encoded* (OP 1) and generates 17 test cases:

- only 10 are non-redundant



summary of generated test cases (i/ii)

TC-POST2-1: *encoded* contains `+`

TC-POST2-2: *encoded* does not contain `+`

TC-POST3-2: *encoded* does not contain a CGI-hexadecimal

TC-POST5-2: *encoded* terminated with `%x`

TC-VAR1-1: *encoded* is the empty sequence

TC-VAR1-2: *encoded* a sequence containing a single character

TC-VAR1-3: *encoded* is a very long sequence

TC-DEF2-1: *encoded* contains `%/y`

TC-DEF2-2: *encoded* contains `%0y`

TC-DEF2-3: *encoded* contains `'%xy'` (*x* in [1..8])

TC-DEF2-4: *encoded* contains `'%9y'`

TC-DEF2-5: *encoded* contains `'%:y'`

TC-DEF2-6: *encoded* contains `'%@y'`

TC-DEF2-7: *encoded* contains `'%Ay'`

TC-DEF2-8: *encoded* contains `'%xy'` (*x* in [B..E])

TC-DEF2-9: *encoded* contains `'%Fy'`

TC-DEF2-10: *encoded* contains `'%Gy'`

TC-DEF2-11: *encoded* contains `%`y'`

TC-DEF2-12: *encoded* contains `%ay`

TC-DEF2-13: *encoded* contains `%xy` (*x* in [b..e])

TC-DEF2-14: *encoded* contains `%fy'`

TC-DEF2-15: *encoded* contains `%gy`

TC-DEF2-16: *encoded* contains `%x/`

TC-DEF2-17: *encoded* contains `%x0`

TC-DEF2-18: *encoded* contains `%xy` (*y* in [1..8])

TC-DEF2-19: *encoded* contains `%x9`

TC-DEF2-20: *encoded* contains `%x:`

TC-DEF2-21: *encoded* contains `%x@`

TC-DEF2-22: *encoded* contains `%xA`

TC-DEF2-23: *encoded* contains `%xy` (*y* in [B..E])

TC-DEF2-24: *encoded* contains `%xF`

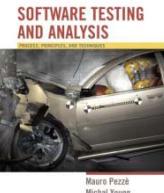
TC-DEF2-25: *encoded* contains `%xG`

TC-DEF2-26: *encoded* contains `%x``

TC-DEF2-27: *encoded* contains `%xa`

TC-DEF2-28: *encoded* contains `%xy` (*y* in [b..e])

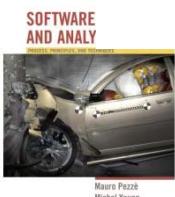
TC-DEF2-29: *encoded* contains `%xf`



Summary of generated test cases (ii/ii)

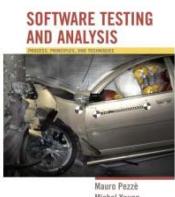
TC-DEF2-30: *encoded* contains %*xg*
TC-DEF2-31: *encoded* terminates with %
TC-DEF2-32: *encoded* contains %*xyz*
TC-DEF3-1: *encoded* contains /
TC-DEF3-2: *encoded* contains 0
TC-DEF3-3: *encoded* contains *c* in [1..8]
TC-DEF3-4: *encoded* contains 9
TC-DEF3-5: *encoded* contains :
TC-DEF3-6: *encoded* contains @
TC-DEF3-7: *encoded* contains A
TC-DEF3-8: *encoded* contains *c* in [B..Y]
TC-DEF3-9: *encoded* contains Z
TC-DEF3-10: *encoded* contains [
TC-DEF3-11: *encoded* contains `
TC-DEF3-12: *encoded* contains a
TC-DEF3-13: *encoded* contains *c* in [b..y]
TC-DEF3-14: *encoded* contains z
TC-DEF3-15: *encoded* contains {

TC-OP1-1: *encoded* starts with an alphanumeric character
TC-OP1-2: *encoded* starts with +
TC-OP1-3: *encoded* starts with %*xy*
TC-OP1-4: *encoded* terminates with an alphanumeric character
TC-OP1-5: *encoded* terminates with +
TC-OP1-6: *encoded* terminated with %*xy*
TC-OP1-7: *encoded* contains two consecutive alphanumeric characters
TC-OP1-8: *encoded* contains ++
TC-OP1-9: *encoded* contains %*xy*%*zw*
TC-OP1-10: *encoded* contains %*x*%*yz*



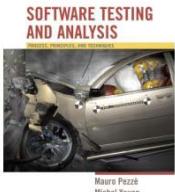
What have we got?

- From category partition testing:
 - Division into a (manual) step of identifying categories and values, with constraints, and an (automated) step of generating combinations
- From catalog based testing:
 - Improving the manual step by recording and using standard patterns for identifying significant values
- From pairwise testing:
 - Systematic generation of smaller test suites
- These ideas can be combined



Summary

- Requirements specifications typically begin in the form of natural language statements
 - but flexibility and expressiveness of natural language is an obstacle to automatic analysis
- Combinatorial approaches to functional testing consist of
 - A manual step of structuring specifications into set of properties
 - An automatizable step of producing combinations of choices
- *Brute force* synthesis of test cases is tedious and error prone
- Combinatorial approaches decompose *brute force*' work into steps to attack the problem incrementally by separating analysis and synthesis activities that can be quantified and monitored, and partially supported by tools



Example

Command: find

Syntax: find <pattern> <file>

Function: The find command is used to locate one or more instance of a given pattern in a text file. All lines in the file that contain the pattern are written to standard output. A line containing the pattern is written only once, regardless of the number of times the pattern occurs in it.

The pattern is any sequence of characters whose length does not exceed the maximum length of a line in the file .To include a blank in the pattern, the entire pattern must be enclosed in quotes (“”).To include quotation mark in the pattern ,two quotes in a row (““”) must be used.

Example:

find john myfile

display lines in the file **myfile** which contain **john**

find “john smith” in myfile

display lines in the file **myfile** which contain **john smith**

find “john”” smith” in myfile

display lines in the file **myfile** which contain **john” smith**

Parameters:

Pattern size:

- empty
- single character
- many character
- longer than any line in the file

Quoting:

- pattern is quoted
- pattern is not quoted
- pattern is improperly quoted

Embedded blanks:

- no embedded blank
- one embedded blank
- several embedded blanks

Embedded quotes:

- no embedded quotes
- one embedded quotes
- several embedded quotes

File name:

- good file name
- no file with this name

Environments:

Number of occurrence of pattern in file:

- none
- exactly one
- more than one

Pattern occurrences on target line:

- one
- more than one

Total Tests frames:
1944

Test Frame - Example:

Pattern size : empty

Quoting : pattern is quoted

Embedded blanks : several embedded blanks

Embedded quotes : no embedded quote

File name : good file name

Number of occurrence of pattern in file : none

Pattern occurrence on target line : one

Parameters:

Pattern size:

empty	[property Empty]
single character	[property NonEmpty]
many character	[property NonEmpty]
longer than any line in the file	[property NonEmpty]

Quoting:

pattern is quoted	[property quoted]
pattern is not quoted	[if NonEmpty]
pattern is improperly quoted	[if NonEmpty]

Embedded blanks:

no embedded blank	[if NonEmpty]
one embedded blank	[if NonEmpty and Quoted]
several embedded blanks	[if NonEmpty and Quoted]

Embedded quotes:

- | | |
|-------------------------|-----------------|
| no embedded quotes | [if NonEmpty] |
| one embedded quotes | [if NonEmpty] |
| several embedded quotes | [if NonEmpty] |

File name:

- | | |
|------------------------|--|
| good file name | |
| no file with this name | |

Environments:

Number of occurrence of pattern in file:

- | | |
|---------------|------------------------------------|
| none | [if NonEmpty] |
| exactly one | [if NonEmpty] [property Match] |
| more than one | [if NonEmpty] [property Match] |

**Total Tests frames:
678**

Pattern occurrences on target line:

- | | |
|---------------|--------------|
| one | [if Match] |
| more than one | [if Match] |

Parameters:

Pattern size:

empty	[property Empty]
single character	[property NonEmpty]
many character	[property NonEmpty]
longer than any line in the file	[error]

Quoting:

pattern is quoted	[property quoted]
pattern is not quoted	[if NonEmpty]
pattern is improperly quoted	[error]

Embedded blanks:

no embedded blank	[if NonEmpty]
one embedded blank	[if NonEmpty and Quoted]
several embedded blanks	[if NonEmpty and Quoted]

Embedded quotes:

- | | |
|-------------------------|----------------------------|
| no embedded quotes | [if NonEmpty] |
| one embedded quotes | [if NonEmpty] |
| several embedded quotes | [if NonEmpty] [single] |

File name:

- | | |
|------------------------|-----------|
| good file name | |
| no file with this name | [error] |

Environments:

Number of occurrence of pattern in file:

- | | |
|---------------|------------------------------------|
| none | [if NonEmpty] [single] |
| exactly one | [if NonEmpty] [property Match] |
| more than one | [if NonEmpty] [property Match] |

Total Tests frames:
40

Pattern occurrences on target line:

- | | |
|---------------|-------------------------|
| one | [if Match] |
| more than one | [if Match] [single] |

Test Frame :

Test case 28 : (Key = 3.1.3.2.1.2.1.)

Pattern size : many character

Quoting : pattern is quoted

Embedded blanks : several embedded blanks

Embedded quotes : one embedded quote

File name : good file name

Number of occurrence of pattern in file : exactly none

Pattern occurrence on target line : one

Command to set up the test:

```
copy/testing/sources/case_28 testfile
```

find command to perform the test:

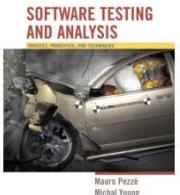
```
find "has" "one quote" testfile
```

Instruction for checking the test :

the following line should be display:

This line has “ one quote on it

Finite Models

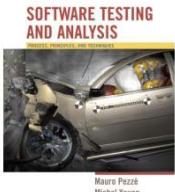


(c) 2007 Mauro Pezzè & Michal Young

Ch 5, slide 1

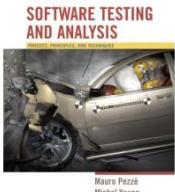
Learning objectives

- Understand goals and implications of finite state abstraction
- Learn how to model program control flow with graphs
- Learn how to model the software system structure with call graphs
- Learn how to model finite state behavior with finite state machines



Properties of Models

- **Compact:** representable and manipulable in a reasonably compact form
 - What is *reasonably compact* depends largely on how the model will be used
- **Predictive:** must represent some salient characteristics of the modeled artifact well enough to distinguish between *good* and *bad* outcomes of analysis
 - no single model represents all characteristics well enough to be useful for all kinds of analysis
- **Semantically meaningful:** it is usually necessary to interpret analysis results in a way that permits diagnosis of the causes of failure
- **Sufficiently general:** models intended for analysis of some important characteristic must be general enough for practical use in the intended domain of application

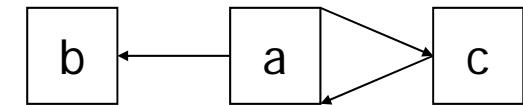
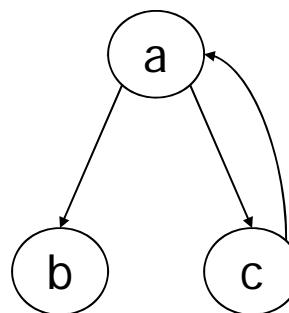


Graph Representations: directed graphs

- Directed graph:
 - N (set of nodes)
 - E (relation on the set of nodes) edges

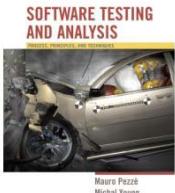
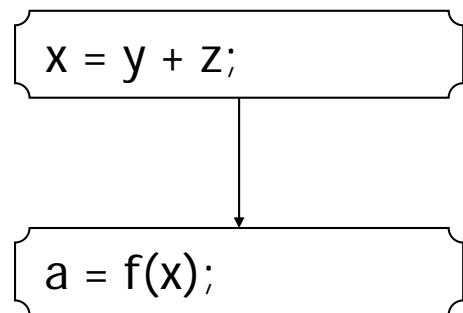
Nodes: $\{a, b, c\}$

Edges: $\{(a,b), (a, c), (c, a)\}$



Graph Representations: labels and code

- We can label nodes with the names or descriptions of the entities they represent.
 - If nodes a and b represent program regions containing assignment statements, we might draw the two nodes and an edge (a,b) connecting them in this way:



Multidimensional Graph Representations

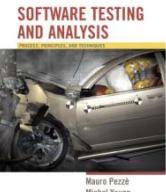
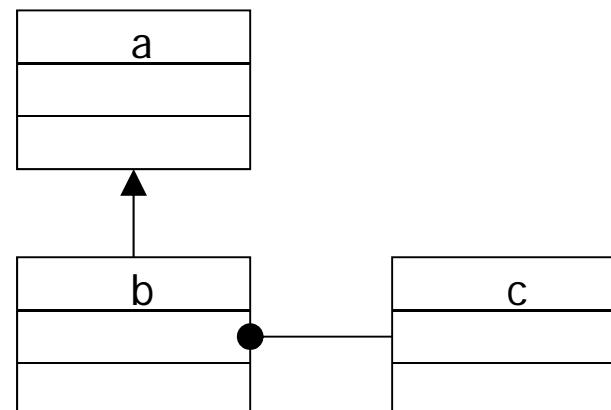
- Sometimes we draw a single diagram to represent more than one directed graph, drawing the shared nodes only once
 - class B extends (is a subclass of) class A
 - class B has a field that is an object of type C

extends relation

NODES = {A, B, C}
EDGES = {(A,B)}

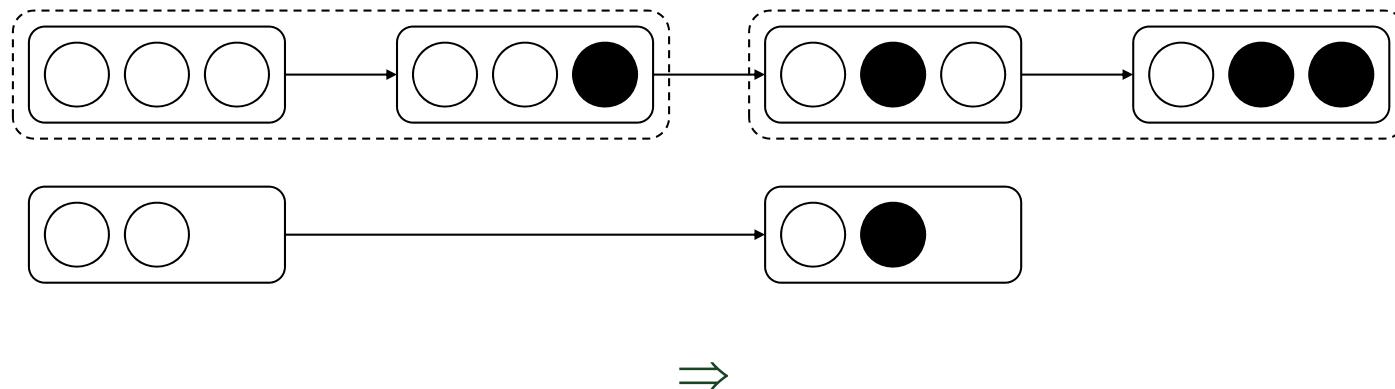
includes relation

NODES = {A, B, C}
EDGES = {(B,C)}

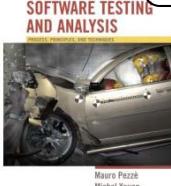
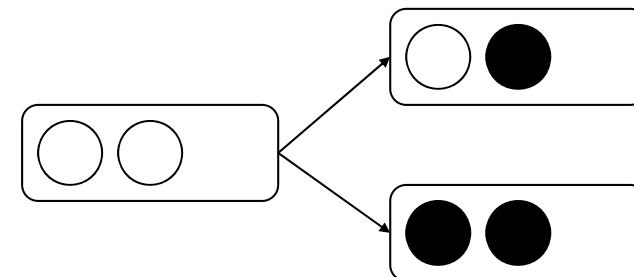
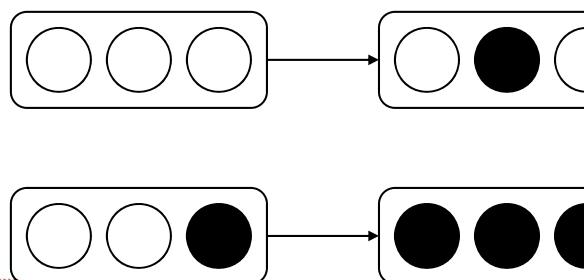


Finite Abstraction of Behavior

an abstraction function suppresses some details of program execution

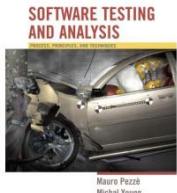


it lumps together execution states that differ with respect to the suppressed details but are otherwise identical



(Intraprocedural) Control Flow Graph

- nodes = regions of source code (basic blocks)
 - Basic block = maximal program region with a single entry and single exit point
 - Often statements are grouped in single regions to get a compact model
 - Sometime single statements are broken into more than one node to model control flow within the statement
- directed edges = possibility that program execution proceeds from the end of one region directly to the beginning of another

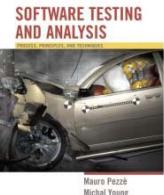
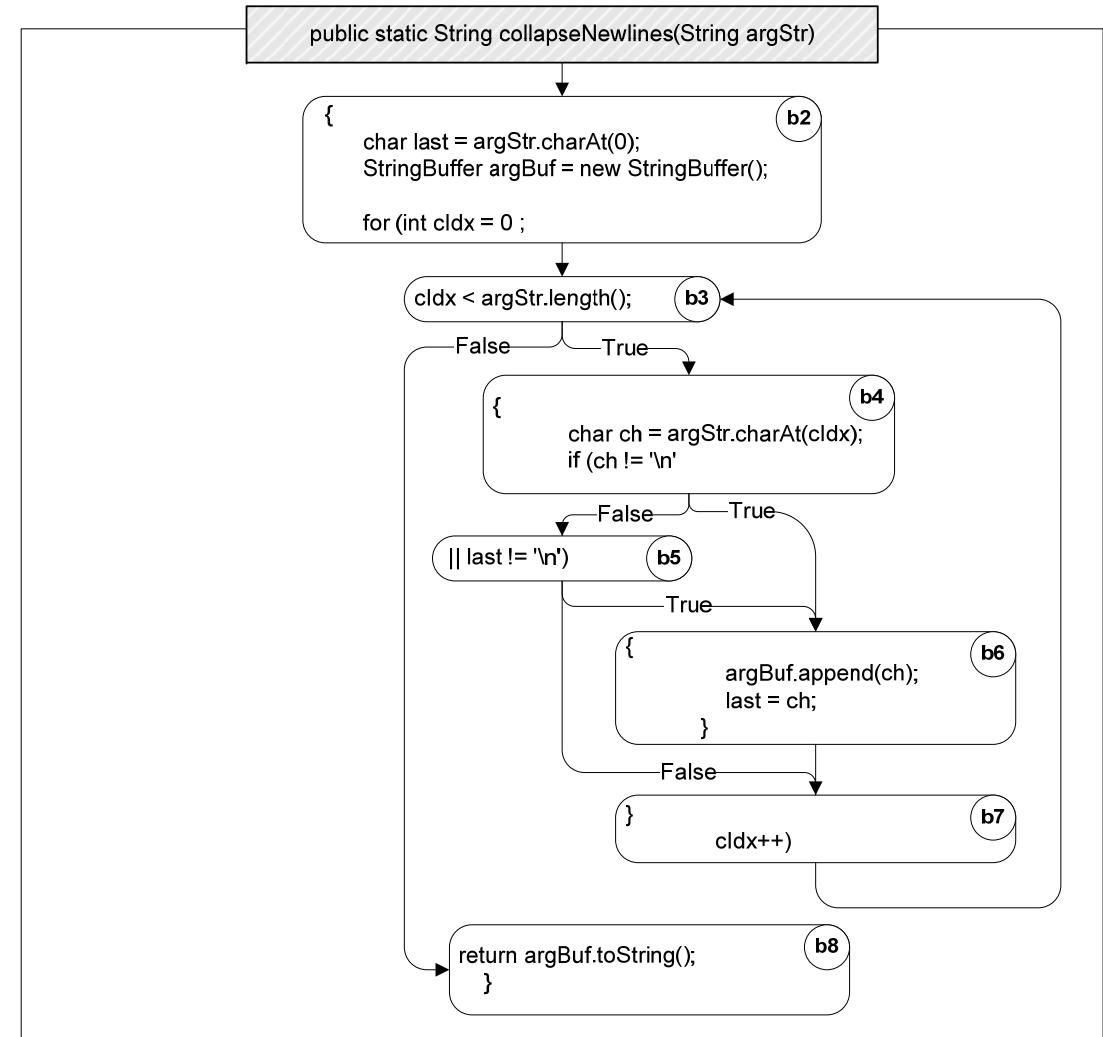


Example of Control Flow Graph

```
public static String collapseNewlines(String argStr)
{
    char last = argStr.charAt(0);
    StringBuffer argBuf = new StringBuffer();

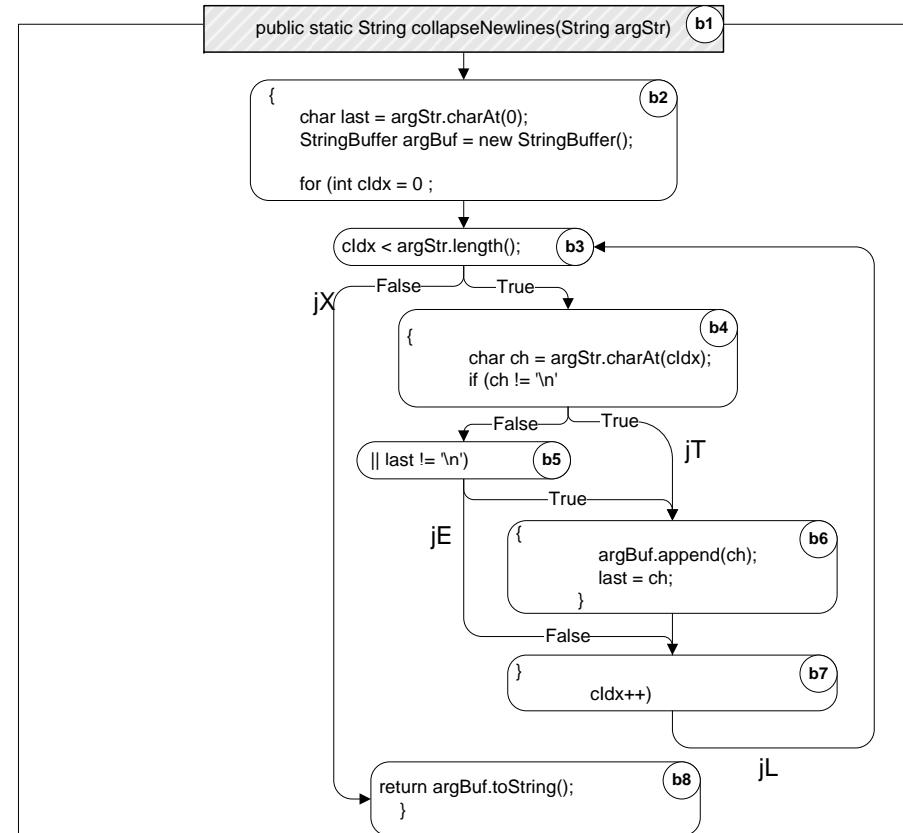
    for (int cIdx = 0 ; cIdx < argStr.length(); cIdx++)
    {
        char ch = argStr.charAt(cIdx);
        if (ch != '\n' || last != '\n')
        {
            argBuf.append(ch);
            last = ch;
        }
    }

    return argBuf.toString();
}
```



Linear Code Sequence and Jump (LCSJ)

Essentially subpaths of the control flow graph from one branch to another

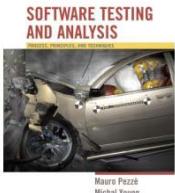


From	Sequence of basic blocs	To
Entry	b1 b2 b3	jX
Entry	b1 b2 b3 b4	jT
Entry	b1 b2 b3 b4 b5	jE
Entry	b1 b2 b3 b4 b5 b6 b7	jL
jX	b8	ret
jL	b3 b4	jT
jL	b3 b4 b5	jE
jL	b3 b4 b5 b6 b7	jL



Interprocedural control flow graph

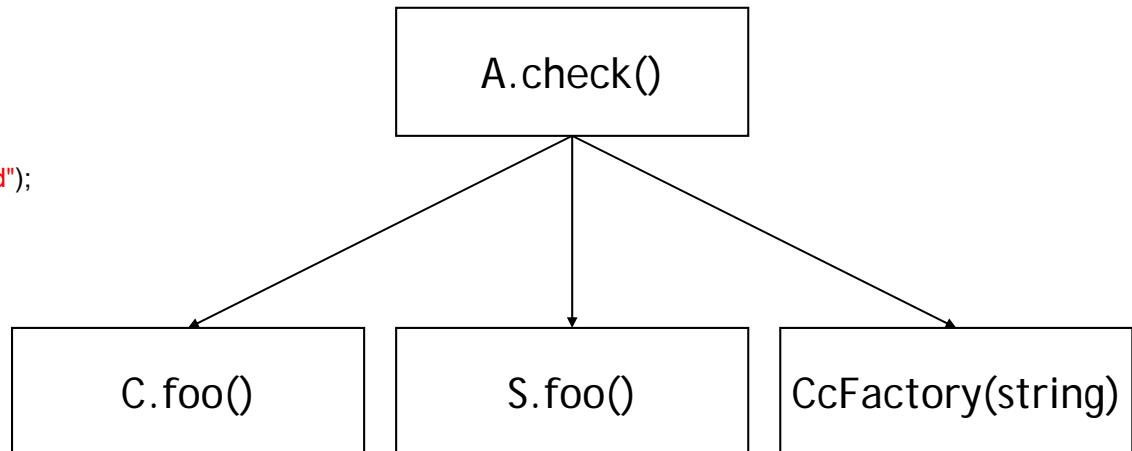
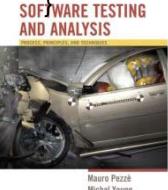
- Call graphs
 - Nodes represent procedures
 - Methods
 - C functions
 - ...
 - Edges represent *calls* relation



Overestimating the *calls* relation

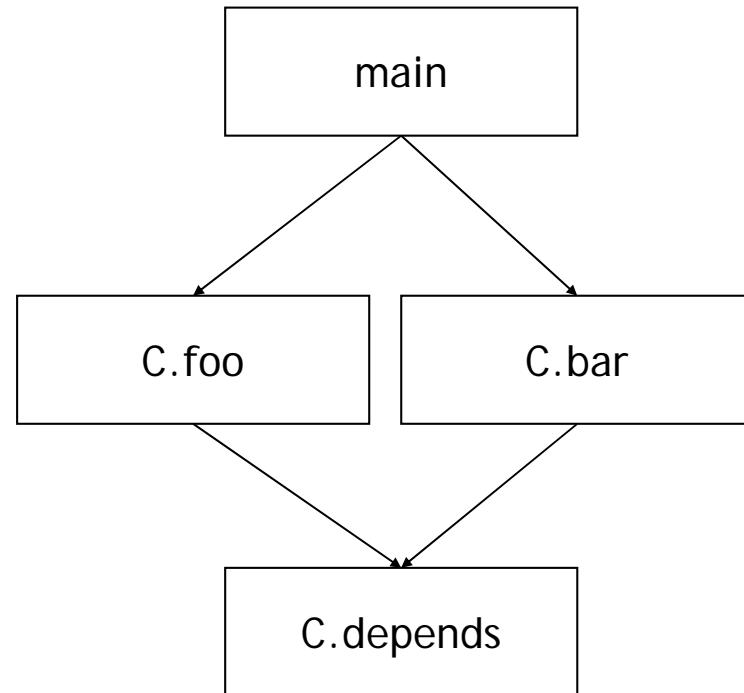
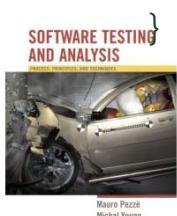
The static call graph includes calls through dynamic bindings that never occur in execution.

```
public class C {  
    public static C cFactory(String kind) {  
        if (kind == "C") return new C();  
        if (kind == "S") return new S();  
        return null;  
    }  
    void foo() {  
        System.out.println("You called the parent's method");  
    }  
    public static void main(String args[]) {  
        (new A()).check();  
    }  
}  
class S extends C {  
    void foo() {  
        System.out.println("You called the child's method");  
    }  
}  
class A {  
    void check() {  
        C myC = C.cFactory("S");  
        myC.foo();  
    }  
}
```



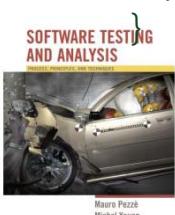
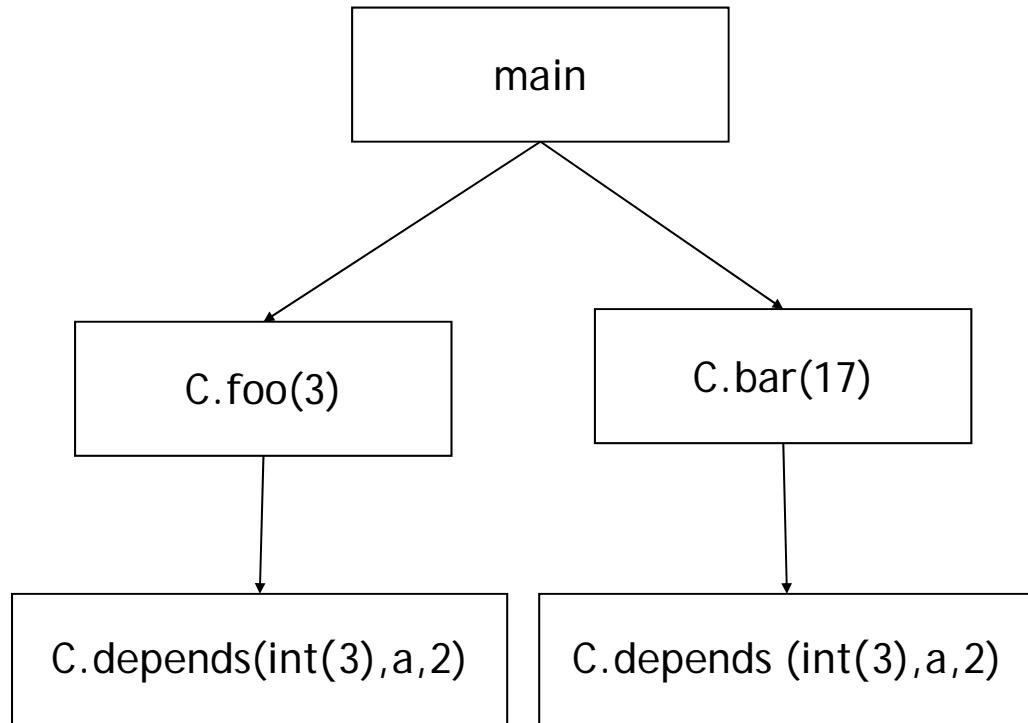
Context Insensitive Call graphs

```
public class Context {  
    public static void main(String args[]) {  
        Context c = new Context();  
        c.foo(3);  
        c.bar(17);  
    }  
  
    void foo(int n) {  
        int[] myArray = new int[ n ];  
        depends( myArray, 2 );  
    }  
  
    void bar(int n) {  
        int[] myArray = new int[ n ];  
        depends( myArray, 16 );  
    }  
  
    void depends( int[] a, int n ) {  
        a[n] = 42;  
    }  
}
```

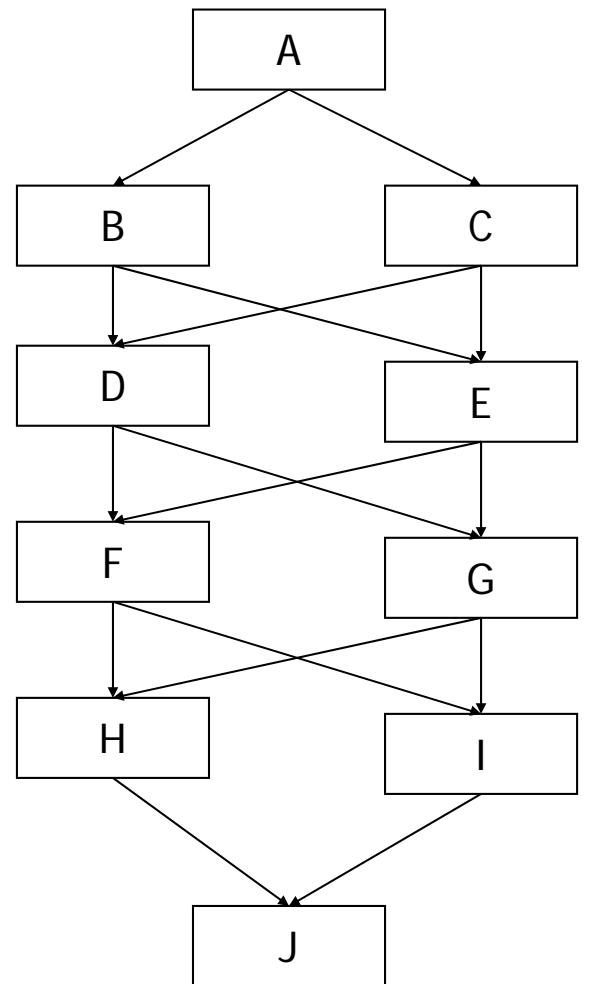


Context Sensitive Call graphs

```
public class Context {  
    public static void main(String args[]) {  
        Context c = new Context();  
        c.foo(3);  
        c.bar(17);  
    }  
  
    void foo(int n) {  
        int[] myArray = new int[ n ];  
        depends( myArray, 2 );  
    }  
  
    void bar(int n) {  
        int[] myArray = new int[ n ];  
        depends( myArray, 16 );  
    }  
  
    void depends( int[] a, int n ) {  
        a[n] = 42;  
    }  
}
```



Context Sensitive CFG exponential growth



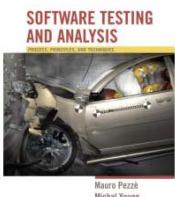
1 context A

2 contexts AB AC

4 contexts ABD ABE ACD ACE

8 contexts ...

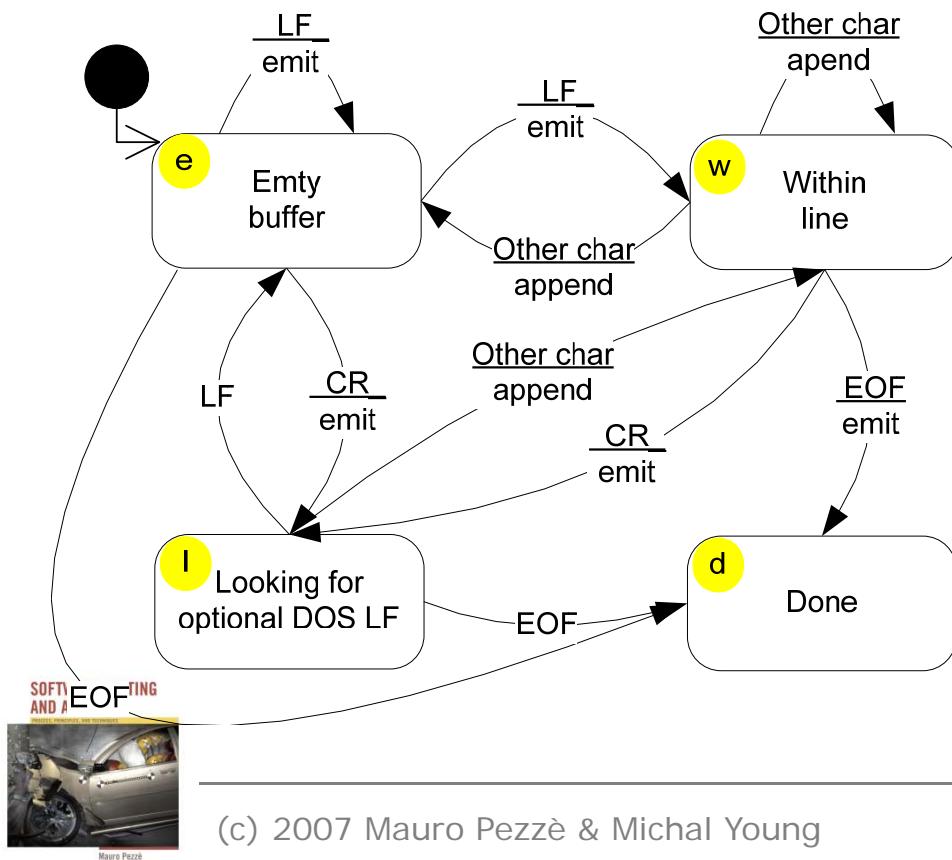
16 calling contexts ...



Finite state machines

- finite set of states (nodes)
- set of transitions among states (edges)

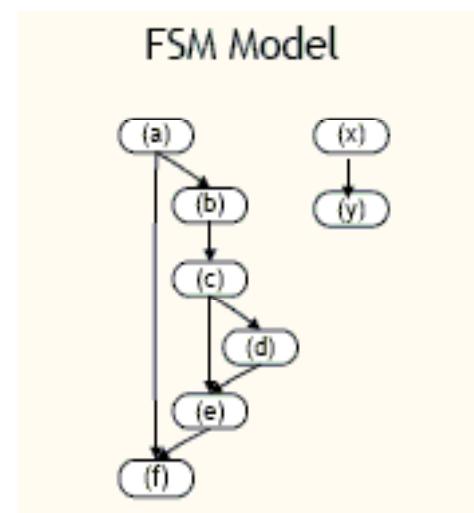
Graph representation (Mealy machine)



Tabular representation

	LF	CR	EOF	other
e	e/emit	e/emit	d/-	w/append
w	e/emit	e/emit	d/emit	w/append
I	e/-		d/-	w/append

Using Models to Reason about System Properties



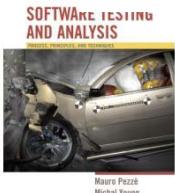
The model satisfies
The specification

The model is syntactically
well-formed, consistent
and complete

Program

```
...  
public static Table1  
getTable1() {  
    if (ref == null) {  
        synchronized(Table1) {  
            if (ref == null){  
                ref = new Table1();  
                ref.initialize();  
            }  
        }  
    }  
    return ref;  
}...
```

The model accurately
represents the program



```

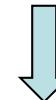
1  /** Convert each line from standard input */
2  void transduce() {
3
4      #define BUflen 1000
5      char buf[BUflen]; /* Accumulate line into this buffer */
6      int pos = 0; /* Index for next character in buffer */
7
8      char inChar; /* Next character from input */
9
10     int atCR = 0; /* 0="within line", 1="optional DOS LF" */
11
12     while ((inChar = getchar()) != EOF) {
13         switch (inChar) {
14             case LF:
15                 if (atCR) { /* Optional DOS LF */
16                     atCR = 0;
17                 } else { /* Encountered CR within line */
18                     emit(buf, pos);
19                     pos = 0;
20                 }
21                 break;
22             case CR:
23                 emit(buf, pos);
24                 pos = 0;
25                 atCR = 1;
26                 break;
27             default:
28                 if (pos >= BUflen - 2) fail("Buffer overflow");
29                 buf[pos++] = inChar;
30             } /* switch */
31         }
32         if (pos > 0) {
33             emit(buf, pos);
34         }
35     }

```

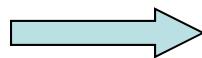


Abstraction Function

	<i>Abstract state</i>	<i>Concrete state</i>		
e (Empty buffer)	3 – 13	Lines	atCR	pos
w (Within line)	13	0	> 0	
l (Looking for LF)	13	1	0	
d (Done)	36	–	–	–

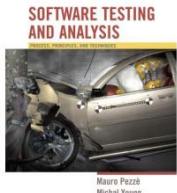


	LF	CR	EOF	other
e	e / emit	1 / emit	d / –	w / append
w	e / emit	1 / emit	d / emit	w / append
l	e / –	1 / emit	d / –	w / append

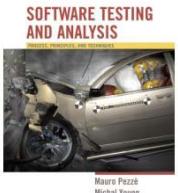


Summary

- Models must be much simpler than the artifact they describe to be understandable and analyzable
- Must also be sufficiently detailed to be useful
- CFG are built from software
- FSM can be built before software to document intended behavior



Structural Testing

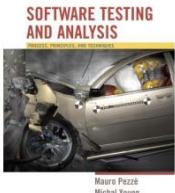


(c) 2007 Mauro Pezzè & Michal Young

Ch 12, slide 1

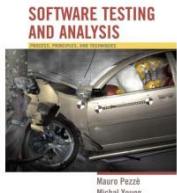
Learning objectives

- Understand rationale for structural testing
 - How structural (code-based or glass-box) testing complements functional (black-box) testing
- Recognize and distinguish basic terms
 - Adequacy, coverage
- Recognize and distinguish characteristics of common structural criteria
- Understand practical uses and limitations of structural testing



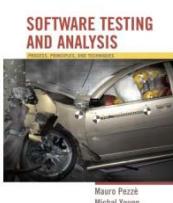
“Structural” testing

- 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.



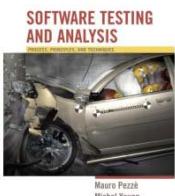
Why structural (code-based) testing?

- One way of answering the question “What is *missing* in our test suite?”
 - If part of a program is not executed by any test case in the suite, faults in that part cannot be exposed
 - But what’s a “part”?
 - Typically, a control flow element or combination:
 - Statements (or CFG nodes), Branches (or CFG edges)
 - Fragments and combinations: Conditions, paths
- Complements functional testing: Another way to recognize cases that are treated differently
 - Recall fundamental rationale: Prefer test cases that are treated *differently* over cases treated the same



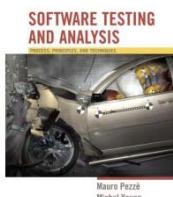
No guarantees

- Executing all control flow elements does not guarantee finding all faults
 - Execution of a faulty statement may not always result in a failure
 - The state may not be corrupted when the statement is executed with some data values
 - Corrupt state may not propagate through execution to eventually lead to failure
- What is the value of structural coverage?
 - Increases confidence in thoroughness of testing
 - Removes some obvious *inadequacies*



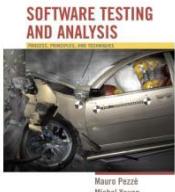
Structural testing *complements* functional testing

- 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



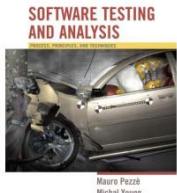
Structural testing in practice

- Create functional test suite first, then measure structural coverage to identify see what is missing
- Interpret unexecuted elements
 - may be due to natural differences between specification and implementation
 - or may reveal flaws of the software or its development process
 - inadequacy of specifications that do not include cases present in the implementation
 - coding practice that radically diverges from the specification
 - inadequate functional test suites
- Attractive because automated
 - coverage measurements are convenient progress indicators
 - sometimes used as a criterion of completion
 - use with caution: does not ensure *effective* test suites



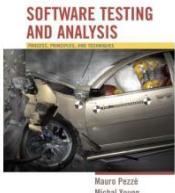
Statement testing

- Adequacy criterion: each statement (or node in the CFG) must be executed at least once
- Coverage:
$$\frac{\text{\# executed statements}}{\text{\# statements}}$$
- Rationale: a fault in a statement can only be revealed by executing the faulty statement



Statements or blocks?

- Nodes in a control flow graph often represent basic blocks of multiple statements
 - Some standards refer to *basic block* coverage or *node coverage*
 - Difference in granularity, not in concept
- No essential difference
 - 100% node coverage <-> 100% statement coverage
 - but levels will differ below 100%
 - A test case that improves one will improve the other
 - though not by the same amount, in general



Example

$T_0 =$

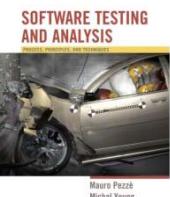
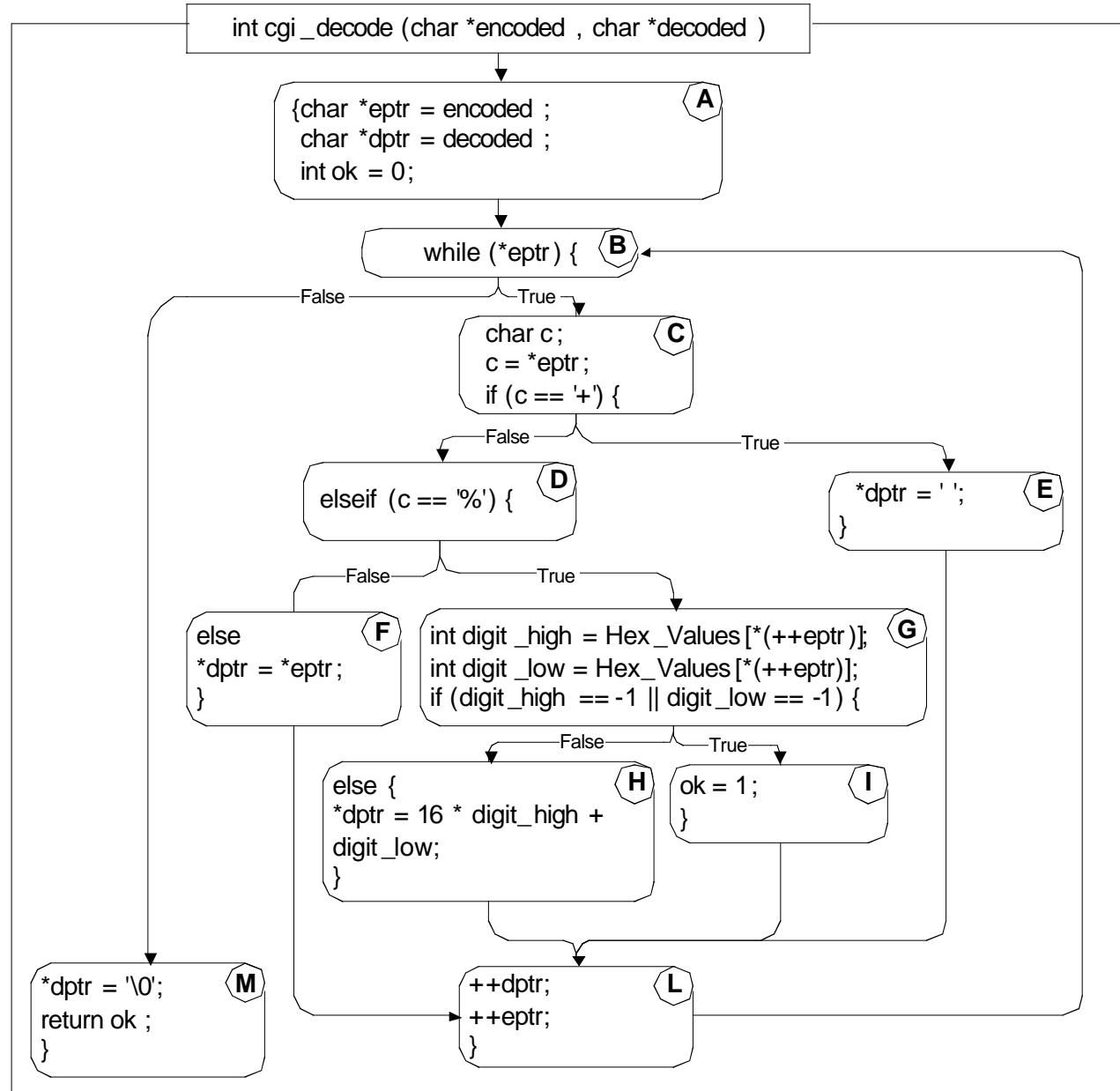
```
{"", "test",
"test+case%1Dadequacy"}  
17/18 = 94% Stmt Cov.
```

$T_1 =$

```
{"adequate+test%0Dexecuti
on%7U"}  
18/18 = 100% Stmt Cov.
```

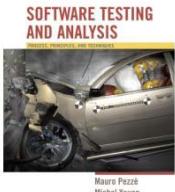
$T_2 =$

```
{"%3D", "%A", "a+b",
"test"}  
18/18 = 100% Stmt Cov.
```



Coverage is not size

- Coverage does not depend on the number of test cases
 - $T_0, T_1 : T_1 >_{\text{coverage}} T_0 \quad T_1 <_{\text{cardinality}} T_0$
 - $T_1, T_2 : T_2 =_{\text{coverage}} T_1 \quad T_2 >_{\text{cardinality}} T_1$
- Minimizing test suite size is seldom the goal
 - small test cases make failure diagnosis easier
 - a failing test case in T_2 gives more information for fault localization than a failing test case in T_1



“All statements” can miss some cases

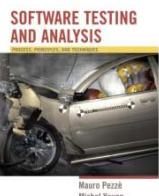
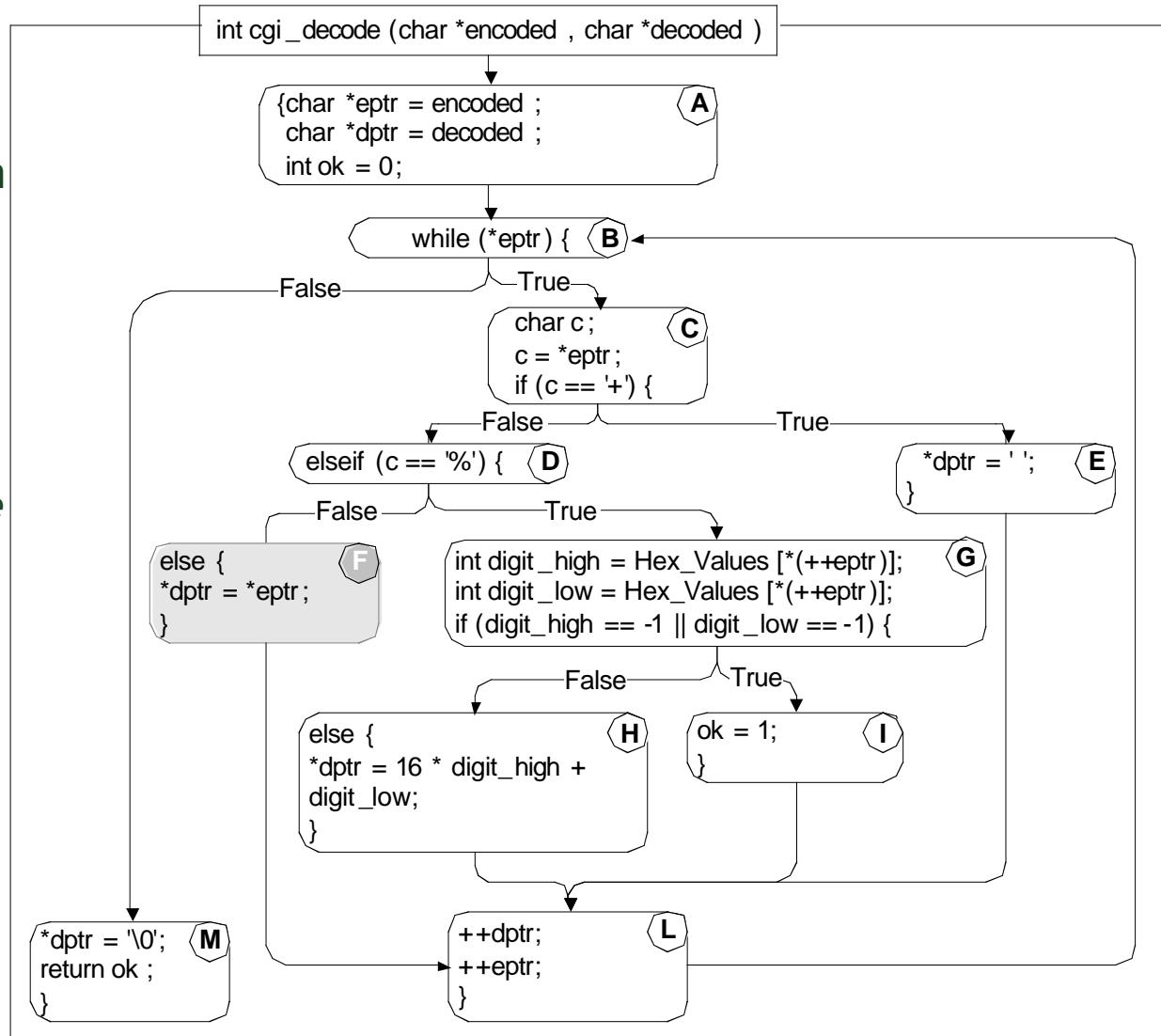
- Complete statement coverage may not imply executing all branches in a program
- Example:
 - Suppose block F were missing
 - Statement adequacy would not require *false* branch from D to L

$T_3 =$

{ "", "+%0D+%4J" }

100% Stmt Cov.

No *false* branch from D



Branch testing

- Adequacy criterion: each branch (edge in the CFG) must be executed at least once
- Coverage:

executed branches

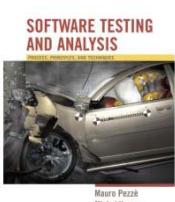
branches

$$T_3 = \{"", "+%0D+%4J"\}$$

100% Stmt Cov. 88% Branch Cov. (7/8 branches)

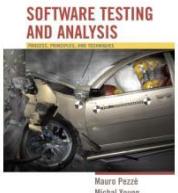
$$T_2 = {"%3D", "%A", "a+b", "test"}$$

100% Stmt Cov. 100% Branch Cov. (8/8 branches)



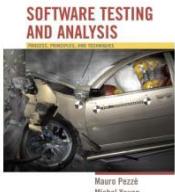
Statements vs branches

- Traversing all edges of a graph causes all nodes to be visited
 - So test suites that satisfy the branch adequacy criterion for a program P also satisfy the statement adequacy criterion for the same program
- The converse is not true (see T_3)
 - A statement-adequate (or node-adequate) test suite may not be branch-adequate (edge-adequate)



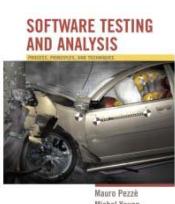
“All branches” can still miss conditions

- Sample fault: missing operator (negation)
 $\text{digit_high} == 1 \text{ || digit_low} == -1$
- Branch adequacy criterion can be satisfied by varying only `digit_low`
 - The faulty sub-expression might never determine the result
 - We might never really test the faulty condition, even though we tested both outcomes of the branch



Condition testing

- Branch coverage exposes faults in how a computation has been decomposed into cases
 - intuitively attractive: check the programmer's case analysis
 - but only roughly: groups cases with the same outcome
- Condition coverage considers case analysis in more detail
 - also *individual conditions* in a compound Boolean expression
 - e.g., both parts of `digit_high == 1 || digit_low == -1`

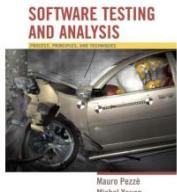


Basic condition testing

- Adequacy criterion: each basic condition must be executed at least once
- Coverage:

truth values taken by all basic conditions

$$2^* \# \text{ basic conditions}$$



Basic conditions vs branches

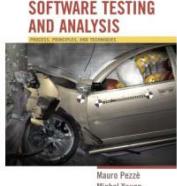
- Basic condition adequacy criterion can be satisfied without satisfying branch coverage

$T4 = \{\text{"first+test\%9Ktest\%K9"}\}$

satisfies basic condition adequacy

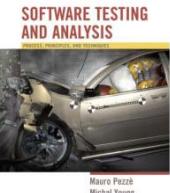
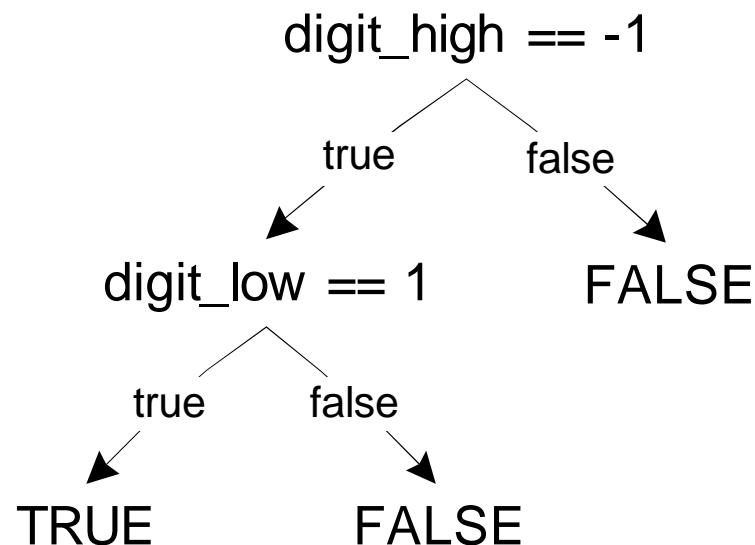
does not satisfy branch condition adequacy

Branch and basic condition are not comparable
(neither implies the other)



Covering branches and conditions

- Branch and condition adequacy:
 - cover all conditions and all decisions
- Compound condition adequacy:
 - Cover all possible evaluations of compound conditions
 - Cover all branches of a decision tree

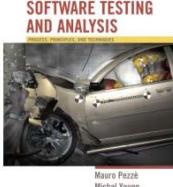


Compound conditions: Exponential complexity

$$(((a \mid\mid b) \ \&\& c) \mid\mid d) \ \&\& e$$

Test Case	a	b	c	d	e
(1)	T	-	T	-	T
(2)	F	T	T	-	T
(3)	T	-	F	T	T
(4)	F	T	F	T	T
(5)	F	F	-	T	T
(6)	T	-	T	-	F
(7)	F	T	T	-	F
(8)	T	-	F	T	F
(9)	F	T	F	T	F
(10)	F	F	-	T	F
(11)	T	-	F	F	-
(12)	F	T	F	F	-
(13)	F	F	-	F	-

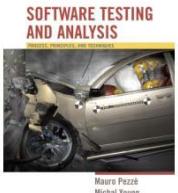
- short-circuit evaluation often reduces this to a more manageable number, but not always



Summary

- Structural Testing complements functional testing.
- Structural coverage measures what elements of the implementation we have missed testing.
- Some common control flow coverage criteria are
 - Statement coverage
 - Branch coverage
 - Condition coverage
 - Compound condition coverage
- Coverage does not depend on the number of test cases

Structural Testing

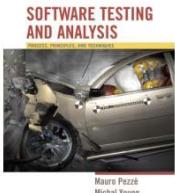


(c) 2007 Mauro Pezzè & Michal Young

Ch 12, slide 1

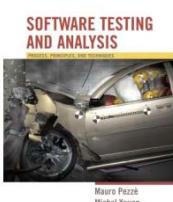
Learning objectives

- Understand rationale for structural testing
 - How structural (code-based or glass-box) testing complements functional (black-box) testing
- Recognize and distinguish basic terms
 - Adequacy, coverage
- Recognize and distinguish characteristics of common structural criteria
- Understand practical uses and limitations of structural testing



Modified condition/decision (MC/DC)

- Motivation: Effectively test **important combinations** of conditions, without exponential blowup in test suite size
 - “Important” combinations means: Each basic condition shown to independently affect the outcome of each decision
- Requires:
 - For each basic condition C, two test cases,
 - values of all *evaluated* conditions except C are the same
 - compound condition as a whole evaluates to *true* for one and *false* for the other



MC/DC: linear complexity

- $N+1$ test cases for N basic conditions

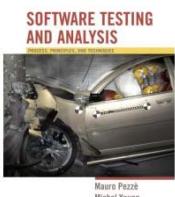
$$(((a \mid\mid b) \ \&\& \ c) \mid\mid d) \ \&\& \ e$$

Test Case	a	b	c	d	e	outcome
(1)	<u>true</u>	--	<u>true</u>	--	<u>true</u>	true
(2)	false	<u>true</u>	true	--	true	true
(3)	true	--	false	<u>true</u>	true	true
(6)	true	--	true	--	<u>false</u>	false
(11)	true	--	<u>false</u>	<u>false</u>	--	false
(13)	<u>false</u>	<u>false</u>	--	false	--	false

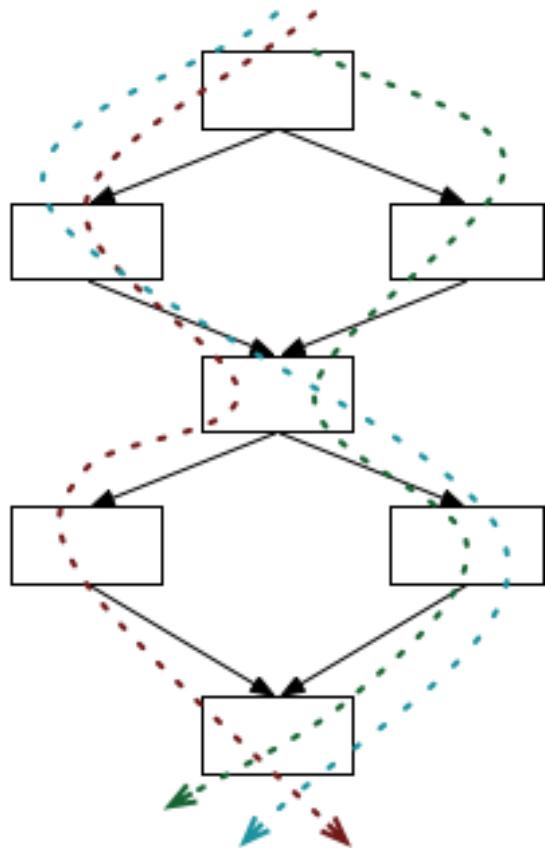
- Underlined values independently affect the output of the decision
- Required by the RTCA/DO-178B standard

Comments on MC/DC

- MC/DC is
 - basic condition coverage (C)
 - branch coverage (DC)
 - plus one additional condition (M):
every condition must *independently affect* the decision's output
- It is subsumed by compound conditions and subsumes all other criteria discussed so far
 - stronger than statement and branch coverage
- A good balance of thoroughness and test size (and therefore widely used)



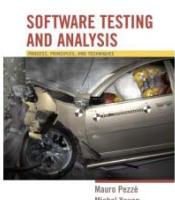
Paths? (Beyond individual branches)



- Should we explore sequences of branches (paths) in the control flow?
- Many more paths than branches
 - A pragmatic compromise will be needed

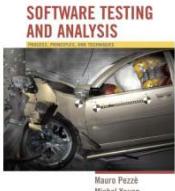
Path adequacy

- Decision and condition adequacy criteria consider individual program decisions
- Path testing focuses consider combinations of decisions along paths
- Adequacy criterion: each path must be executed at least once
- Coverage:
$$\frac{\text{\# executed paths}}{\text{\# paths}}$$



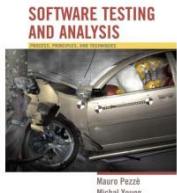
Practical path coverage criteria

- The number of paths in a program with loops is unbounded
 - the simple criterion is usually impossible to satisfy
- For a feasible criterion: Partition infinite set of paths into a finite number of classes
- Useful criteria can be obtained by limiting
 - the number of traversals of loops
 - the length of the paths to be traversed
 - the dependencies among selected paths

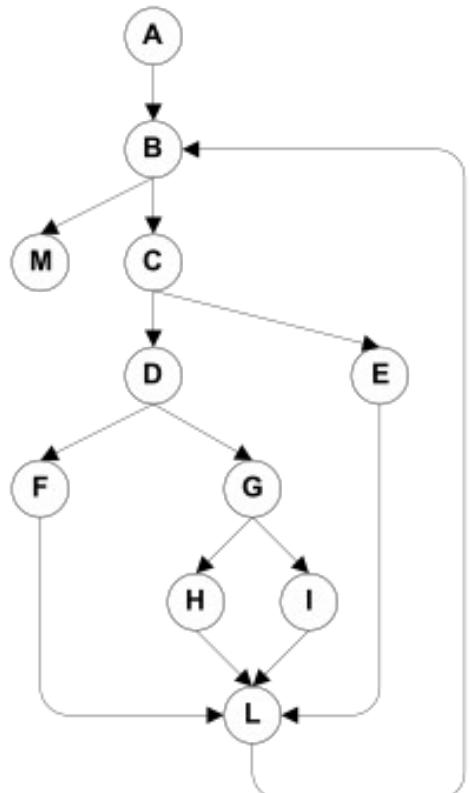


Boundary interior path testing

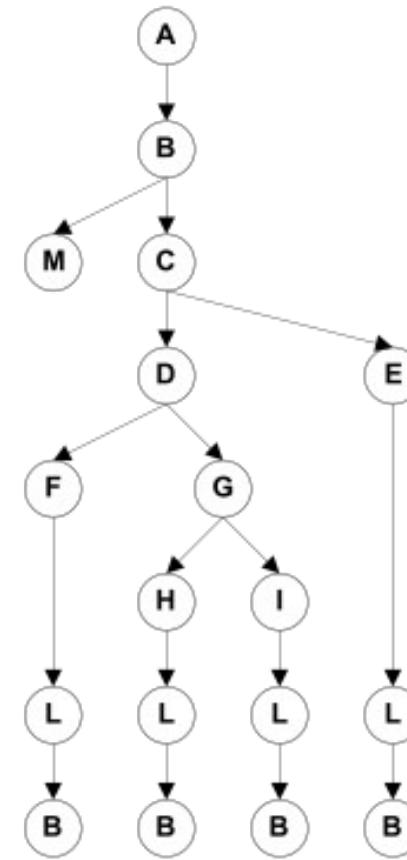
- Group together paths that differ only in the subpath they follow when repeating the body of a loop
 - Follow each path in the control flow graph up to the first repeated node
 - The set of paths from the root of the tree to each leaf is the required set of subpaths for boundary/interior coverage



Boundary interior adequacy for cgi-decode



(i)



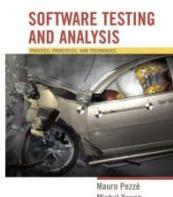
(ii)

Limitations of boundary interior adequacy

- The number of paths can still grow exponentially

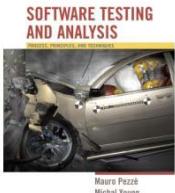
```
if (a) {  
    s1;  
}  
if (b) {  
    s2;  
}  
if (c) {  
    s3;  
}  
...  
if (x) {  
    sn;  
}
```

- The subpaths through this control flow can include or exclude each of the statements S_i , so that in total N branches result in 2^N paths that must be traversed
- Choosing input data to force execution of one particular path may be very difficult, or even impossible if the conditions are not independent



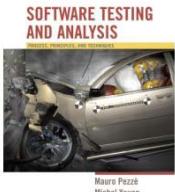
Loop boundary adequacy

- Variant of the boundary/interior criterion that treats loop boundaries similarly but is less stringent with respect to other differences among paths
- Criterion: A test suite satisfies the loop boundary adequacy criterion iff for every loop:
 - In at least one test case, the loop body is iterated zero times
 - In at least one test case, the loop body is iterated once
 - In at least one test case, the loop body is iterated more than once
- Corresponds to the cases that would be considered in a formal correctness proof for the loop



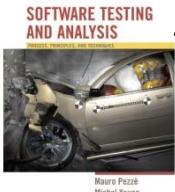
LCSAJ adequacy

- Linear Code Sequence And Jumps:
sequential subpath in the CFG starting and ending in a branch
 - TER_1 = statement coverage
 - TER_2 = branch coverage
 - TER_{n+2} = coverage of n consecutive LCSAJs



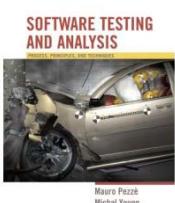
Cyclomatic adequacy

- Cyclomatic number:
number of independent paths in the CFG
 - A path is representable as a bit vector, where each component of the vector represents an edge
 - “Dependence” is ordinary linear dependence between (bit) vectors
- If $e = \#edges$, $n = \#nodes$, $c = \#\text{connected components}$ of a graph, it is:
 - $e - n + c$ for an arbitrary graph
 - $e - n + 2$ for a CFG
- Cyclomatic coverage counts the number of independent paths that have been exercised, relative to cyclomatic complexity



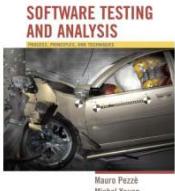
Towards procedure call testing

- The criteria considered to this point measure coverage of control flow within individual procedures.
 - not well suited to integration or system testing
- Choose a coverage granularity commensurate with the granularity of testing
 - if unit testing has been effective, then faults that remain to be found in integration testing will be primarily interface faults, and testing effort should focus on interfaces between units rather than their internal details

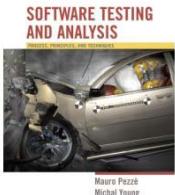
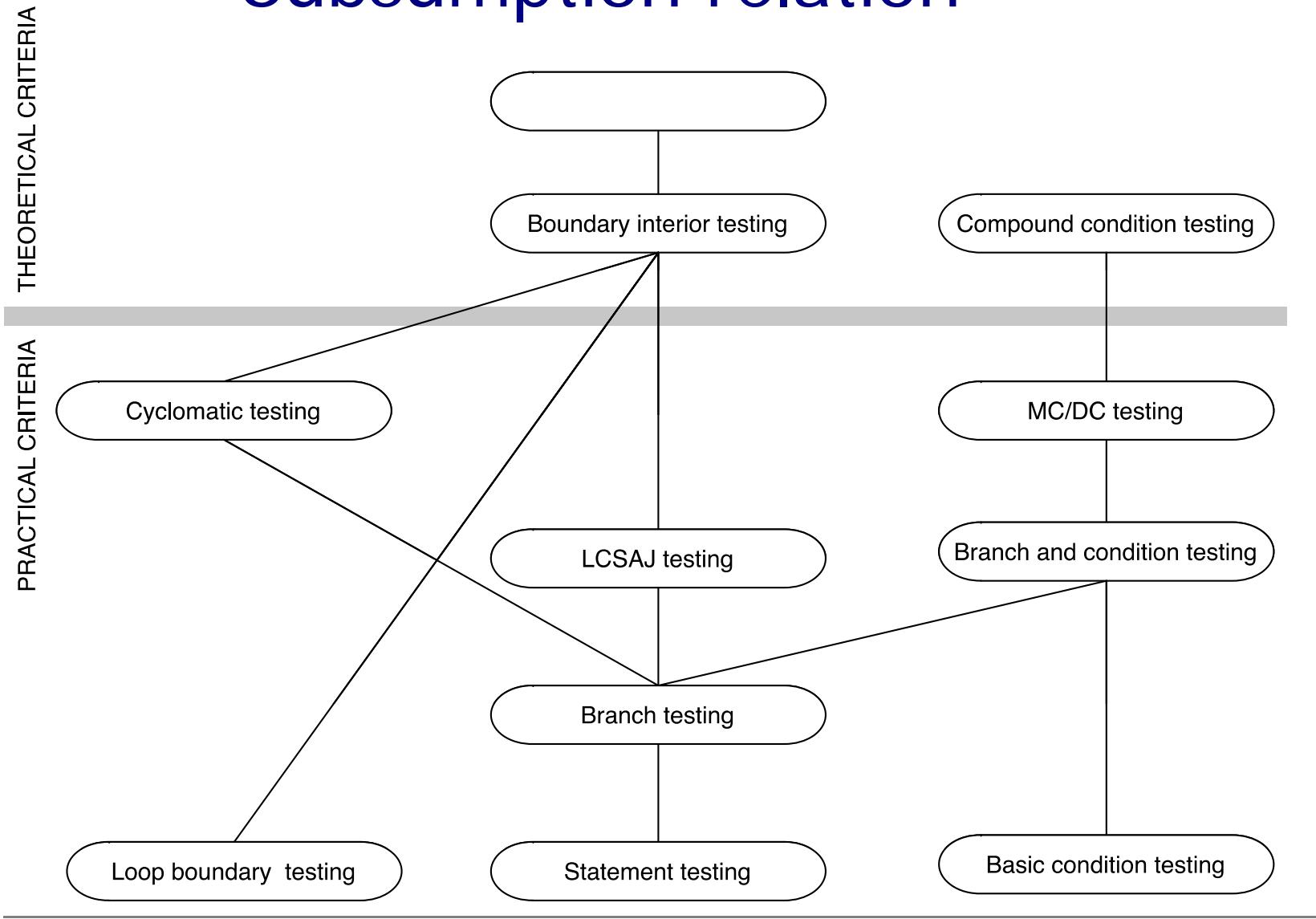


Procedure call testing

- Procedure entry and exit testing
 - procedure may have multiple entry points (e.g., Fortran) and multiple exit points
- Call coverage
 - The same entry point may be called from many points

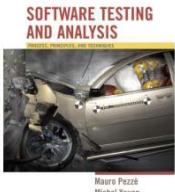


Subsumption relation



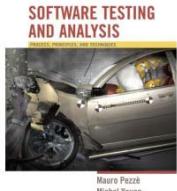
Satisfying structural criteria

- Sometimes criteria may not be satisfiable
 - The criterion requires execution of
 - statements that cannot be executed as a result of
 - defensive programming
 - code reuse (reusing code that is more general than strictly required for the application)
 - conditions that cannot be satisfied as a result of
 - interdependent conditions
 - paths that cannot be executed as a result of
 - interdependent decisions



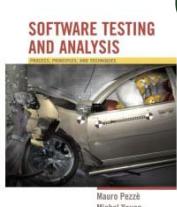
Satisfying structural criteria

- Large amounts of *fossil* code may indicate serious maintainability problems
 - But some unreachable code is common even in well-designed, well-maintained systems
- Solutions:
 - make allowances by setting a coverage goal less than 100%
 - require justification of elements left uncovered
 - RTCA-DO-178B and EUROCAE ED-12B for modified MC/DC

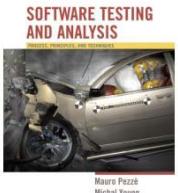


Summary

- 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



Dependence and Data Flow Models

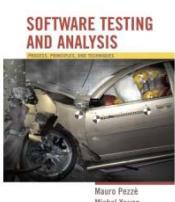


(c) 2007 Mauro Pezzè & Michal Young

Ch 6, slide 1

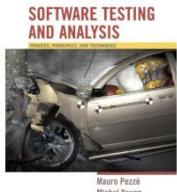
Why Data Flow Models?

- Models from Chapter 5 emphasized control
 - Control flow graph, call graph, finite state machines
- We also need to reason about dependence
 - Where does this value of x come from?
 - What would be affected by changing this?
 - ...
- Many program analyses and test design techniques use data flow information
 - Often in combination with control flow
 - Example: “Taint” analysis to prevent SQL injection attacks
 - Example: Dataflow test criteria (Ch.13)



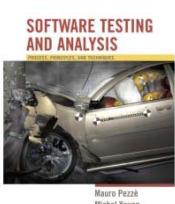
Learning objectives

- Understand basics of data-flow models and the related concepts (def-use pairs, dominators...)
- Understand some analyses that can be performed with the data-flow model of a program
 - The data flow analyses to build models
 - Analyses that use the data flow models
- Understand basic trade-offs in modeling data flow
 - variations and limitations of data-flow models and analyses, differing in precision and cost



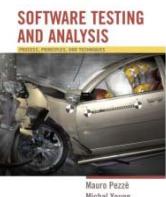
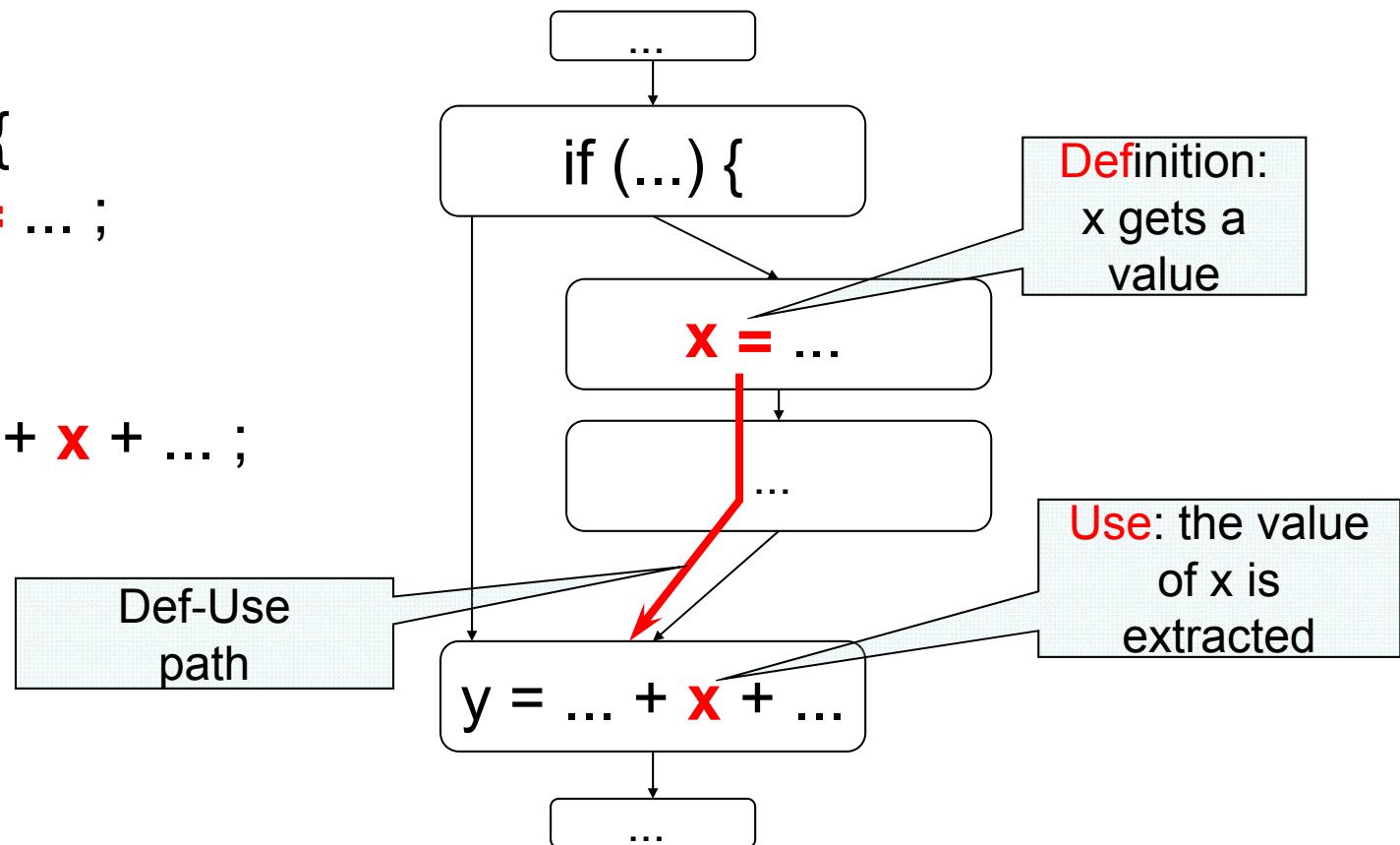
Def-Use Pairs (1)

- A def-use (du) pair associates a point in a program where a value is produced with a point where it is used
- Definition: where a variable gets a value
 - Variable declaration (often the special value “uninitialized”)
 - Variable initialization
 - Assignment
 - Values received by a parameter
- Use: extraction of a value from a variable
 - Expressions
 - Conditional statements
 - Parameter passing
 - Returns



Def-Use Pairs

```
...
if (...) {
    X = ... ;
}
y = ... + X + ... ;
```



Def-Use Pairs (3)

```
/** Euclid's algorithm */
public class GCD
{
    public int gcd(int x, int y) {
        int tmp;          // A: def x, y, tmp
        while (y != 0) {  // B: use y
            tmp = x % y; // C: def tmp; use x, y
            x = y;         // D: def x; use y
            y = tmp;        // E: def y; use tmp
        }
        return x;          // F: use x
    }
}
```

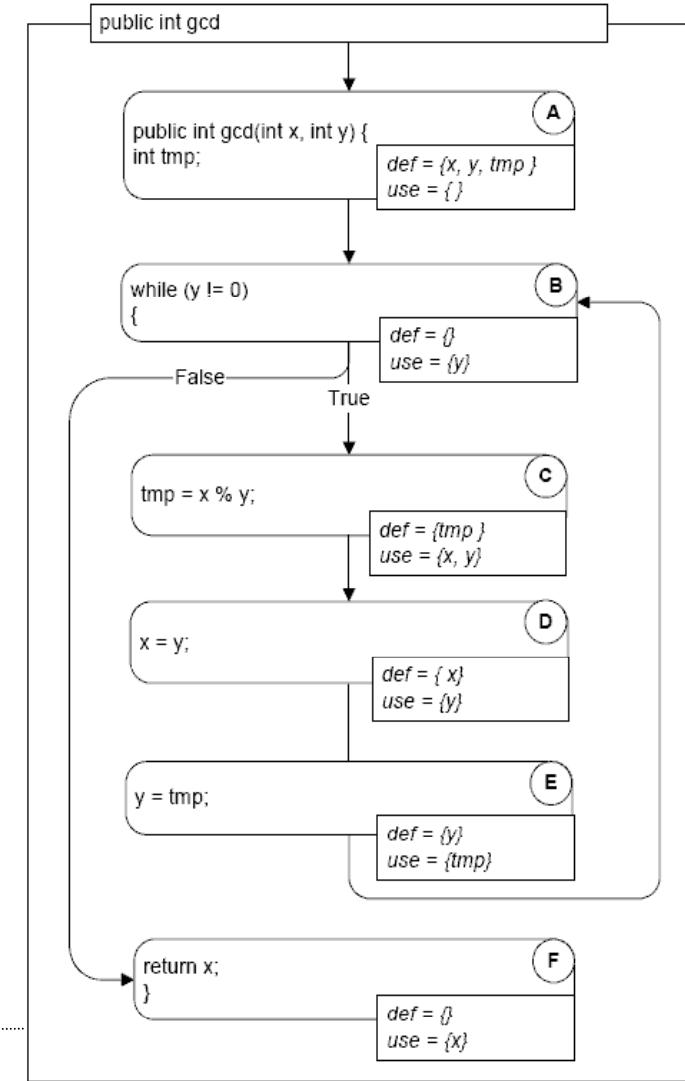
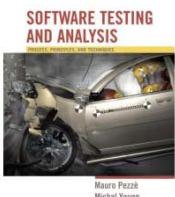


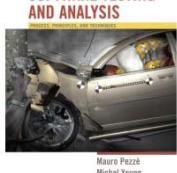
Figure 6.2, page 79



Def-Use Pairs (3)

- A **definition-clear** path is a path along the CFG from a definition to a use of the same variable without* another definition of the variable between
 - If, instead, another definition is present on the path, then the latter definition **kills** the former
- A def-use pair is formed if and only if there is a definition-clear path between the definition and the use

**There is an over-simplification here, which we will repair later.*

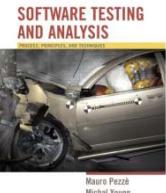
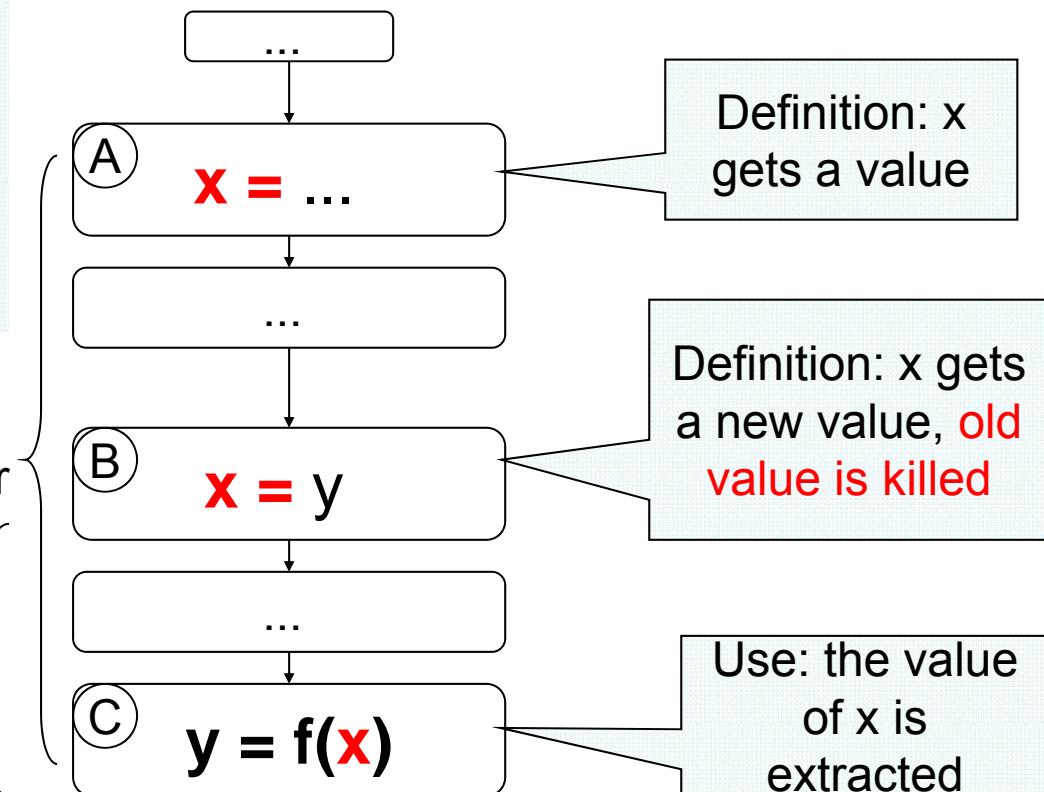


Definition-Clear or Killing

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

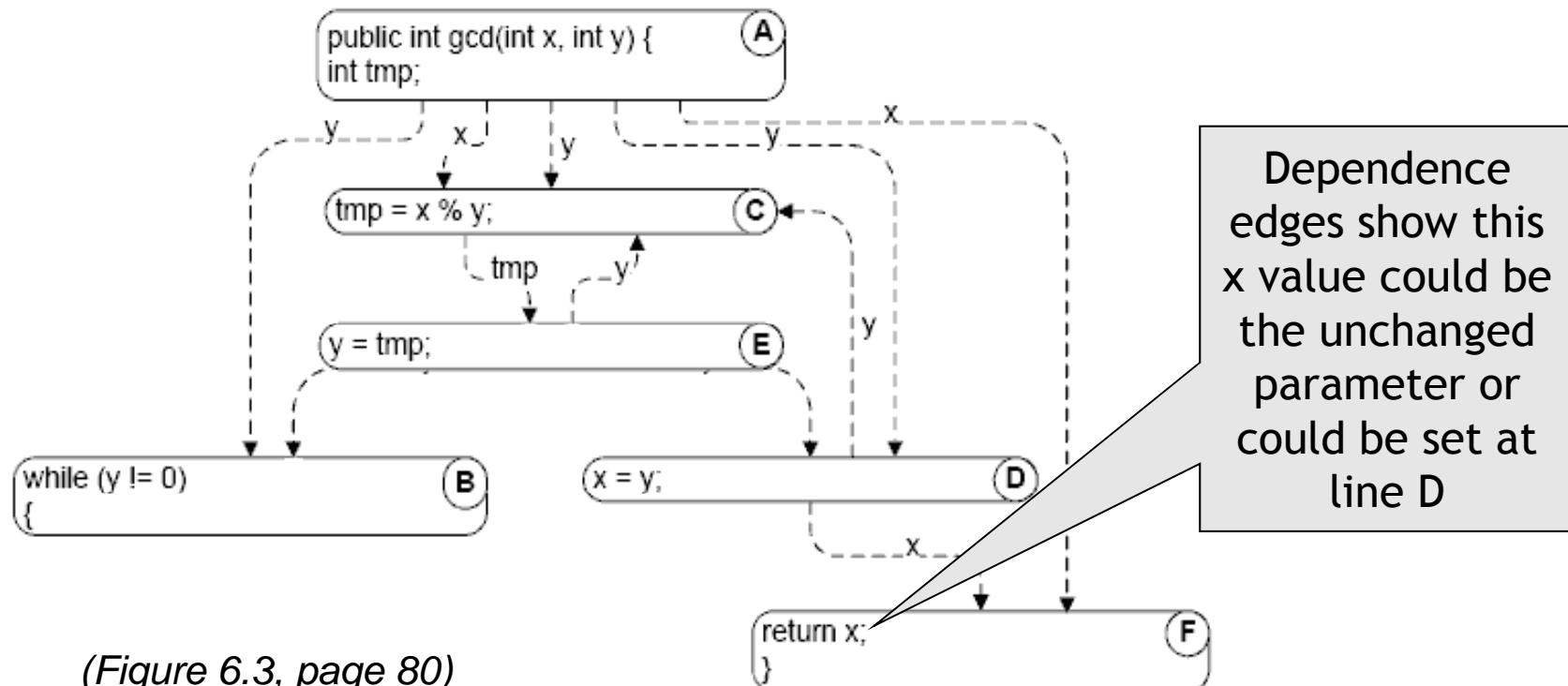
Path A..C is not definition-clear

Path B..C is definition-clear

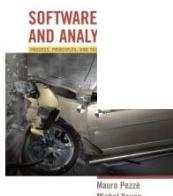


(Direct) Data Dependence Graph

- A direct data dependence graph is:
 - Nodes: as in the control flow graph (CFG)
 - Edges: def-use (du) pairs, labelled with the variable name

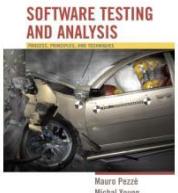


(Figure 6.3, page 80)



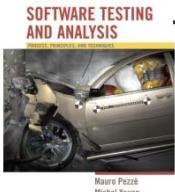
Data Flow Analysis

Computing data flow information

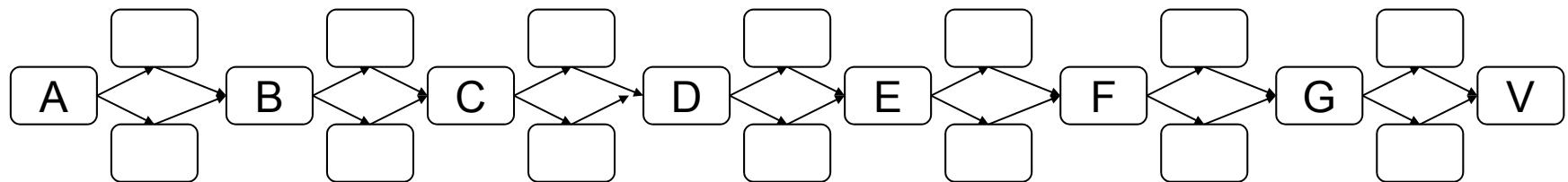


Calculating def-use pairs

- Definition-use pairs can be defined in terms of paths in the program control flow graph:
 - There is an association (d,u) between a definition of variable v at d and a use of variable v at u iff
 - there is at least one control flow path from d to u
 - with no intervening definition of v .
 - v_d reaches u (v_d is a reaching definition at u).
 - If a control flow path passes through another definition e of the same variable v , v_e kills v_d at that point.
- Even if we consider only loop-free paths, the number of paths in a graph can be exponentially larger than the number of nodes and edges.
- Practical algorithms therefore do not search every individual path. Instead, they summarize the reaching definitions at a node over all the paths reaching that node.



Exponential paths (even without loops)



2 paths from A to B

4 from A to C

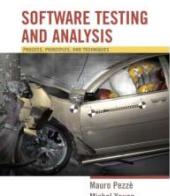
8 from A to D

16 from A to E

...

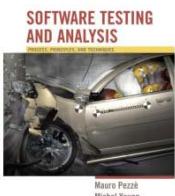
128 paths from A to V

*Tracing each path is
not efficient, and we
can do much better.*



DF Algorithm

- An efficient algorithm for computing reaching definitions (and several other properties) is based on the way reaching definitions at one node are related to the reaching definitions at an adjacent node.
- Suppose we are calculating the reaching definitions of node n, and there is an edge (p,n) from an immediate predecessor node p.
 - If the predecessor node p can assign a value to variable v, then the definition v_p reaches n. We say the definition v_p is generated at p.
 - If a definition v_p of variable v reaches a predecessor node p, and if v is not redefined at that node (in which case we say the v_p is killed at that point), then the definition is propagated on from p to n.



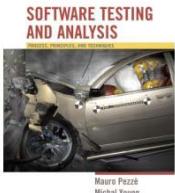
Equations of node E ($y = \text{tmp}$)

Calculate reaching definitions at E in terms of its immediate predecessor D

```
public class GCD {  
    public int gcd(int x, int y) {  
        int tmp;           // A: def x, y, tmp  
        while (y != 0) {   // B: use y  
            tmp = x % y; // C: def tmp; use x, y  
            x = y;         // D: def x; use y  
            y = tmp;        // E: def y; use tmp  
        }  
        return x;          // F: use x  
    }  
}
```

$$\text{Reach}(E) = \text{ReachOut}(D)$$

$$\text{ReachOut}(E) = (\text{Reach}(E) \setminus \{y_A\}) \cup \{y_E\}$$

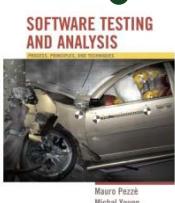


Equations of node B (while ($y \neq 0$))

*This line has two predecessors:
Before the loop,
end of the loop*

```
public class GCD {  
    public int gcd(int x, int y) {  
        int tmp; // A: def x, y, tmp  
        while (y != 0) { // B: use y  
            tmp = x % y; // C: def tmp; use x, y  
            x = y; // D: def x; use y  
            y = tmp; // E: def y; use tmp  
        }  
        return x; // F: use x  
    }  
}
```

- $\text{Reach}(B) = \text{ReachOut}(A) \cup \text{ReachOut}(E)$
- $\text{ReachOut}(A) = \text{gen}(A) = \{x_A, y_A, \text{tmp}_A\}$
- $\text{ReachOut}(E) = (\text{Reach}(E) \setminus \{y_A\}) \cup \{y_E\}$



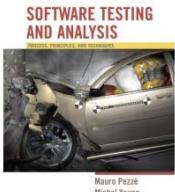
General equations for Reach analysis

$$\text{Reach}(n) = \bigcup_{m \in \text{pred}(n)} \text{ReachOut}(m)$$

$$\text{ReachOut}(n) = (\text{Reach}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$$

$\text{gen}(n) = \{ v_n \mid v \text{ is defined or modified at } n \}$

$\text{kill}(n) = \{ v_x \mid v \text{ is defined or modified at } x, x \neq n \}$



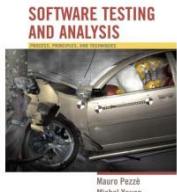
Avail equations

$$\text{Avail}(n) = \bigcap_{m \in \text{pred}(n)} \text{AvailOut}(m)$$

$$\text{AvailOut}(n) = (\text{Avail}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$$

$\text{gen}(n) = \{ \text{exp} \mid \text{exp is computed at } n \}$

$\text{kill}(n) = \{ \text{exp} \mid \text{exp has variables assigned at } n \}$



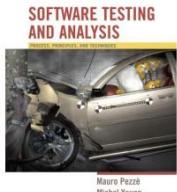
Live variable equations

$$\text{Live}(n) = \bigcup_{m \in \text{succ}(n)} \text{LiveOut}(m)$$

$$\text{LiveOut}(n) = (\text{Live}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$$

$\text{gen}(n) = \{ v \mid v \text{ is used at } n \}$

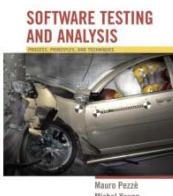
$\text{kill}(n) = \{ v \mid v \text{ is modified at } n \}$



Classification of analyses

- Forward/backward: a node's set depends on that of its predecessors/successors
- Any-path/all-path: a node's set contains a value iff it is coming from any/all of its inputs

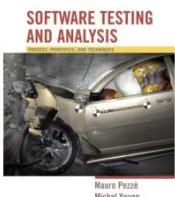
	Any-path (\cup)	All-paths (\cap)
Forward (pred)	Reach	Avail
Backward (succ)	Live	“inevitable”



Iterative Solution of Dataflow Equations

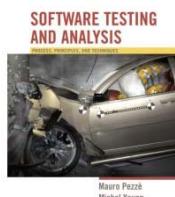
- Initialize values (first estimate of answer)
 - For “any path” problems, first guess is “nothing” (empty set) at each node
 - For “all paths” problems, first guess is “everything” (set of all possible values = union of all “gen” sets)
- Repeat until nothing changes
 - Pick some node and recalculate (new estimate)

This will converge on a “fixed point” solution where every new calculation produces the same value as the previous guess.



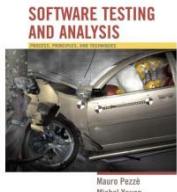
Cooking your own: From Execution to Conservative Flow Analysis

- We can use the same data flow algorithms to approximate other dynamic properties
 - Gen set will be “facts that become true here”
 - Kill set will be “facts that are no longer true here”
 - Flow equations will describe propagation
- Example: Taintedness (in web form processing)
 - “Taint”: a user-supplied value (e.g., from web form) that has not been validated
 - Gen: we get this value from an untrusted source here
 - Kill: we validated to make sure the value is proper



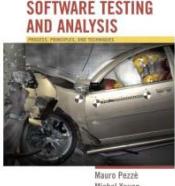
Data flow analysis with arrays and pointers

- Arrays and pointers introduce uncertainty:
Do different expressions access the same storage?
 - $a[i]$ same as $a[k]$ when $i = k$
 - $a[i]$ same as $b[i]$ when $a = b$ (aliasing)
- The uncertainty is accommodated depending to the kind of analysis
 - Any-path: gen sets should include all potential aliases and kill set should include only what is definitely modified
 - All-path: vice versa

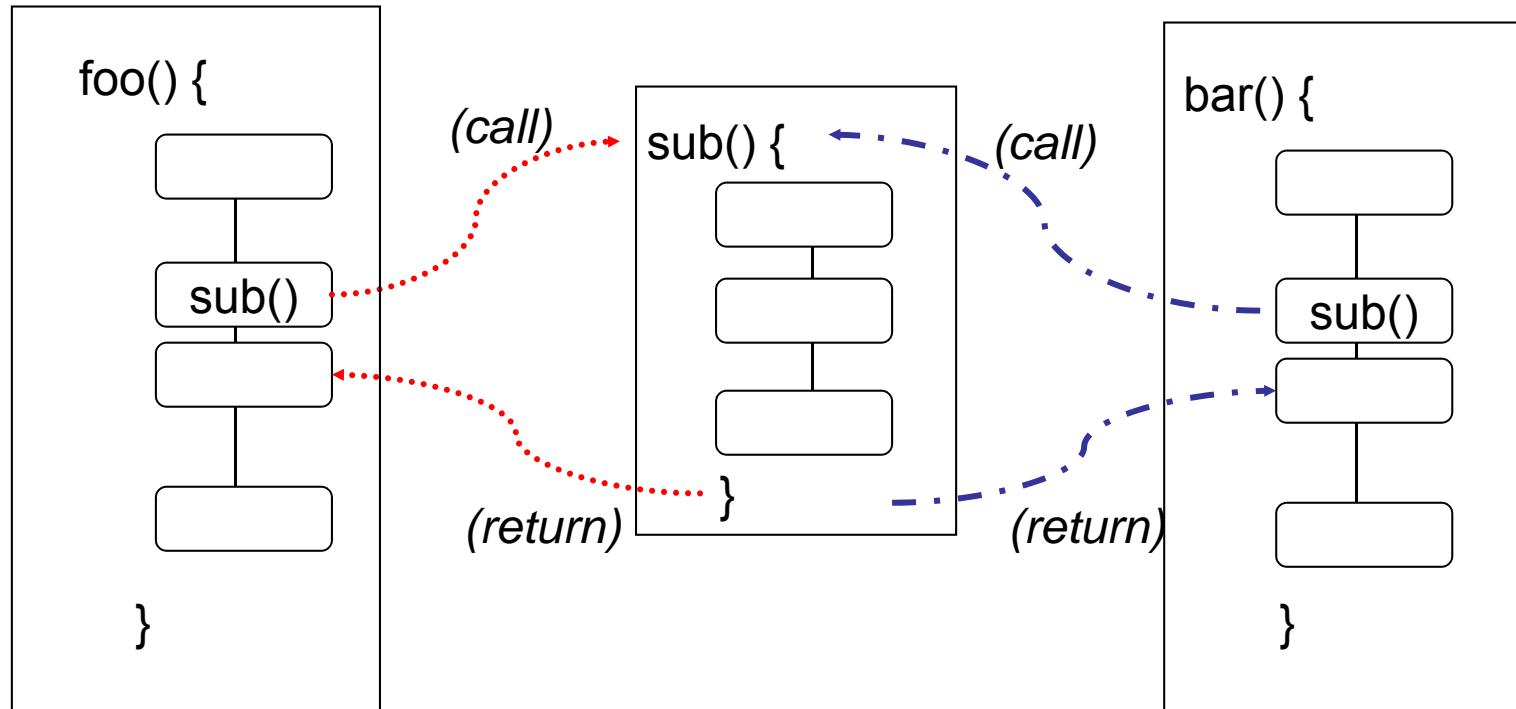


Scope of Data Flow Analysis

- Intraprocedural
 - Within a single method or procedure
 - as described so far
- Interprocedural
 - Across several methods (and classes) or procedures
- Cost/Precision trade-offs for interprocedural analysis are critical, and difficult
 - context sensitivity
 - flow-sensitivity



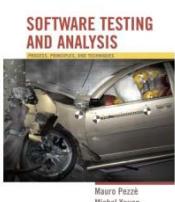
Context Sensitivity



A **context-sensitive** (interprocedural) analysis distinguishes `sub()` called from `foo()` from `sub()` called from `bar()`;
A **context-insensitive** (interprocedural) analysis does not separate them, as if `foo()` could call `sub()` and `sub()` could then return to `bar()`

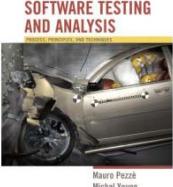
Flow Sensitivity

- Reach, Avail, etc. were **flow-sensitive, intraprocedural analyses**
 - They considered ordering and control flow decisions
 - Within a single procedure or method, this is (fairly) cheap – $O(n^3)$ for n CFG nodes
- Many interprocedural flow analyses are **flow-insensitive**
 - $O(n^3)$ would not be acceptable for all the statements in a program!
 - Though $O(n^3)$ on each individual procedure might be ok
 - Often flow-insensitive analysis is good enough ... consider type checking as an example

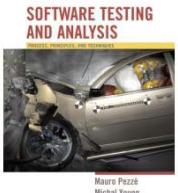


Summary

- Data flow models detect patterns on CFGs:
 - Nodes initiating the pattern
 - Nodes terminating it
 - Nodes that may interrupt it
- Often, but not always, about flow of information (dependence)
- Pros:
 - Can be implemented by efficient iterative algorithms
 - Widely applicable (not just for classic “data flow” properties)
- Limitations:
 - Unable to distinguish feasible from infeasible paths
 - Analyses spanning whole programs (e.g., alias analysis) must trade off precision against computational cost



Data flow testing

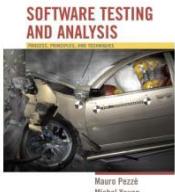


(c) 2007 Mauro Pezzè & Michal Young

Ch 13, slide 1

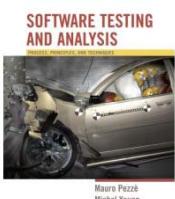
Learning objectives

- Understand why data flow criteria have been designed and used
- Recognize and distinguish basic DF criteria
 - All DU pairs, all DU paths, all definitions
- Understand how the infeasibility problem impacts data flow testing
- Appreciate limits and potential practical uses of data flow testing

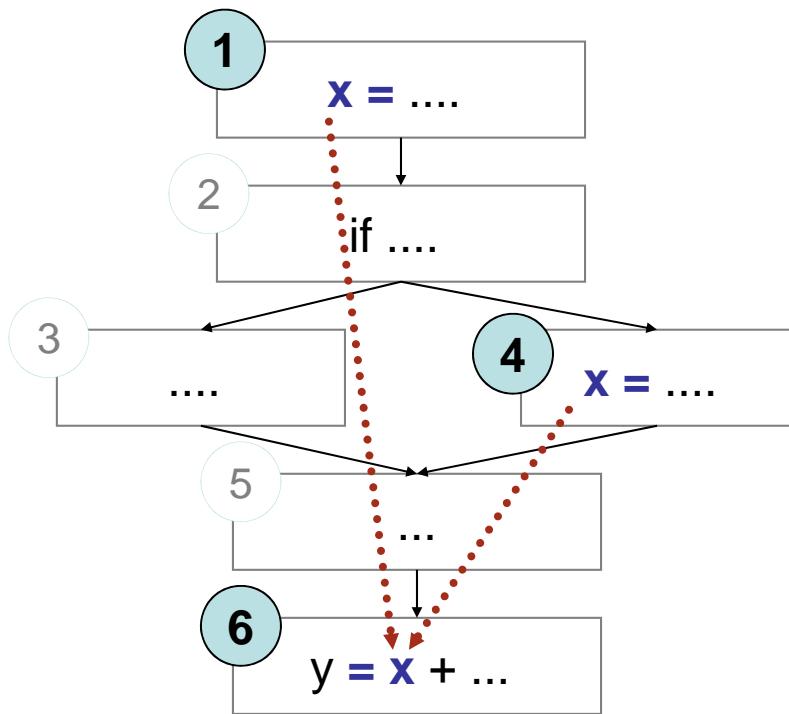


Motivation

- Middle ground in structural testing
 - Node and edge coverage don't test interactions
 - Path-based criteria require impractical number of test cases
 - And only a few paths uncover additional faults, anyway
 - Need to distinguish "important" paths
- Intuition: Statements interact through *data flow*
 - Value computed in one statement, used in another
 - Bad value computation revealed only when it is used



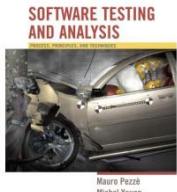
Data flow concept



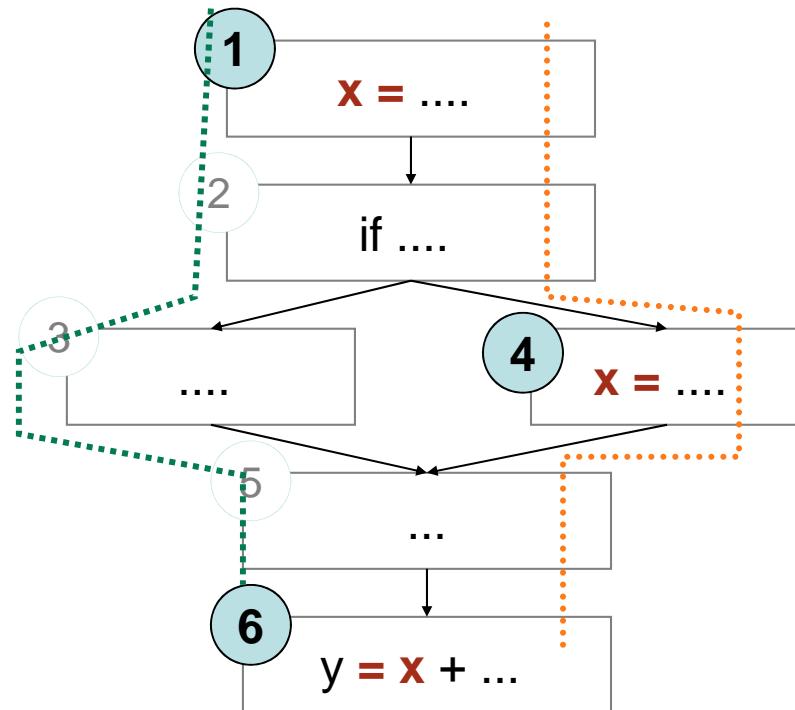
- Value of x at 6 could be computed at 1 or at 4
- Bad computation at 1 or 4 could be revealed only if they are used at 6
- (1,6) and (4,6) are *def-use (DU) pairs*
 - defs at 1,4
 - use at 6

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 = ... is a *definition* of x
 - = ... x ... 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



Definition-clear path

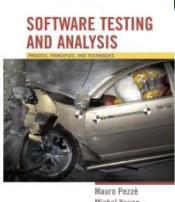


- $1,2,3,5,6$ is a definition-clear path from 1 to 6
 - x is not re-assigned between 1 and 6
- $1,2,4,5,6$ is not a definition-clear path from 1 to 6
 - the value of x is “killed” (reassigned) at node 4
- $(1,6)$ is a DU pair because $1,2,3,5,6$ is a definition-clear path

Adequacy criteria

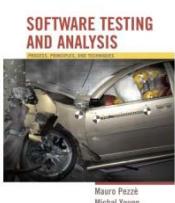
- All DU pairs: Each DU pair is exercised by at least one test case
- All DU paths: Each *simple* (non looping) DU path is exercised by at least one test case
- All definitions: For each definition, there is at least one test case which exercises a DU pair containing it
 - (Every computed value is used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

Corresponding coverage fractions can also be defined



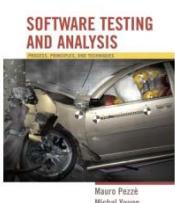
Difficult cases

- $x[i] = \dots ; \dots ; y = x[j]$
 - DU pair (only) if $i==j$
- $p = &x ; \dots ; *p = 99 ; \dots ; q = x$
 - $*p$ is an alias of x
- $m.putFoo(...); \dots ; y=n.getFoo(...);$
 - Are m and n the same object?
 - Do m and n share a “foo” field?
- Problem of *aliases*: Which references are (always or sometimes) the same?

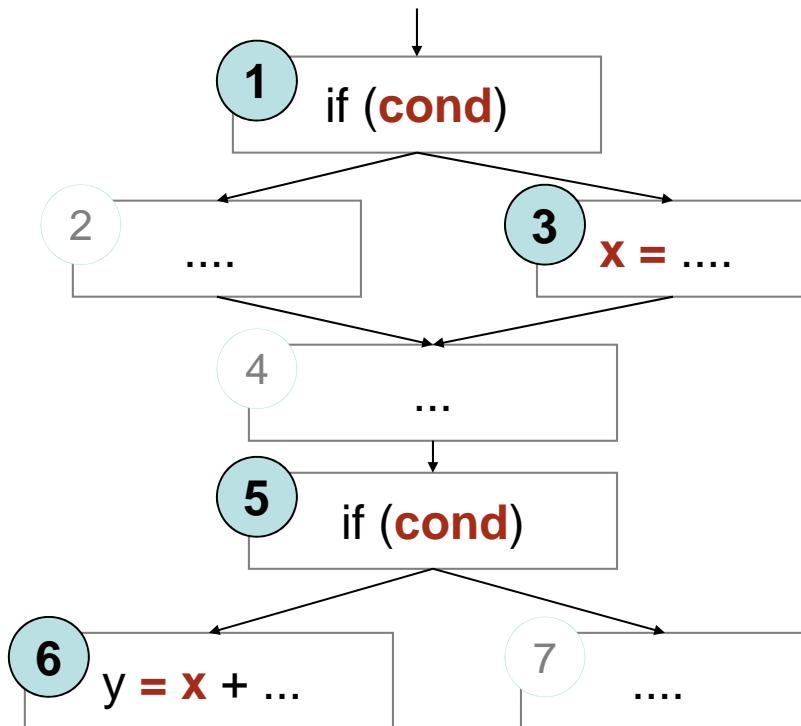


Data flow coverage with complex structures

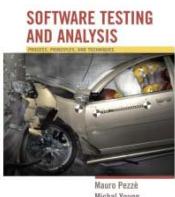
- Arrays and pointers are critical for data flow analysis
 - Under-estimation of aliases may fail to include some DU pairs
 - Over-estimation, on the other hand, may introduce unfeasible test obligations
- For testing, it may be preferable to accept under-estimation of alias set rather than over-estimation or expensive analysis
 - Controversial: In other applications (e.g., compilers), a *conservative* over-estimation of aliases is usually required
 - Alias analysis may rely on external guidance or other global analysis to calculate good estimates
 - Undisciplined use of dynamic storage, pointer arithmetic, etc. may make the whole analysis infeasible



Infeasibility

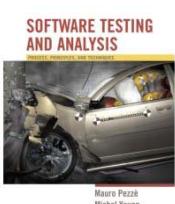


- Suppose *cond* has not changed between 1 and 5
 - Or the conditions could be different, but the first implies the second
- Then (3,5) is not a (feasible) DU pair
 - But it is difficult or impossible to determine which pairs are infeasible
- Infeasible test obligations are a problem
 - No test case can cover them



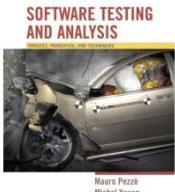
Infeasibility

- The path-oriented nature of data flow analysis makes the infeasibility problem especially relevant
 - Combinations of elements matter!
 - Impossible to (infallibly) distinguish feasible from infeasible paths. More paths = more work to check manually.
- In practice, reasonable coverage is (often, not always) achievable
 - Number of paths is exponential in worst case, but often linear
 - All DU *paths* is more often impractical

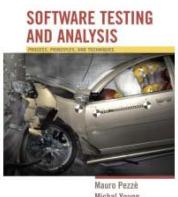


Summary

- Data flow testing attempts to distinguish “important” paths: Interactions between statements
 - Intermediate between simple statement and branch coverage and more expensive path-based structural testing
- Cover Def-Use (DU) pairs: From computation of value to its use
 - Intuition: Bad computed value is revealed only when it is used
 - Levels: All DU pairs, all DU paths, all defs (some use)
- Limits: Aliases, infeasible paths
 - Worst case is bad (undecidable properties, exponential blowup of paths), so pragmatic compromises are required



Test Case Selection and Adequacy Criteria

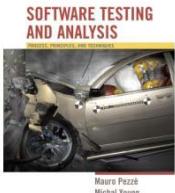


(c) 2007 Mauro Pezzè & Michal Young

Ch 9, slide 1

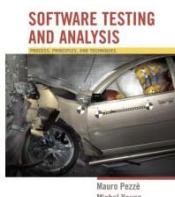
Learning objectives

- Understand the purpose of defining test adequacy criteria, and their limitations
- Understand basic terminology of test selection and adequacy
- Know some sources of information commonly used to define adequacy criteria
- Understand how test selection and adequacy criteria are used



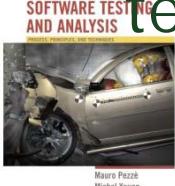
Adequacy: We can't get what we want

- What we would like:
 - A real way of measuring effective testing
If the system passes an adequate suite of test cases, then it must be correct (or dependable)
- But that's impossible!
 - Adequacy of test suites, in the sense above, is provably undecidable.
- So we'll have to settle on weaker proxies for adequacy
 - Design rules to highlight inadequacy of test suites



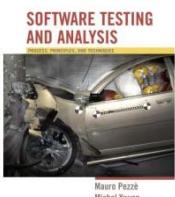
Practical (in)Adequacy Criteria

- Criteria that identify inadequacies in test suites.
 - Examples
 - *if the specification describes different treatment in two cases, but the test suite does not check that the two cases are in fact treated differently, we may conclude that the test suite is inadequate to guard against faults in the program logic.*
 - *If no test in the test suite executes a particular program statement, the test suite is inadequate to guard against faults in that statement.*
- If a test suite fails to satisfy some criterion, the obligation that has not been satisfied may provide some useful information about improving the test suite.
- If a test suite satisfies all the obligations by all the criteria, we do not know definitively that it is an effective test suite, but we have some evidence of its thoroughness.



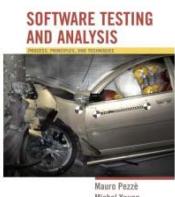
Some useful terminology

- **Test case:** a set of inputs, execution conditions, and a pass/fail criterion.
- **Test case specification:** a requirement to be satisfied by one or more test cases.
- **Test obligation:** a partial test case specification, requiring some property deemed important to thorough testing.
- **Test suite:** a set of test cases.
- **Test or test execution:** the activity of executing test cases and evaluating their results.
- **Adequacy criterion:** a predicate that is true (satisfied) or false (not satisfied) of a ⟨program, test suite⟩ pair.



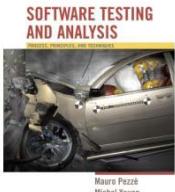
Where do test obligations come from?

- Functional (black box, specification-based): from software specifications
 - Example: If spec requires robust recovery from power failure, test obligations should include simulated power failure
- Structural (white or glass box): from code
 - Example: Traverse each program loop one or more times.
- Model-based: from model of system
 - Models used in specification or design, or derived from code
 - Example: Exercise all transitions in communication protocol model
- Fault-based: from hypothesized faults (common bugs)
 - Example: Check for buffer overflow handling (common vulnerability) by testing on very large inputs



Adequacy criteria

- Adequacy criterion = set of test obligations
- A test suite satisfies an adequacy criterion if
 - all the tests succeed (pass)
 - every test obligation in the criterion is satisfied by at least one of the test cases in the test suite.
 - Example:
the statement coverage adequacy criterion is satisfied by test suite S for program P if each executable statement in P is executed by at least one test case in S, and the outcome of each test execution was “pass”.

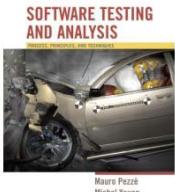


Satisfiability

- Sometimes *no* test suite can satisfy a criterion for a given program
 - Example: Defensive programming style includes “can’t happen” sanity checks

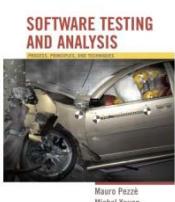
```
if (z < 0) {  
    throw new LogicError(  
        "z must be positive here!")  
}
```

No test suite can satisfy statement coverage for this program (if it's correct)



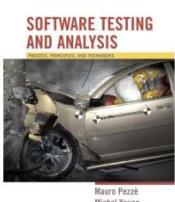
Coping with Unsatisfiability

- Approach A: exclude any unsatisfiable obligation from the criterion.
 - Example: modify statement coverage to require execution only of statements that can be executed.
 - But we can't know for sure which are executable!
- Approach B: measure the extent to which a test suite approaches an adequacy criterion.
 - Example: if a test suite satisfies 85 of 100 obligations, we have reached 85% *coverage*.
 - Terms: An adequacy criterion is satisfied or not, a coverage measure is the fraction of satisfied obligations



Comparing Criteria

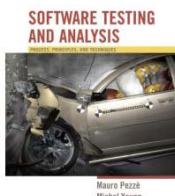
- Can we distinguish stronger from weaker adequacy criteria?
- Empirical approach: Study the effectiveness of different approaches to testing in industrial practice
 - What we really care about, but ...
 - Depends on the setting; may not generalize from one organization or project to another
- Analytical approach: Describe conditions under which one adequacy criterion is provably stronger than another
 - Stronger = gives stronger guarantees
 - One piece of the overall “effectiveness” question



The *subsumes* relation

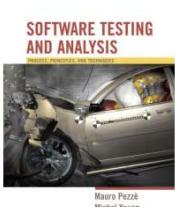
Test adequacy criterion A subsumes test adequacy criterion B iff, for every program P, every test suite satisfying A with respect to P also satisfies B with respect to P.

- Example:
 - Exercising all program branches (branch coverage) *subsumes* exercising all program statements
- A common analytical comparison of closely related criteria
 - Useful for working from easier to harder levels of coverage, but not a direct indication of quality



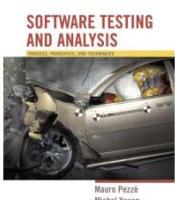
Uses of Adequacy Criteria

- Test selection approaches
 - Guidance in devising a thorough test suite
 - Example: A specification-based criterion may suggest test cases covering representative combinations of values
- Revealing missing tests
 - Post hoc analysis: What might I have missed with this test suite?
- Often in combination
 - Example: Design test suite from specifications, then use structural criterion (e.g., coverage of all branches) to highlight missed logic



Summary

- Adequacy criteria provide a way to define a notion of “thoroughness” in a test suite
 - But they don’t offer guarantees; more like *design rules to highlight inadequacy*
- Defined in terms of “covering” some information
 - Derived from many sources: Specs, code, models, ...
- May be used for selection as well as measurement
 - With caution! An aid to thoughtful test design, not a substitute



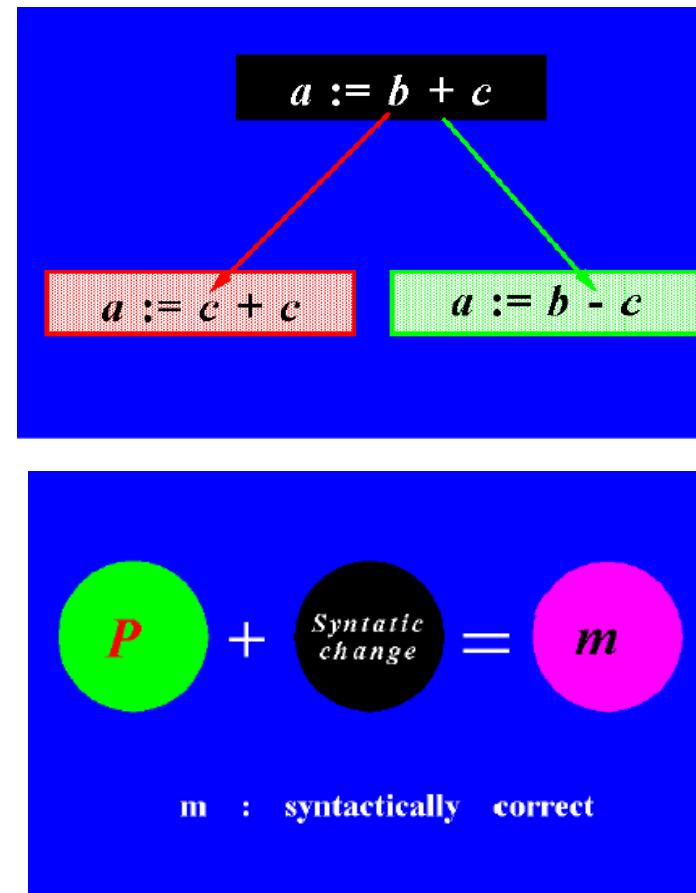
Software Verification and Validation

Prof. Lionel Briand
Ph.D., IEEE Fellow

Mutation Testing

Definitions

- *Fault-based Testing*: directed towards "typical" faults that could occur in a program
- Basic idea:
 - Take a program and test data generated for that program
 - Create a number of *similar* programs (mutants), each differing from the original in one small way, i.e., each possessing a fault
 - e.g., replace addition operator by multiplication operator
 - The original test data are then run through the *mutants*
 - If test data detect all differences in mutants, then the mutants are said to be *dead*, and the test set is *adequate*



Different types of Mutants

- Stillborn mutants: Syntactically incorrect, killed by compiler, e.g., $x = a ++ b$
- Trivial mutants: Killed by almost any test case
- Equivalent mutant: Always acts in the same behavior as the original program, e.g., $x = a + b$ and $x = a - (-b)$
- None of the above are interesting from a mutation testing perspective
- Those mutants are interesting which behave differently than the original program, and we do not have test cases to identify them (to cover those specific changes)

Example of an Equivalent mutant

Original program

```
int index=0;  
while (...) {  
    . . .;  
    index++;  
    if (index==10)  
        break;  
}
```

A mutant

```
int index=0;  
while (...) {  
    . . .;  
    index++;  
    if (index>=10)  
        break;  
}
```

Basic Ideas (I)

In Mutation Testing:

1. We take a program and a test suite generated for that program (using other test techniques)
2. We create a number of *similar* programs (mutants), each differing from the original in one small way, i.e., each possessing a fault
 - E.g., replacing an addition operator by a multiplication operator
3. The original test data are then run on the *mutants*
4. If test cases detect differences in mutants, then the mutants are said to be *dead (killed)*, and the test set is considered *adequate*

Basic Ideas (II)

- A mutant remains *live* either
 - because it is equivalent to the original program (functionally identical although syntactically different - called an *equivalent mutant*) or,
 - the test set is inadequate to kill the mutant
- In the latter case, the test data need to be augmented (by adding one or more new test cases) to kill the *live* mutant
- For the automated generation of mutants, we use *mutation operators*, that is predefined program modification rules (i.e., corresponding to a fault model)
- Example mutation operators next...

A Simple Example

Original Function	With Embedded Mutants
<pre>int Min (int A, int B) int minVal; { minVal = A; if (B < A) { minVal = B; } return (minVal); } // end Min</pre>	<pre>int Min (int A, int B) int minVal; { minVal = A; Δ1 minVal = B; if (B < A) Δ2 if (B > A) Δ3 if (B < minVal) { minVal = B; Δ4 Bomb(); Δ5 minVal = A; Δ6 minVal = failOnZero (B); } return (minVal); } // end Min</pre>

Delta's represent syntactic modifications. In fact, each of them will be embedded in a different program version, a mutant.

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

Example of Mutation Operators II

Specific to object-oriented programming languages:

- Replacing a type with a compatible subtype (inheritance)
- Changing the access modifier of an attribute, a method
- Changing the instance creation expression (inheritance)
- Changing the order of parameters in the definition of a method
- Changing the order of parameters in a call
- Removing an overloading method
- Reducing the number of parameters
- Removing an overriding method
- Removing a hiding Field
- Adding a hiding field

Specifying Mutations Operators

- Ideally, we would like the mutation operators to be representative of (and generate) all realistic types of faults that could occur in practice.
- Mutation operators change with programming languages, design and specification paradigms, though there is much overlap.
- In general, the number of mutation operators is large as they are supposed to capture all possible *syntactic* variations in a program.
- Recent paper suggests random sampling of mutants can be used.
- Some recent studies seem to suggest that mutants are good indicators of test effectiveness (Andrews et al, ICSE 2005).

Mutation Coverage

- Complete coverage equals to killing all non-equivalent mutants (or random sample)
- The amount of coverage is also called "mutation score"
- We can see each mutant as a test requirement
- The number of mutants depends on the definition of mutation operators and the syntax/structure of the software
- Numbers of mutants tend to be large, even for small programs (hence random sampling)

A Simple Example (again)

Original Function	With Embedded Mutants
<pre>int Min (int A, int B) int minVal; { minVal = A; if (B < A) { minVal = B; } return (minVal); } // end Min</pre>	<pre>int Min (int A, int B) int minVal; { minVal = A; Δ1 minVal = B; if (B < A) Δ2 if (B > A) Δ3 if (B < minVal) { minVal = B; Δ4 Bomb(); Δ5 minVal = A; Δ6 minVal = failOnZero (B); } return (minVal); } // end Min</pre>

Delta's represent syntactic modifications. In fact, each of them will be embedded in a different program version, a mutant.

Discussion of the Example

- Mutant 3 is equivalent as, at this point, `minVal` and `A` have the same value
- Mutant 1: In order to find an appropriate test case to kill it, we must
 1. Reach the fault seeded during execution (Reachability)
 - Always true (i.e., we can always reach the seeded fault)
 1. Cause the program state to be incorrect (Infection)
 - $A \leftrightarrow B$
 3. Cause the program output and/or behavior to be Incorrect (Propagation)
 - $(B < A) = \text{false}$

Original Function	With Embedded Mutants
<pre>int Min (int A, int B) int minVal; { minVal = A; if (B < A) { minVal = B; } return (minVal); } // end Min</pre>	<pre>int Min (int A, int B) int minVal; { minVal = A; Δ1 minVal = B; if (B < A) { minVal = B; Δ2 if (B > A) Δ3 if (B < minVal) { minVal = B; Δ4 Bomb(); Δ5 minVal = A; Δ6 minVal = failOnZero (B); } return (minVal); } // end Min</pre>

Assumptions

- What about more complex errors, involving several statements?
- Let's discuss two assumptions:
 - Competent programmer assumption: They write programs that are nearly correct
 - Coupling effect assumption: Test cases that distinguish all programs differing from a correct one by only simple errors is so sensitive that they also implicitly distinguish more complex errors
- There is some empirical evidence of the above two hypotheses: Offutt, A.J., *Investigations of the Software Testing Coupling Effect*, ACM Transactions on Software Engineering and Methodology, vol. 1 (1), pp. 3-18, 1992.

Another Example

Specification:

- The program should prompt the user for a positive integer in the range 1 to 20 and then for a string of that length.
- The program then prompts for a character and returns the position in the string at which the character was first found or a message indicating that the character was not present in the string.

Code Chunk

```
...
found := FALSE;
i := 1;
while(not(found)) and (i <= x) do begin // x is the length
  if a[i] = c then
    found := TRUE
  else
    i := i + 1
end
if (found)
  print("Character %c appears at position %i");
else
  print("Character is not present in the string");
end
...
```

Mutation Testing Example: Test Set 1

Input				Expected Output (oracle)
x	a[]	c	Response	
25				The input integer should be between 1 and 20
1	x	x	found	Character x appears at position 1
1	x	a	not found	Character is not present in the string

Mutation Testing Example: Mutant 1 (for Test Set 1)

- Replace `Found := FALSE;` with `Found := TRUE;`
- Re-run original test data set
- Note: It is better in Mutation Testing to make only one small change at a time to avoid the danger of introduced faults with interfering effects (masking)
- Failure: "character *a* appears at position 1" instead of saying "character is not present in the string"
- Mutant 1 is killed (since Output <> Oracle)

```

...
found := FALSE;  TRUE;
i := 1;
while(not(found)) and (i <= x) do begin
    if a[i] = c then
        found := TRUE
    else
        i := i + 1
end
if (found)
    print("Character %c appears at position %i");
else
    print("Character is not present in the string");
end
...

```

Mutation Testing Example: Mutant 2 (for Test Set 1)

- Replace `i := 1;` with `x := 1;`

```

Int i=1;
...
found := FALSE;
i := 1; x := 1;
while(not(found)) and (i <= x) do begin
    if a[i] = c then
        found := TRUE
    else
        i := i + 1
end
if (found)
    print("Character %c appears at position %i");
else
    print("Character is not present in the string");
end
...

```

- Will our original test data (test set 1) reveal the fault?
 - No, our original test data set fails to reveal the fault (because the `x` value was 1 in the second test case of test set 1)
- As a result of the fault, only position 1 in string will be searched for. So what should we do?
- In our test set, we need to increase our input string length and search for a character further along it
- We modify the test set 1 and create a new test set 2 (next) so as
 - To preserve the effect of earlier tests
 - To make sure the live mutant (#2) is killed

Mutation Testing

Example: Test Set 2

Input				Expected Output
x	a	c	Actual output Response	
25				Input Integer between 1 and 20
1	x	x	found	Character x appears at position 1
1	x	a	not found	Character does not occur in string
3	xCv	v	Not found	Character v appears at position 3 (this test case will kill the mutant in the previous slide)

Mutation Testing Example: Mutant 3 (for Test Set 2)

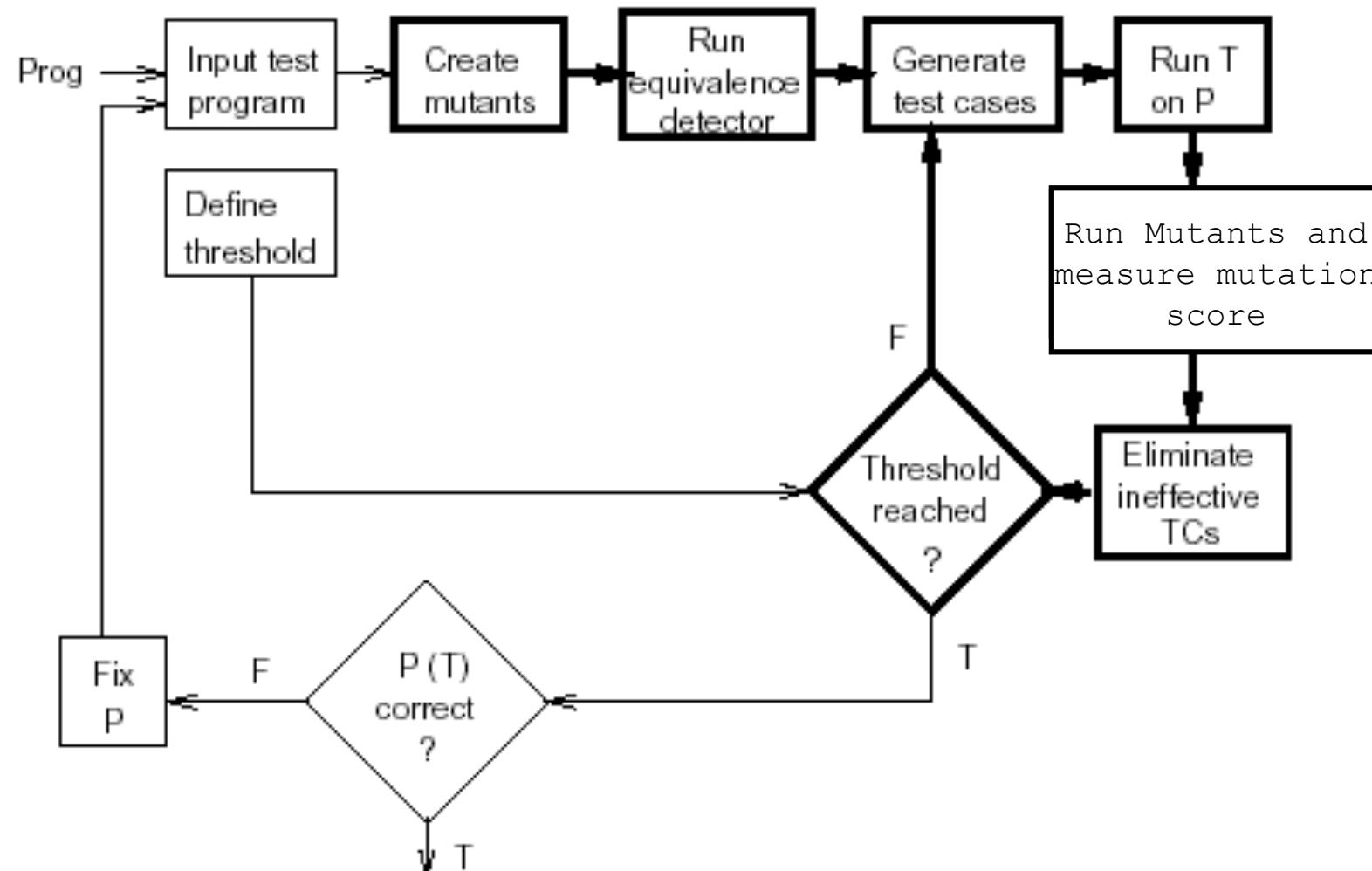
- `i := i + 1;` is replaced with `i := i + 2;`
- Again, our test data (test set 2) fails to kill the mutant
- We must augment the test set 2 and create a new test set 3 (next) to search for a character in the middle of the string
- With the new test set, mutant 3 can be killed
- Many other changes could be made on this short piece of code, e.g., changing array reference, changing the `<=` relational operator

```
...
found := FALSE;
i := 1;
while(not(found)) and (i <= x) do begin
    if a[i] = c then
        found := TRUE
    else
        i := i + 1 - 2
end
if (found)
    print("Character %c appears at position %i");
else
    print("Character is not present in the string");
end
...
```

Mutation Testing Example: Test Set 3

Input				Expected Output
x	a	c	Response	
25				Input Integer between 1 and 20
1	x	x	found	Character x appears at position 1
1	x	a	not found	Character does not occur in string
3	xCv	v	Found	Character v appears at position 3
3	xCv	C	Not found	Character C appears at position 2 (this test case will kill the mutant in the previous slide)

Mutation Testing Process



Mutation Testing: Discussion

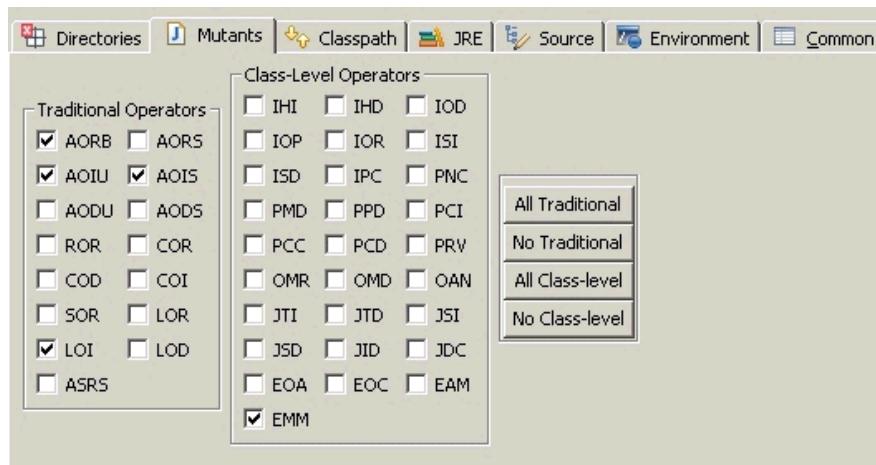
- It measures the quality of test cases
- A tool's slogan: "Jester - the JUnit test tester".
- It provides the tester with a clear target (mutants to kill)
- Mutation testing can also show that certain kinds of faults are unlikely (those specified by the fault model), since the corresponding test case will not fail
- It does force the programmer to inspect the code and think of the test data that will expose certain kinds of faults
- It is computationally intensive, a possibly very large number of mutants is generated: random sampling, selective mutation operators (Offutt)
- Equivalent mutants are a practical problem: It is in general an undecidable problem
- Probably most useful at unit testing level

Mutation Testing: Other Applications

- Mutation operators and systems are also very useful for assessing the effectiveness of test strategies - they have been used in a number of case studies
 - Define a set of realistic mutation operators
 - Generate mutants (automatically)
 - Generate test cases according to alternative strategies
 - Assess the *mutation score* (percentage of mutants killed)
- In our discussion, we saw mutation operators for source code (body)
- There are also works on
 - Mutation operators for module interfaces (aimed at integration testing)
 - Mutation operators on specifications: Petri-nets, state machines, ... (aimed at system testing)

Mutation Testing Tools and Some Key Pointers

- Tools
 - **MuClipse: perhaps the best tool out there...**



- **Jester: A Mutation Testing tool in Java (Open Source)**
- **Pester: A Mutation Testing tool in Python (Open Source)**
- **Nester: A Mutation Testing tool in C# (Open Source)**
- <http://www.parasoft.com/jsp/products/article.jsp?articleId=291>

- Pointers:
 - http://en.wikipedia.org/wiki/Mutation_testing
 - <http://www.mutationtest.net/>
 - <http://www.dcs.kcl.ac.uk/pg/jiayue/repository/>

What is TDD

WHAT IS TDD

- Test-Driven Development (or test driven design) is a methodology.

Common TDD misconception:

- TDD is not about testing
- TDD is about design and development
- By testing first you design your code

WHAT IS TDD

- Short development iterations.
- Based on requirement and pre-written test cases.
- Produces code necessary to pass that iteration's test.
- Refactor both code and tests.
- The goal is to produce working clean code that fulfills requirements.

Principle of TDD

Kent Beck defines:

- Never write a single line of code unless you have a failing automated test.
- Eliminate duplication.

Red (Automated test fail)

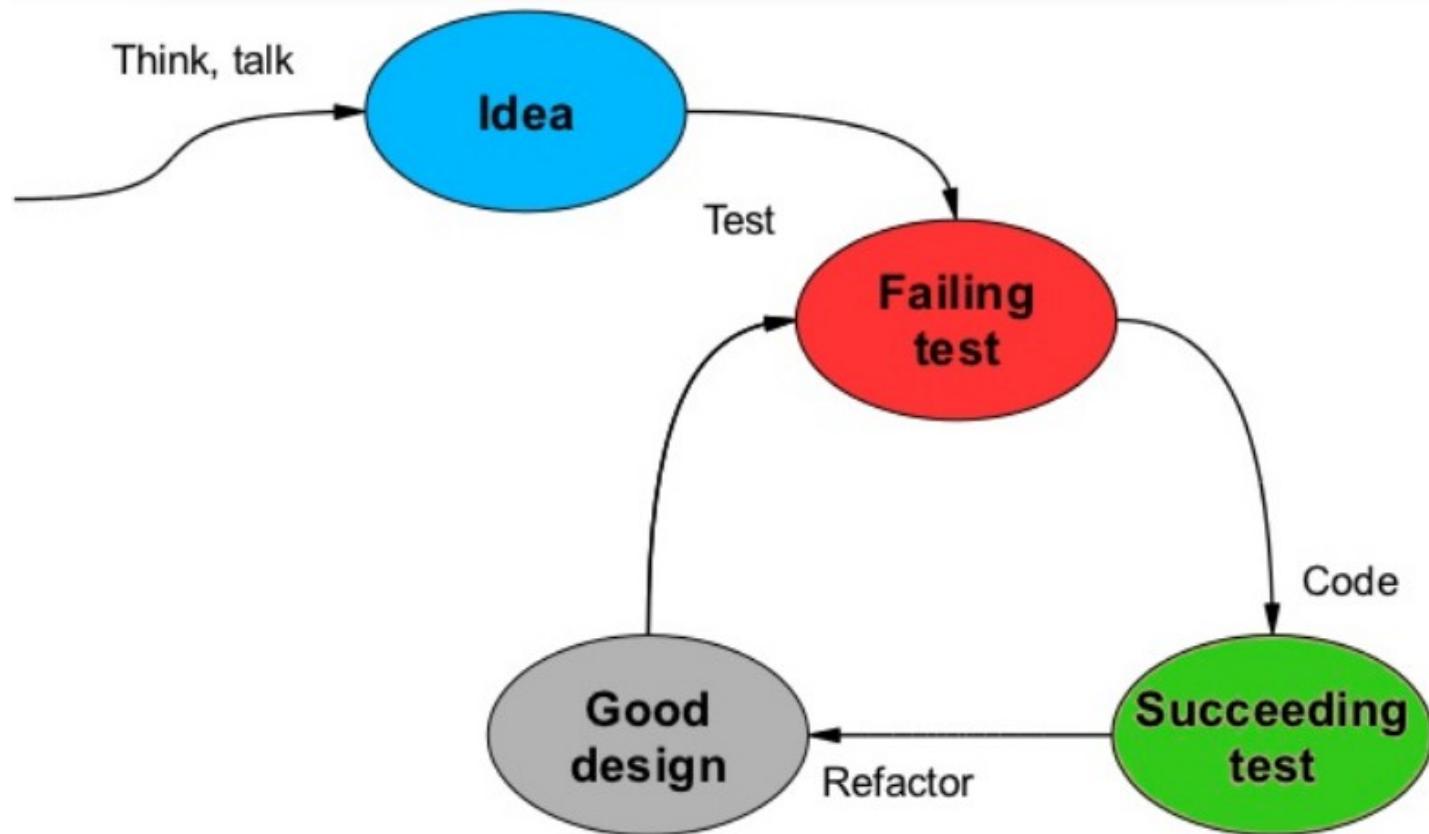
Green (Automated test pass because dev code has been written)

Refactor (Eliminate duplication, Clean the code)

TDD BASICS - UNIT TESTING

- Red, Green, Refactor
- Make it Fail
 - No code without a failing test
- Make it Work
 - As simply as possible
- Make it Better
 - Refactor

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

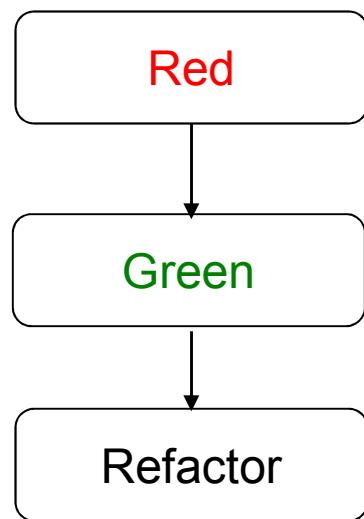
□ Write Functional Code

- 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

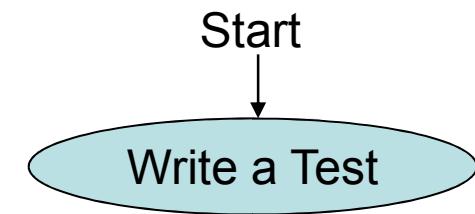
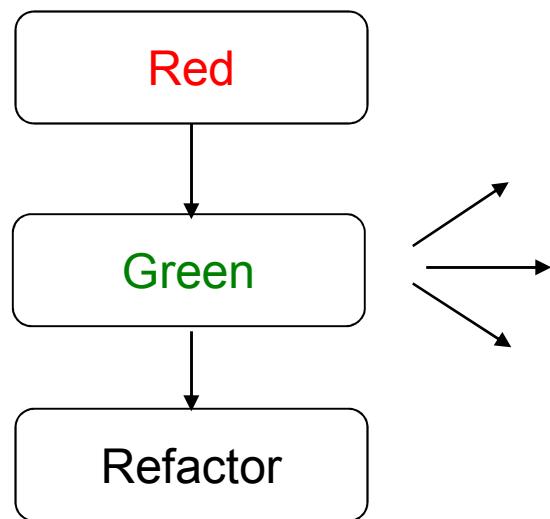
- Clean-up the code (test and functional)
- Make sure the code expresses intent
- Remove code smells
- Re-think the design
- Delete unnecessary code

Principle of TDD (In Practice)



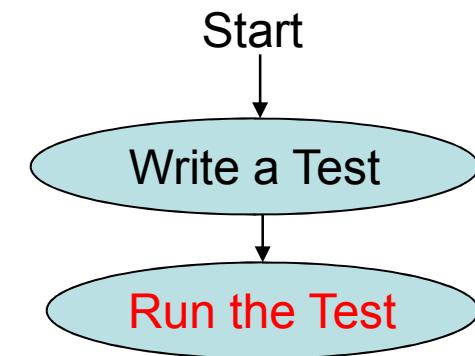
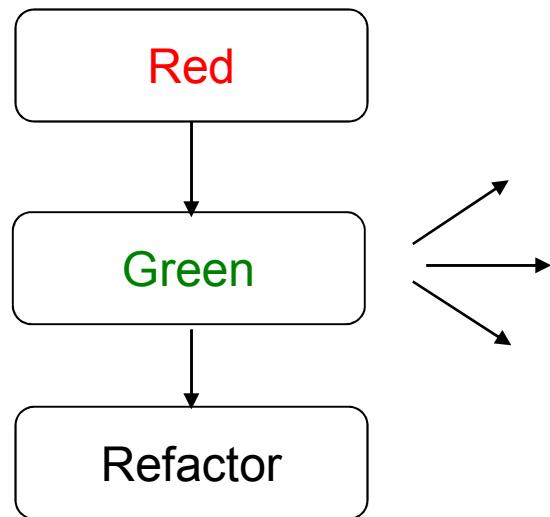
TDD

Principle of TDD (In Practice)



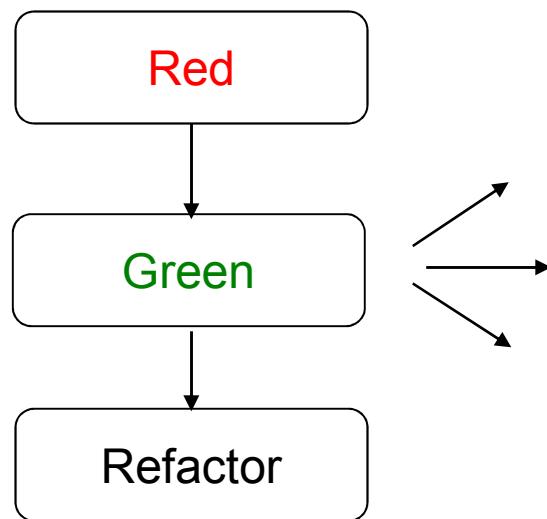
TDD

Principle of TDD (In Practice)

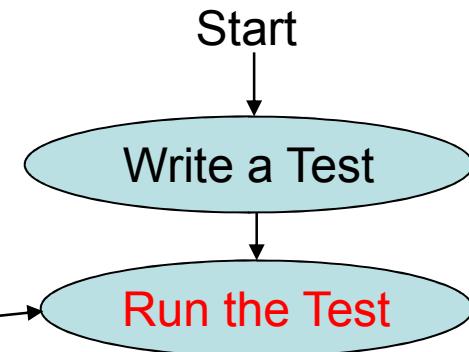


TDD

Principle of TDD (In Practice)

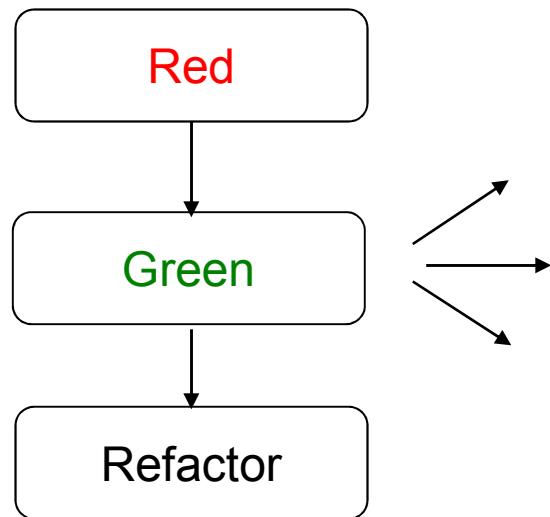


See it fail
because there's
no dev code

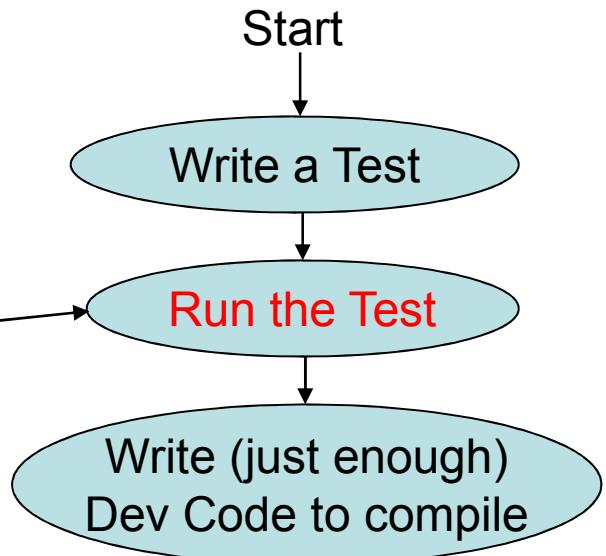


TDD

Principle of TDD (In Practice)

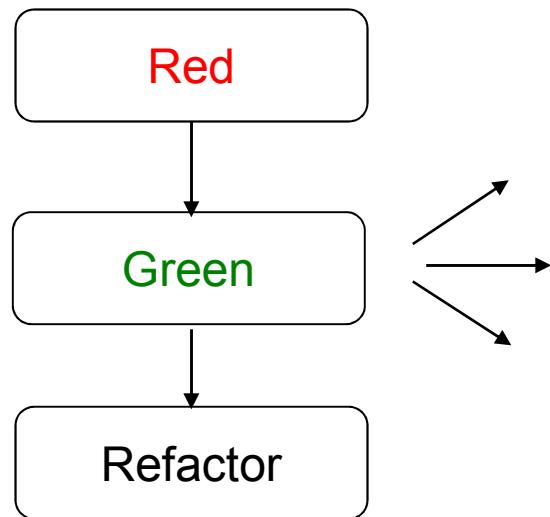


See it fail
because there's
no dev code

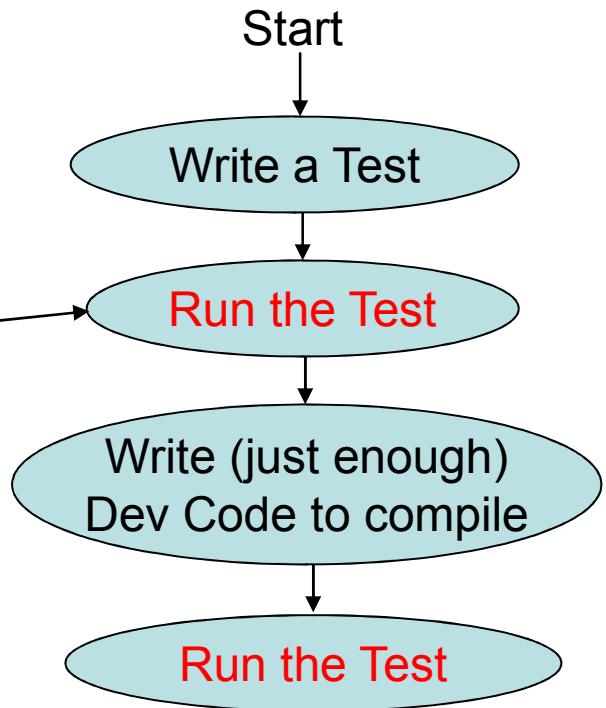


TDD

Principle of TDD (In Practice)

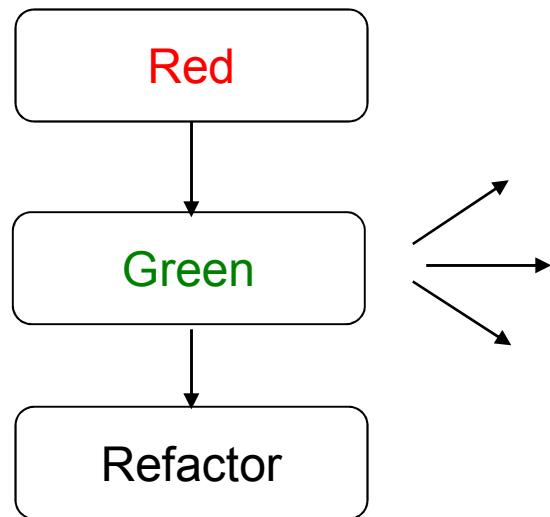


See it fail
because there's
no dev code



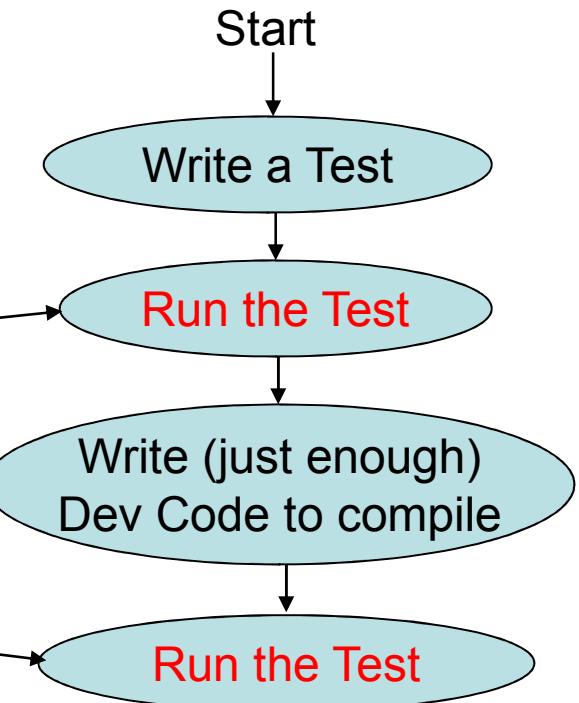
TDD

Principle of TDD (In Practice)



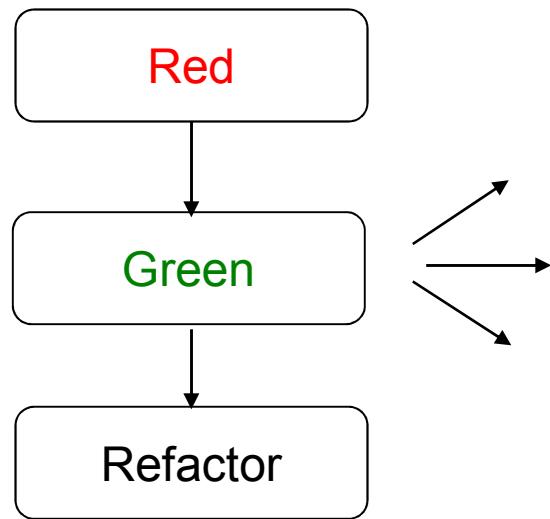
See it fail
because there's
no dev code

See it fail
because no logic
is implemented



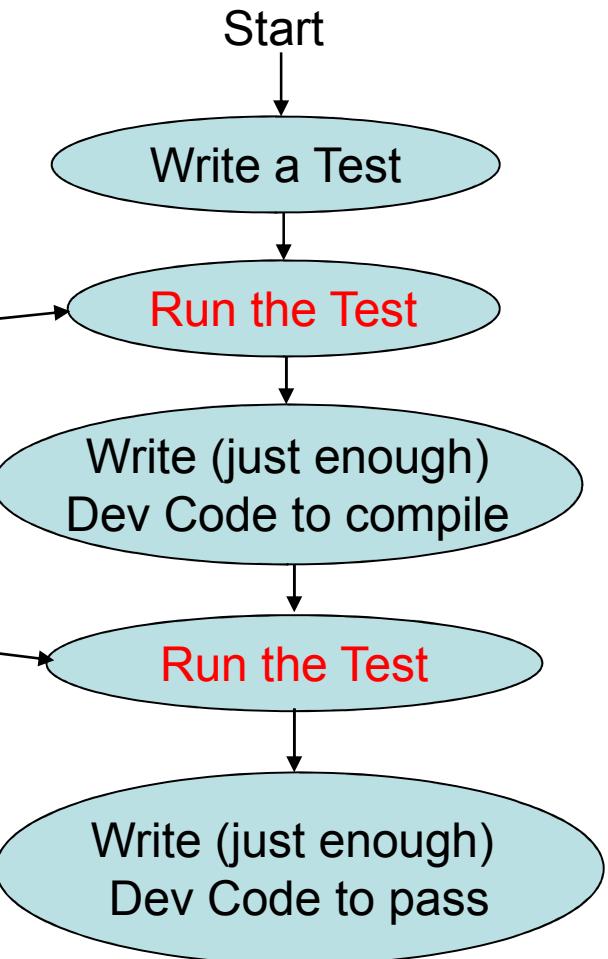
TDD

Principle of TDD (In Practice)



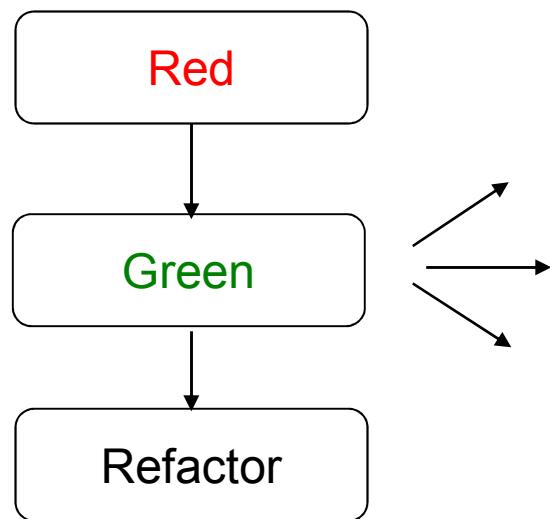
See it fail
because there's
no dev code

See it fail
because no logic
is implemented



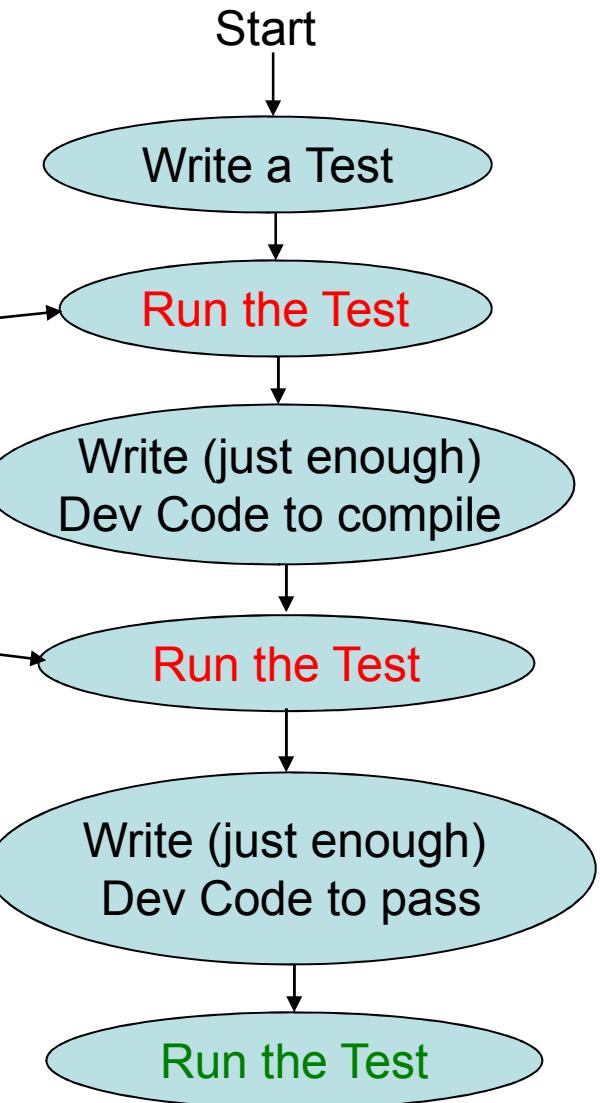
TDD

Principle of TDD (In Practice)



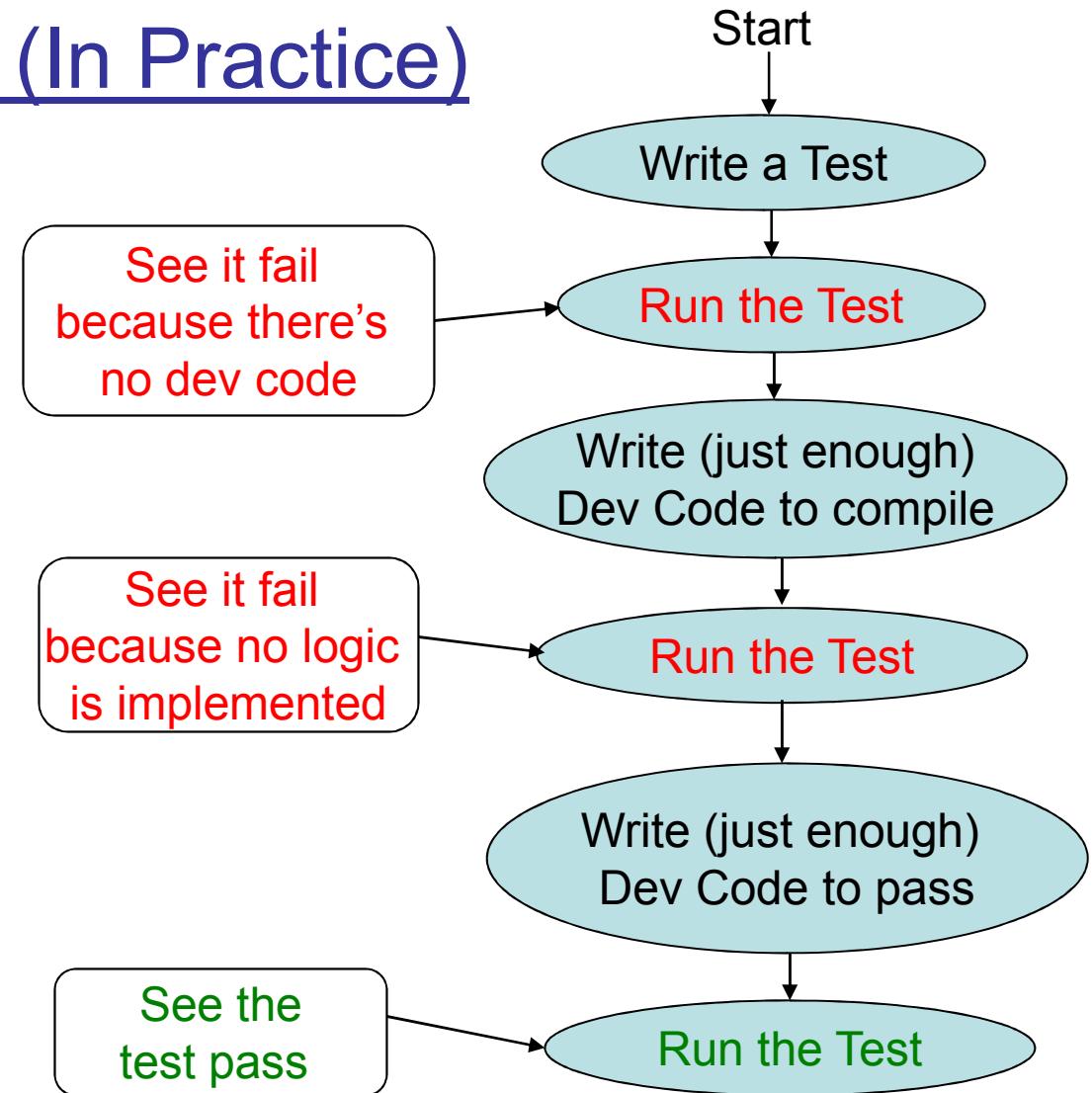
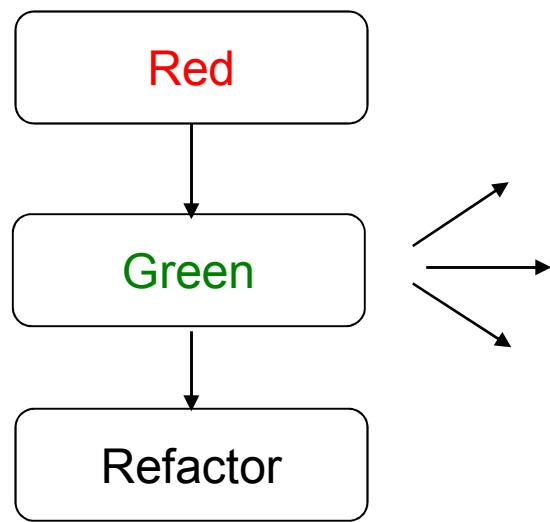
See it fail
because there's
no dev code

See it fail
because no logic
is implemented



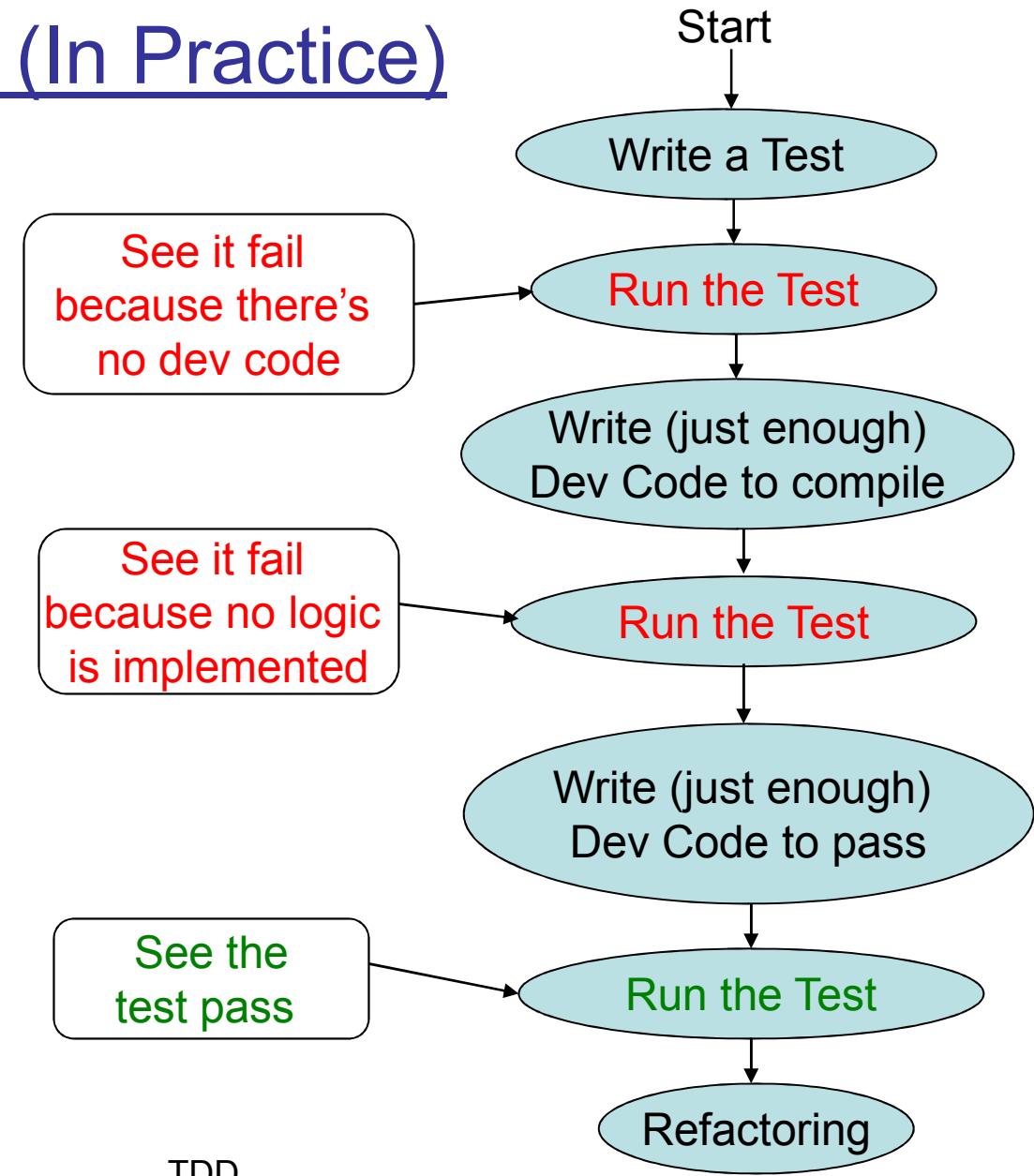
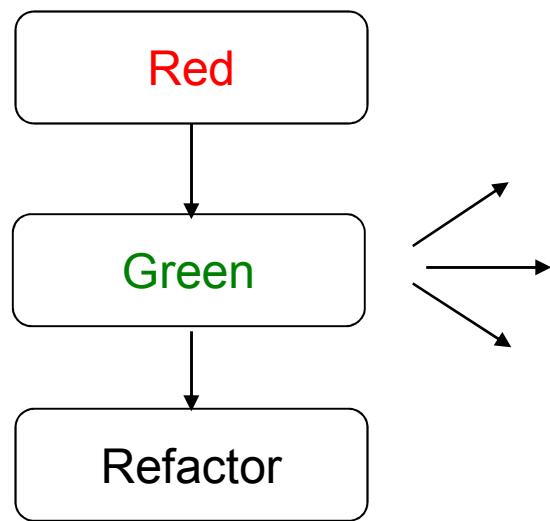
TDD

Principle of TDD (In Practice)



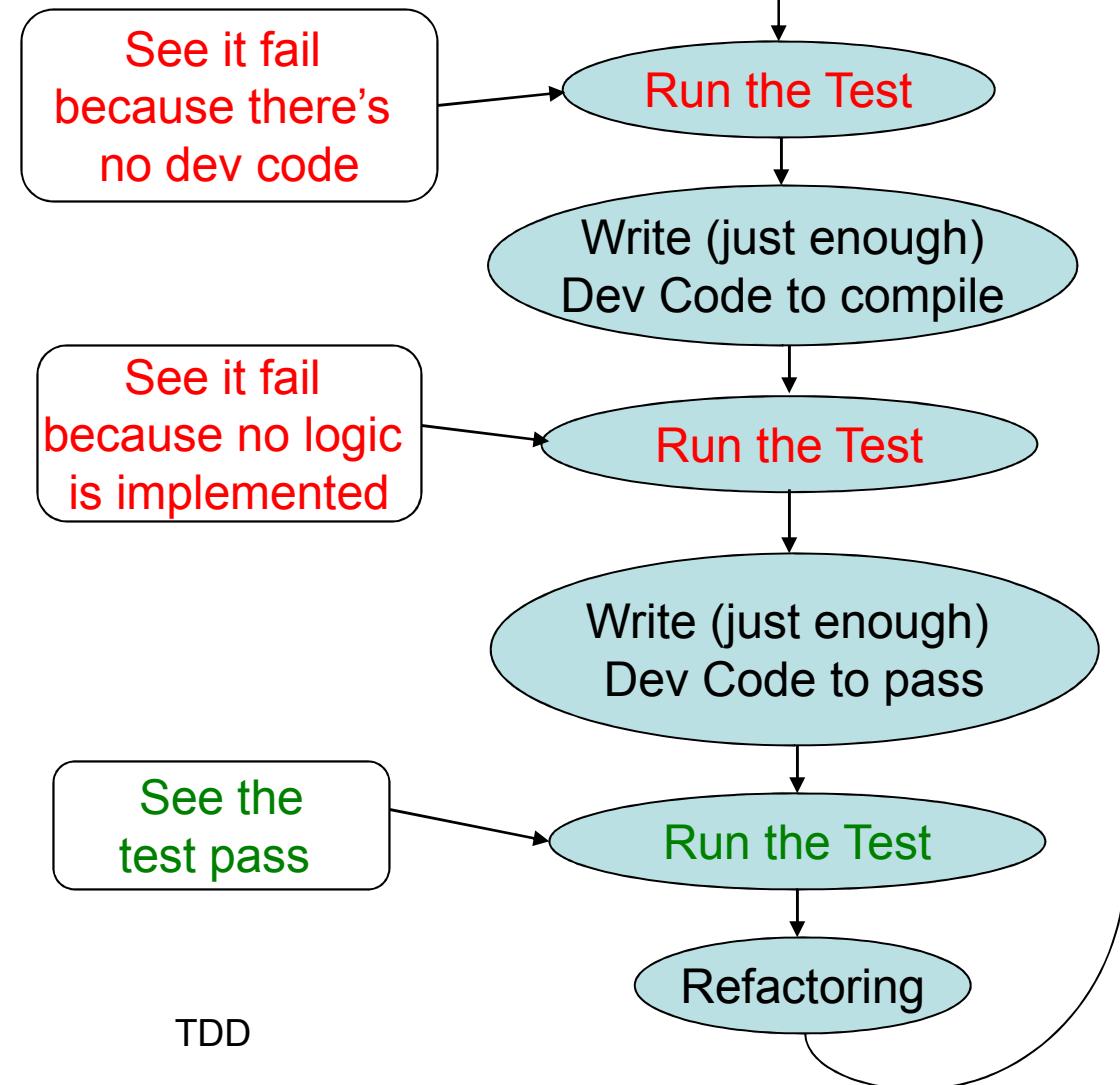
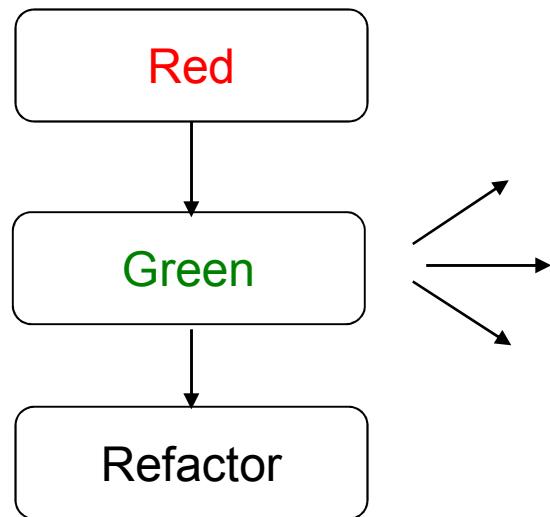
TDD

Principle of TDD (In Practice)



TDD

Principle of TDD (In Practice)

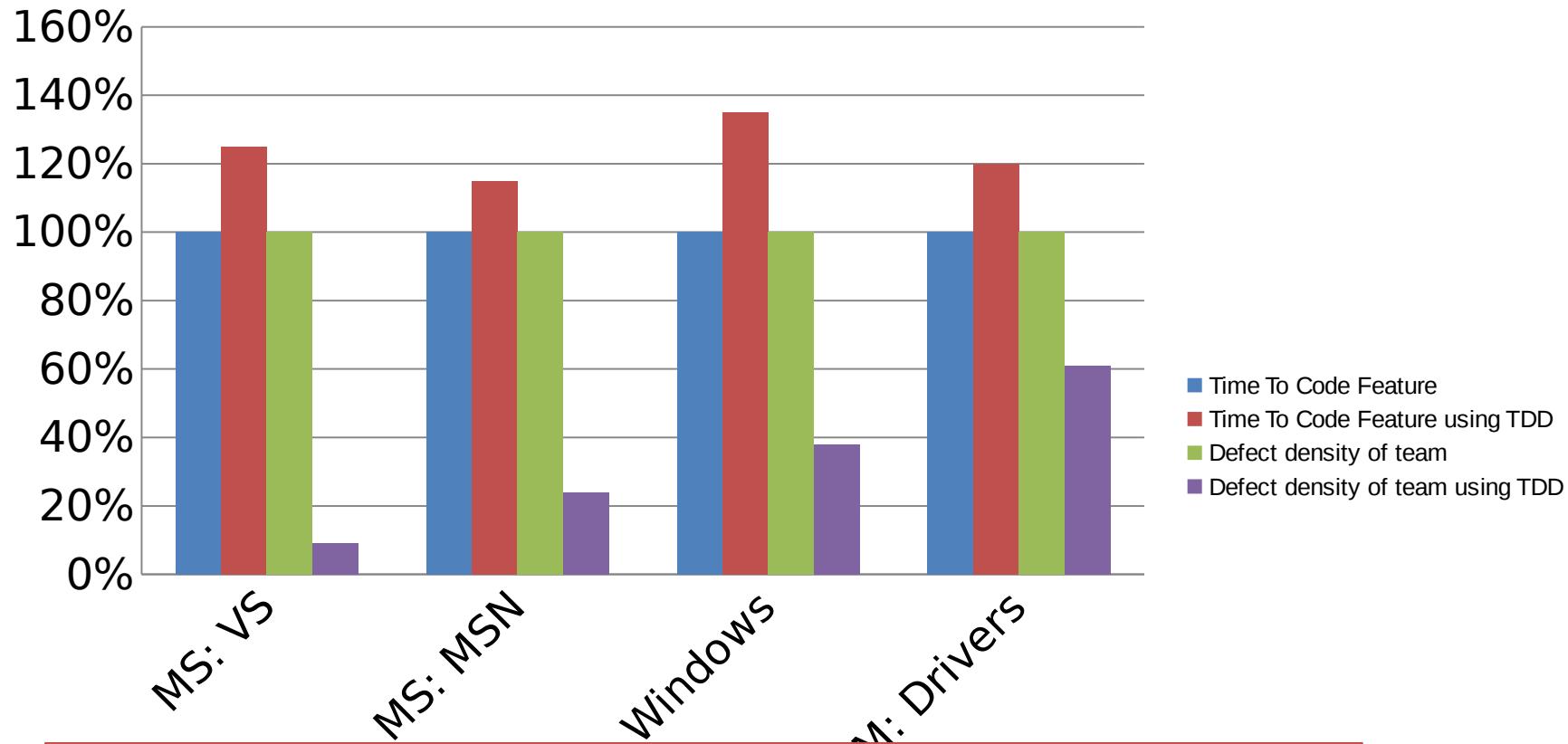


Why TDD

WHY / BENEFITS

- Confidence in change
 - Increase confidence in code
 - Fearlessly change your code
- Document requirements
- Discover usability issues early
- Regression testing = Stable software = Quality software

IS TDD A WASTE OF TIME (MICROSOFT RESEARCH)



Major quality improvement for minor time investment



ADVANTAGES OF TDD

- TDD shortens the programming feedback loop
 - TDD promotes the development of high-quality code
 - User requirements more easily understood
 - Reduced interface misunderstandings
 - TDD provides concrete evidence that your software works
 - Reduced software defect rates
 - Better Code
 - Less Debug Time.
-

DISADVANTAGES OF TDD

- Programmers like to code, not to test
- Test writing is time consuming
- Test completeness is difficult to judge
- TDD may not always work

How to

REMEMBER - THERE ARE OTHER KINDS OF TESTS

- Unit test (Unit)
- Integration test (Collaboration)
- User interface test (Frontend)
- Regression test (Continuous Integration)
- ..., System, Performance, Stress, Usability, ...

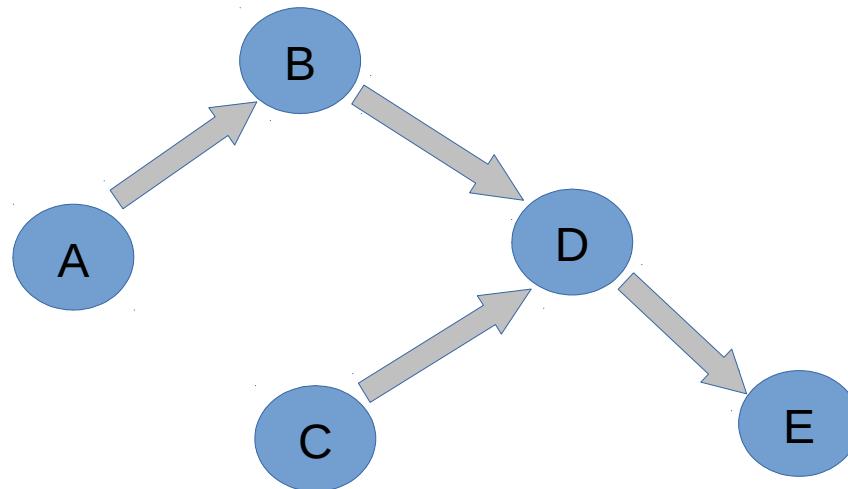
The only tests relevant to TDD is Black-box Unit Testing

- White-box test



HOW TO DO TDD

- ... on my component A?
- Unit Test A, but what about B, C, D...?



HOW TO DO TDD

- ... on my component A?

This, and the previous slide, are all thoughts on implementing tests on existing code.

This is all White-box Unit- or Integration Testing and has therefore nothing to do with TDD.

In TDD you write a Black-box Unit test that fails, first, and worry about the code implementation later.

SINGLE MOST IMPORTANT THING WHEN LEARNING TDD

**Do not
write the code in your
head
before you write the
test**



SINGLE MOST IMPORTANT THING WHEN LEARNING TDD

- When you first start at doing TDD you "know" what the design should be. You "know" what code you want to write. So you write a test that will let you write that bit of code.
- When you do this you aren't really doing TDD - since you are writing the code first (even if the code is only in your head)
- It takes some time to (and some poking by clever folk) to realize that you need to focus on the test. Write the test for the behavior you want - then write the minimal code needed to make it pass - then let the design emerge through refactoring. Repeat until done.



Where to begin

UNBOUNDED STACK EXAMPLE

- Requirement: FILO / LIFO messaging system
- Brainstorm a list of tests for the requirement:
 - Create a *Stack* and verify that *IsEmpty* is true.
 - *Push* a single object on the *Stack* and verify that *IsEmpty* is false.
 - *Push* a single object, *Pop* the object, and verify that *IsEmpty* is true.
 - *Push* a single object, remembering what it is; *Pop* the object, and verify that the two objects are equal.
 - *Push* three objects, remembering what they are; *Pop* each one, and verify that they are removed in the correct order.
 - *Pop* a *Stack* that has no elements.
 - *Push* a single object and then call *Top*. Verify that *IsEmpty* is false.
 - *Push* a single object, remembering what it is; and then call *Top*. Verify that the object that is returned is the same as the one that was pushed.
 - Call *Top* on a *Stack* with no elements.

UNBOUNDED STACK EXAMPLE

- Choosing the First Test?
 - The simplest.
 - The essence.

Answers:

- If you need to write code that is untested, choose a simpler test.
- If the essence approach takes too much time to implement, choose a simpler test.

UNBOUNDED STACK EXAMPLE

- Anticipating future tests, or not?

Answers:

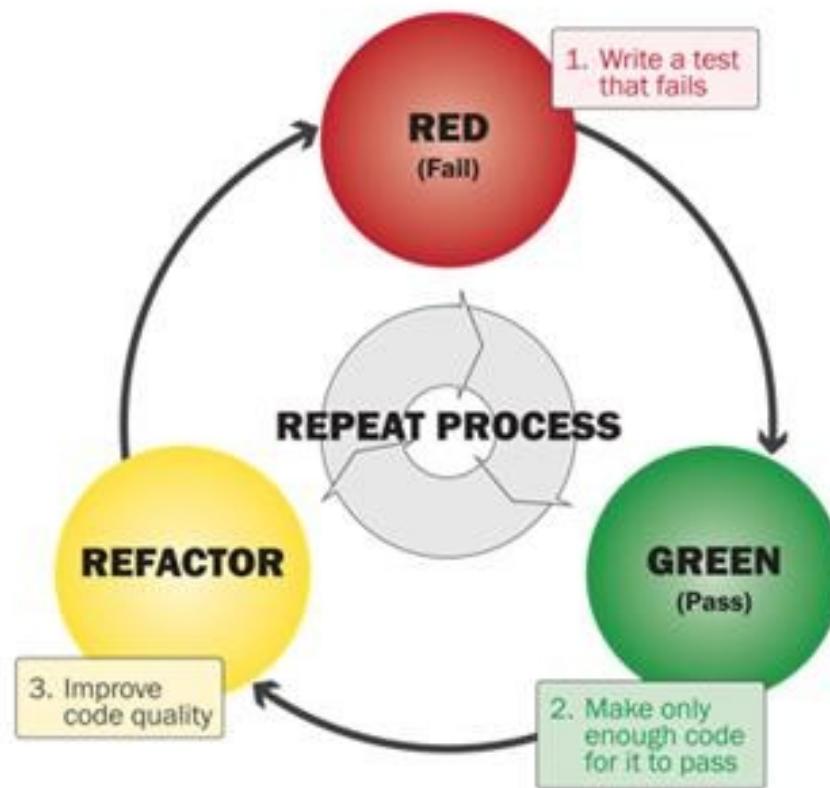
- In the beginning, focus on the test you are writing, and do not think of the other tests.
- As you become familiar with the technique and the task, you can increase the size of the steps.
- But remember still, no written code must be untested.

WHERE TO GO FROM HERE

- You don't have to start big
- Start new tasks with TDD
- Add Tests to code that you need to change or maintain – but only to small parts
- Proof of concept

If it's worth building, it's worth testing.
If it's not worth testing,
*why are you wasting your time working on
it?*

SUMMARY



LINKS

- Books
 - Test-Driven Development in Microsoft® .NET (<http://www.amazon.co.uk/gp/product/0735619484>)
 - Test-Driven Development by Kent Beck (C++) (<http://www.amazon.co.uk/gp/product/0321146530>)
- Other links:
 - http://en.wikipedia.org/wiki/Test-driven_development
 - <http://www.testdriven.com/>
 - <http://www.mockobjects.com/> - Online TDD book:
<http://www.martinfowler.com/articles/mocksArentStubs.html>
<http://www.mockobjects.com/book/index.html>
 - <http://dannorth.net/introducing-bdd>
 - <http://behaviour-driven.org/Introduction>

TDD by examples

- ▶ Example 1: write a method that reverse last 2 characters of string.
- ▶ If null -> return null, if empty -> return empty, if length of string equal 1 -> return itself
- ▶ Ex: “A” -> “A”, “”-> “”, null -> null, “AB”-> “BA”, “RAIN” -> “RANI”

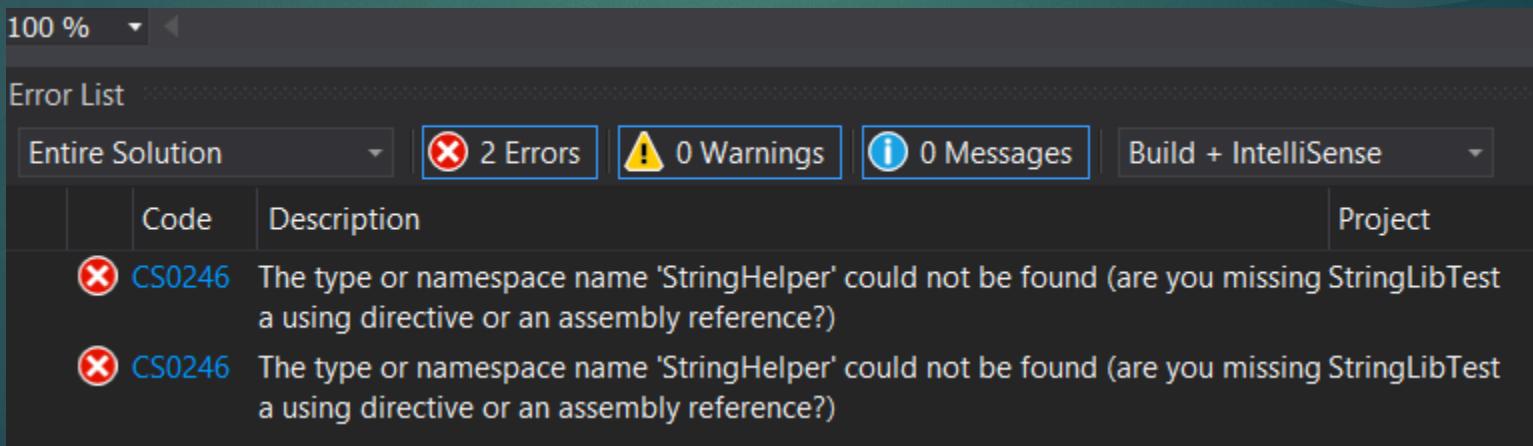
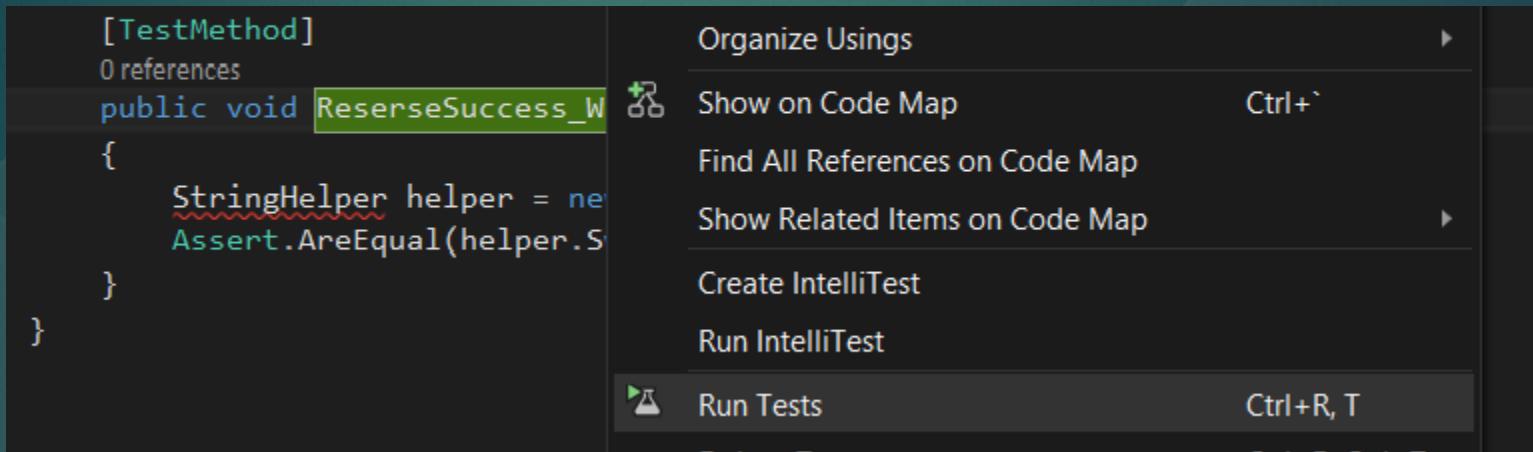
TDD by examples

- ▶ Create first test case

```
[TestClass]
0 references
public class StringHelperTest
{
    [TestMethod]
    0 references
    public void ReserseSuccess_WhenInputHas2Characters()
    {
        StringHelper helper = new StringHelper();
        Assert.AreEqual(helper.SwapLast2Chars("BA"), "AB");
    }
}
```

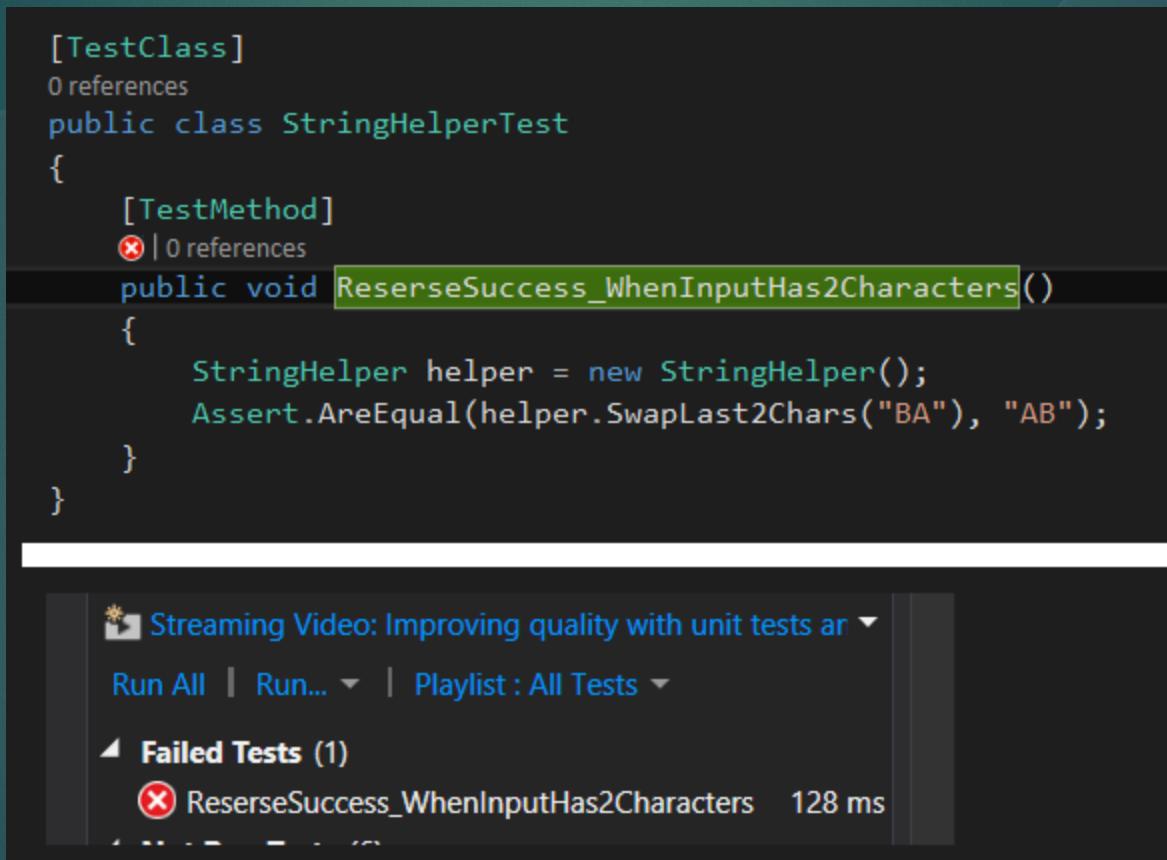
TDD by examples

- ▶ Can not compile because StringHelper class is not created



TDD by examples

- ▶ After StringHelper class is created, run **ALL TEST CASE** to see they(or one of them) fail



The screenshot shows a code editor with a dark theme. A file named 'StringHelperTest.cs' is open, containing the following C# code:

```
[TestClass]
0 references
public class StringHelperTest
{
    [TestMethod]
    ❌ | 0 references
    public void ReserseSuccess_WhenInputHas2Characters()
    {
        StringHelper helper = new StringHelper();
        Assert.AreEqual(helper.SwapLast2Chars("BA"), "AB");
    }
}
```

Below the code editor is a test runner interface. It displays the title 'Streaming Video: Improving quality with unit tests ar' and navigation buttons 'Run All' and 'Run...'. Under the 'Failed Tests' section, there is one entry:

- Failed Tests (1)
 - ❌ ReserseSuccess_WhenInputHas2Characters 128 ms

TDD by examples

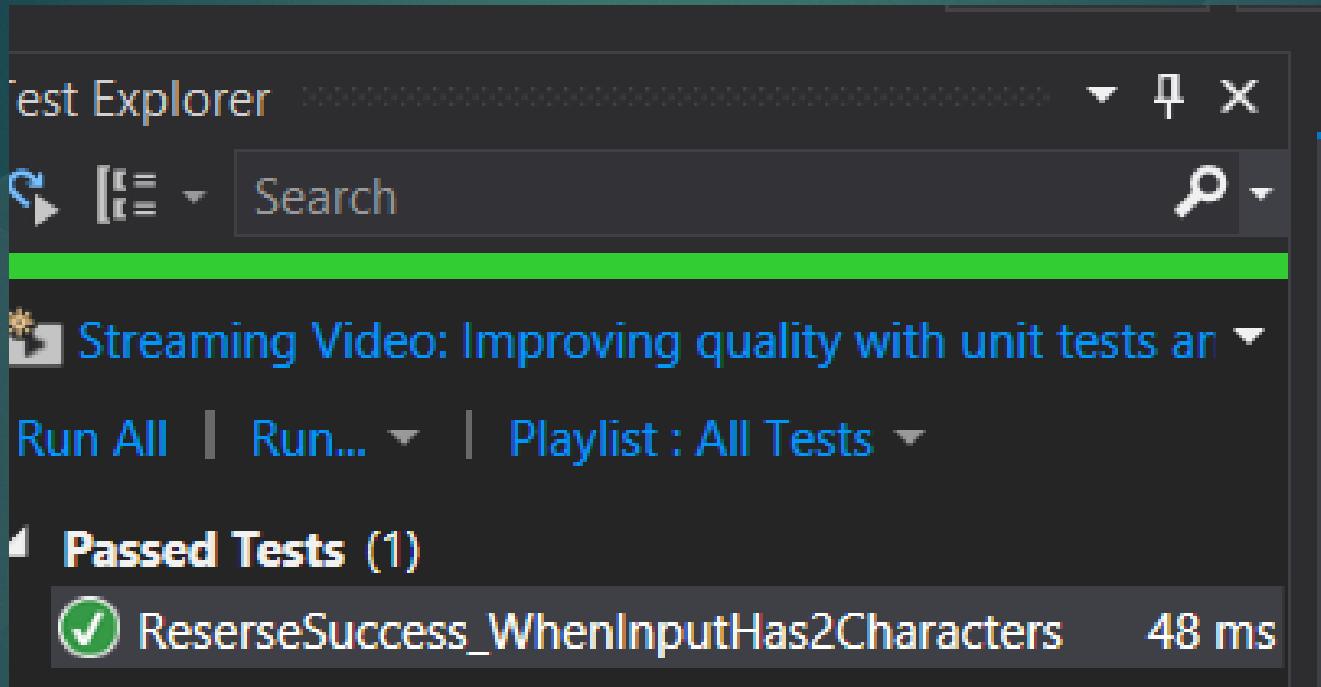
- ▶ Make a **little change** to pass this test cases

```
4 references
public class StringHelper
{
    2 references | 1/1 passing
    public string SwapLast2Chars(string input)
    {
        string first = input[0].ToString();
        string second = input[1].ToString();
        return (second + first).ToString();
    }
}
```

Question : Who has a better implementation ?

TDD by examples

- ▶ Run ALL TEST CASES to see they pass



TDD by examples

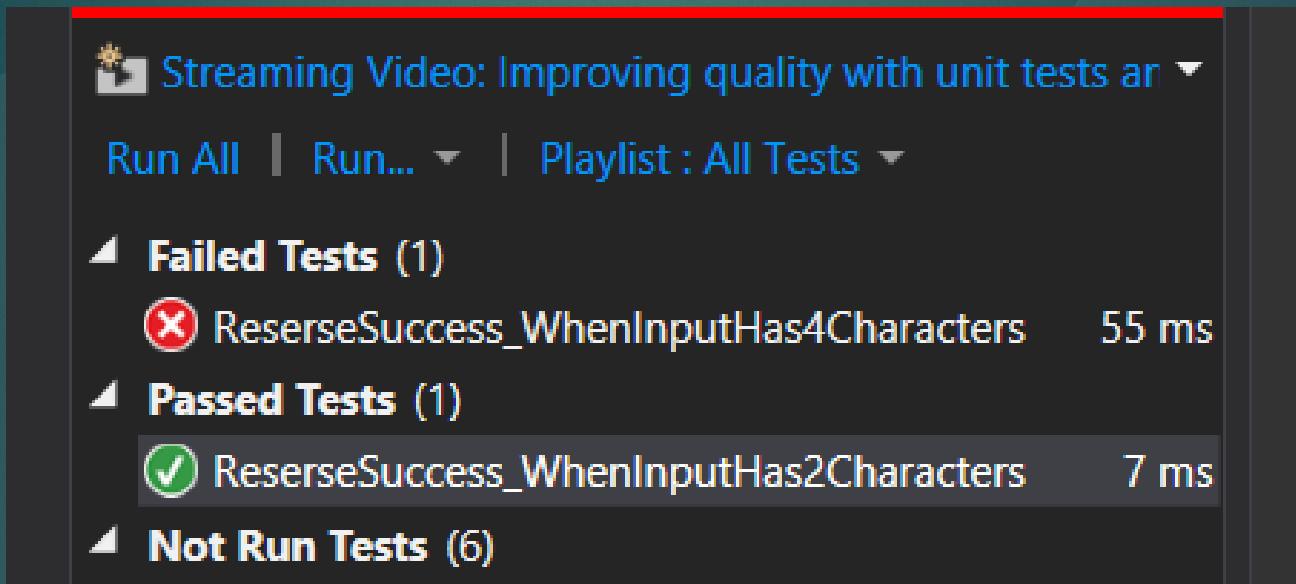
- We have just finished a baby step of TDD cycle. Repeat it again.

```
[TestClass]
0 references
public class StringHelperTest
{
    [TestMethod]
    ✓ | 0 references
    public void ReserseSuccess_WhenInputHas2Characters()
    {
        StringHelper helper = new StringHelper();
        Assert.AreEqual(helper.SwapLast2Chars("BA"), "AB");
    }

    [TestMethod]
    0 references
    public void ReserseSuccess_WhenInputHas4Characters()
    {
        StringHelper helper = new StringHelper();
        Assert.AreEqual(helper.SwapLast2Chars("ABCD"), "ABDC");
    }
}
```

TDD by examples

- ▶ After StringHelper class is created, run **ALL TEST CASE** to see they(or one of them) fail



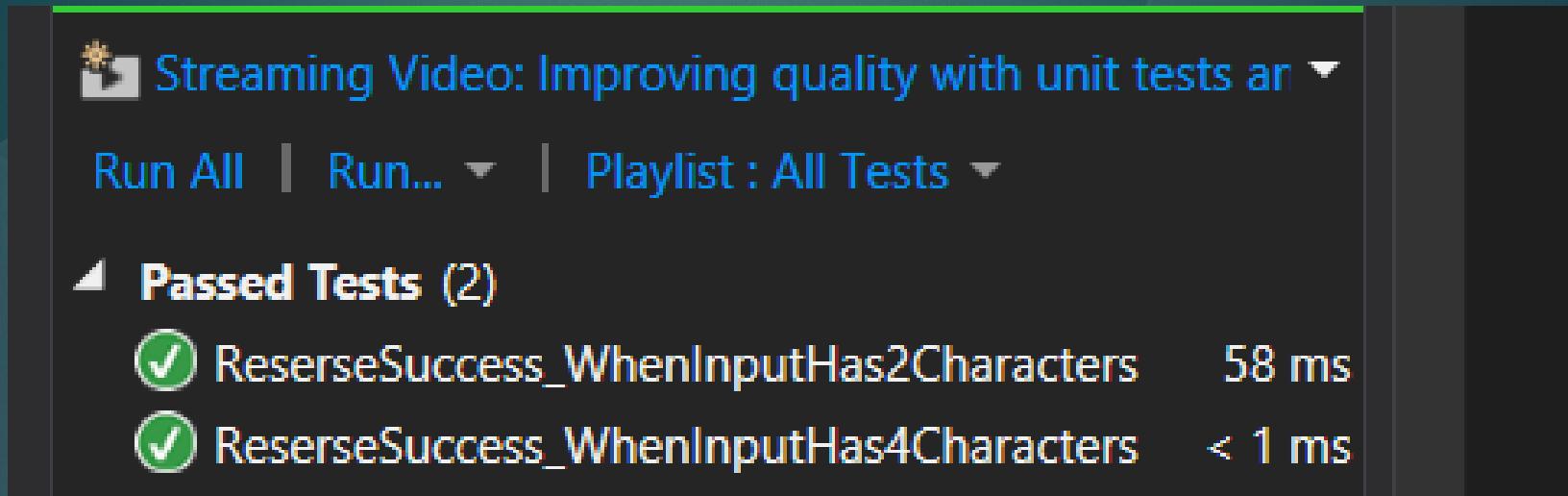
TDD by examples

- ▶ Refactor code to pass this test case

```
3 references | 2/2 passing
public string SwapLast2Chars(string input)
{
    //string first = input[0].ToString();
    //string second = input[1].ToString();
    //return (second + first).ToString();
    int length = input.Length;
    string stringMinus2LastChars = input.Substring(0, length - 2);
    string secondLastChar = input[length - 2].ToString();
    string lastChar = input[length - 1].ToString();
    return stringMinus2LastChars + lastChar + secondLastChar;
}
```

TDD by examples

- ▶ Run all test cases to see they pass



TDD by examples

- Refactor test code if necessary

```
[TestClass]
0 references
public class StringHelperTest
{
    StringHelper helper = new StringHelper();

    [TestMethod]
    ✓ | 0 references
    public void ReserseSuccess_WhenInputHas2Characters()
    {
        //StringHelper helper = new StringHelper();
        Assert.AreEqual(helper.SwapLast2Chars("BA"), "AB");
    }

    [TestMethod]
    ✓ | 0 references
    public void ReserseSuccess_WhenInputHas4Characters()
    {
        //StringHelper helper = new StringHelper();
        Assert.AreEqual(helper.SwapLast2Chars("ABCD"), "ABDC");
    }
}
```

TDD by examples

- ▶ Add new test case

```
StringHelper helper = new StringHelper();

[TestMethod]
✓ | 0 references
public void ReserseSuccess_WhenInputHas2Characters()
{
    Assert.AreEqual(helper.SwapLast2Chars("BA"), "AB");
}

[TestMethod]
✓ | 0 references
public void ReserseSuccess_WhenInputHas4Characters()
{
    Assert.AreEqual(helper.SwapLast2Chars("ABCD"), "ABDC");
}

[TestMethod]
0 references
public void ReserseSuccess_WhenInputHas1Characters()
{
    Assert.AreEqual(helper.SwapLast2Chars("A"), "A");
}
```

TDD by examples

- ▶ After StringHelper class is created, run **ALL TEST CASE** to see they(or one of them) fail

The screenshot shows a test runner interface with the following details:

- Header: Streaming Video: Improving quality with unit tests and more
- Control buttons: Run All | Run... | Playlist : All Tests
- Test Results:
 - Failed Tests (1): ReserseSuccess_WhenInputHas1Characters (62 ms)
 - Passed Tests (8):
 - ReserseSuccess_WhenInputHas2Characters (6 ms)
 - ReserseSuccess_WhenInputHas4Characters (< 1 ms)

TDD by examples

- Refactor code to pass the test case

```
4 references | ✘ 2/3 passing
public string SwapLast2Chars(string input)
{
    int length = input.Length;
    if (length < 2)
    {
        return input;
    }
    string stringMinus2LastChars = input.Substring(0, length - 2);
    string secondLastChar = input[length - 2].ToString();
    string lastChar = input[length - 1].ToString();
    return stringMinus2LastChars + lastChar + secondLastChar;
}
```

TDD by examples

- ▶ Run all test cases to see they pass

◀ Passed Tests (9)

- ✓ ReserseSuccess_WhenInputHas1Characters < 1 ms
- ✓ ReserseSuccess_WhenInputHas2Characters 7 ms
- ✓ ReserseSuccess_WhenInputHas4Characters < 1 ms

TDD by examples

- ▶ Add new test case, this test case will be pass without any changes

```
[TestMethod]
✔ | 0 references
public void ReserseSuccess_WhenInputEmpty()
{
    Assert.AreEqual(helper.SwapLast2Chars(""), "");
}
```

◀ Passed Tests (10)

✓ ReserseSuccess_WhenInputEmpty	< 1 ms
✓ ReserseSuccess_WhenInputHas1Characters	< 1 ms
✓ ReserseSuccess_WhenInputHas2Characters	8 ms
✓ ReserseSuccess_WhenInputHas4Characters	< 1 ms

TDD by examples

- ▶ Add new test case, this test case will be fail

```
[TestMethod]
✖ | 0 references
public void ReserseSuccess_WhenInputIsNull()
{
    Assert.AreEqual(helper.SwapLast2Chars(null), null);
}
```

◀ **Failed Tests** (1)
✖ ReserseSuccess_WhenInputIsNull 69 ms

TDD by examples

- Refactor code

```
6 references | ✘ 4/5 passing
public string SwapLast2Chars(string input)
{
    if (input == null)
    {
        return null;
    }
    int length = input.Length;
    if (length < 2)
    {
        return input;
    }
    string stringMinus2LastChars = input.Substring(0, length - 2);
    string secondLastChar = input[length - 2].ToString();
    string lastChar = input[length - 1].ToString();
    return stringMinus2LastChars + lastChar + secondLastChar;
}
```

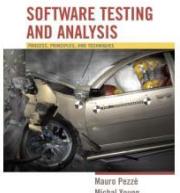
TDD by examples

- ▶ Run all test cases, if they pass and you don't have any test case. The TDD cycle is completed. Refactor code if necessary

► Passed Tests (5)

	ReserseSuccess_WhenInputEmpty	< 1 ms
	ReserseSuccess_WhenInputHas1Characters	< 1 ms
	ReserseSuccess_WhenInputHas2Characters	19 ms
	ReserseSuccess_WhenInputHas4Characters	< 1 ms
	ReserseSuccess_WhenInputIsNull	< 1 ms

Integration and Component-based Software Testing

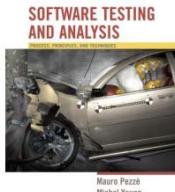


(c) 2007 Mauro Pezzè & Michal Young

Ch 21, slide 1

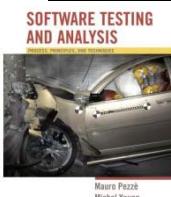
Learning objectives

- Understand the purpose of integration testing
 - Distinguish typical integration faults from faults that should be eliminated in unit testing
 - Understand the nature of integration faults and how to prevent as well as detect them
- Understand strategies for ordering construction and testing
 - Approaches to incremental assembly and testing to reduce effort and control risk
- Understand special challenges and approaches for testing component-based systems



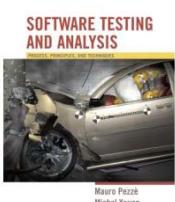
What is integration testing?

	Module test	Integration test	System test
Specification:	Module interface	Interface specs, module breakdown	Requirements specification
Visible structure:	Coding details	Modular structure (software architecture)	— none —
Scaffolding required:	Some	Often extensive	Some
Looking for faults in:	Modules	Interactions, compatibility	System functionality



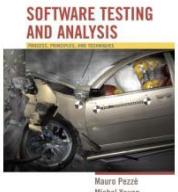
Integration versus Unit Testing

- Unit (module) testing is a necessary foundation
 - Unit level has maximum controllability and visibility
 - Integration testing can never compensate for inadequate unit testing
- Integration testing may serve as a *process check*
 - If module faults are revealed in integration testing, they signal inadequate unit testing
 - If integration faults occur in interfaces between correctly implemented modules, the errors can be traced to module breakdown and interface specifications



Integration Faults

- Inconsistent interpretation of parameters or values
 - Example: Mixed units (meters/yards) in Martian Lander
- Violations of value domains, capacity, or size limits
 - Example: Buffer overflow
- Side effects on parameters or resources
 - Example: Conflict on (unspecified) temporary file
- Omitted or misunderstood functionality
 - Example: Inconsistent interpretation of web hits
- Nonfunctional properties
 - Example: Unanticipated performance issues
- Dynamic mismatches
 - Example: Incompatible polymorphic method calls



Example: A Memory Leak

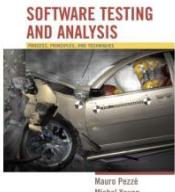
Apache web server, version 2.0.48

Response to normal page request on secure (https) port

```
static void ssl io filter disable(ap filter t *f)
{
    bio filter in ctx t *inctx = f->ctx;

    inctx->ssl = NULL;
    inctx->filter ctx->pssl
}
```

No obvious error, but
Apache leaked memory
slowly (in normal use) or
quickly (if exploited for a
DOS attack)



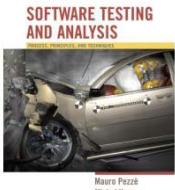
Example: A Memory Leak

Apache web server, version 2.0.48

Response to normal page request on secure (https) port

```
static void ssl io filter disable(ap filter t *f)
{
    bio filter in ctx t *inctx = f->ctx;
    SSL_free(inctx -> ssl);
    inctx->ssl = NULL;
    inctx->filter ctx->pssl
}
```

The missing code is for a **structure defined and created elsewhere**, accessed through an opaque pointer.



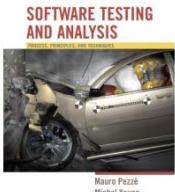
Example: A Memory Leak

Apache web server, version 2.0.48

Response to normal page request on secure (https) port

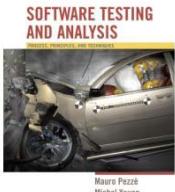
```
static void ssl io filter disable(ap filter t *f)
{
    bio filter in ctx t *inctx = f->ctx;
    SSL_free(inctx -> ssl);
    inctx->ssl = NULL;
    inctx->filter ctx->pssl
}
```

Almost impossible to find
with unit testing.
(Inspection and some
dynamic techniques could
have found it.)



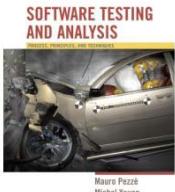
Maybe you've heard ...

- Yes, I implemented ⟨module A⟩, but I didn't test it thoroughly yet. It will be tested along with ⟨module A⟩ when that's ready.

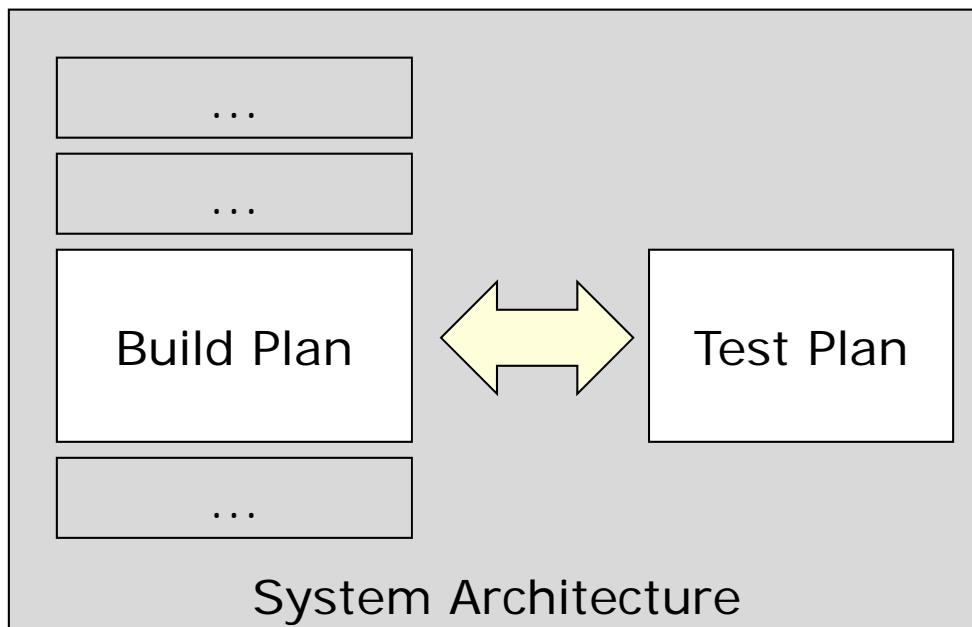


Translation...

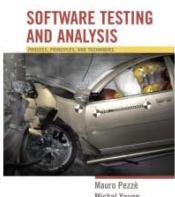
- Yes, I implemented **{module A}**, but I didn't test it thoroughly yet. It will be tested along with **{module A}** when that's ready.
- I didn't think at all about the **strategy** for testing. I didn't design **{module A}** for testability and I didn't think about the best order to build and test modules **{A}** and **{B}**.



Integration Plan + Test Plan



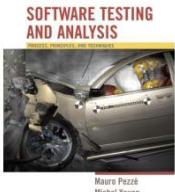
- Integration test plan drives and is driven by the project “build plan”
 - A key feature of the system architecture and project plan



Big Bang Integration Test

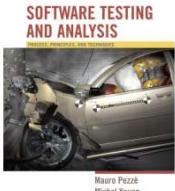
An extreme and desperate approach:
Test only after integrating all modules

- + Does not require scaffolding
 - The only excuse, and a bad one
- Minimum observability, diagnosability, efficacy, feedback
- High cost of repair
 - Recall: Cost of repairing a fault rises as a function of *time between error and repair*

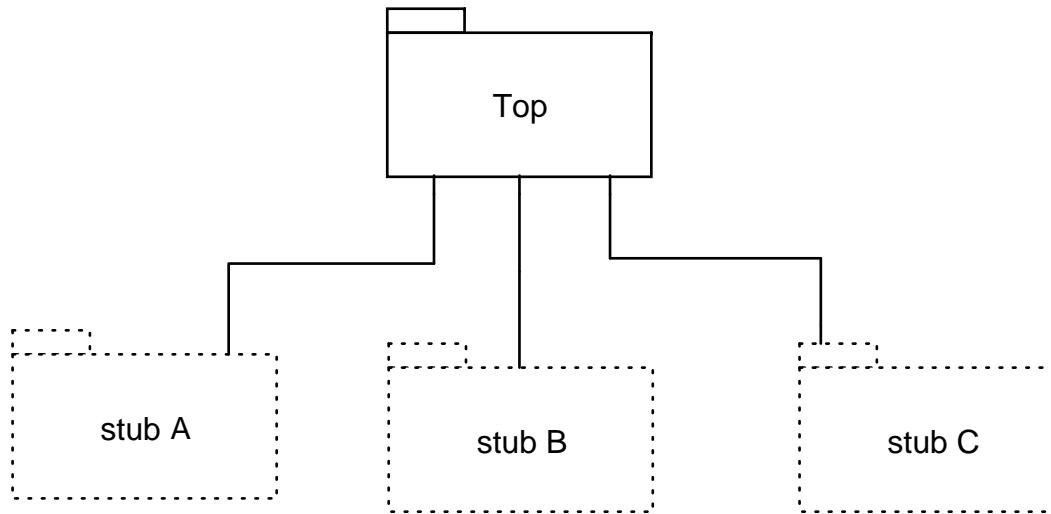


Structural and Functional Strategies

- Structural orientation:
Modules constructed, integrated and tested based on a hierarchical project structure
 - Top-down, Bottom-up, Sandwich, Backbone
- Functional orientation:
Modules integrated according to application characteristics or features
 - Threads, Critical module

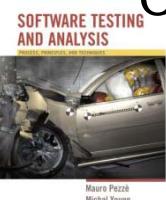


Top down .

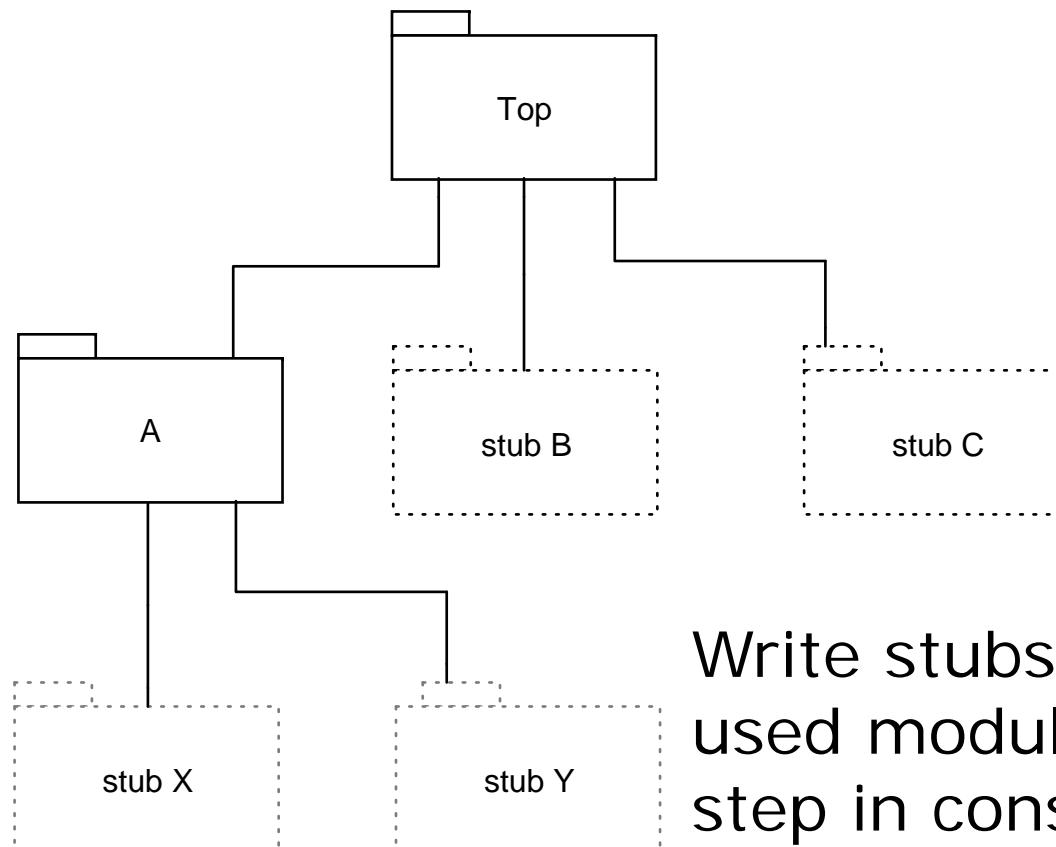


Working from the top level (in terms of “use” or “include” relation) toward the bottom.

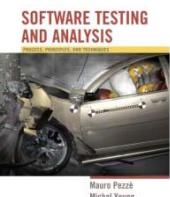
No drivers required if program tested from top-level interface (e.g. GUI, CLI, web app, etc.)



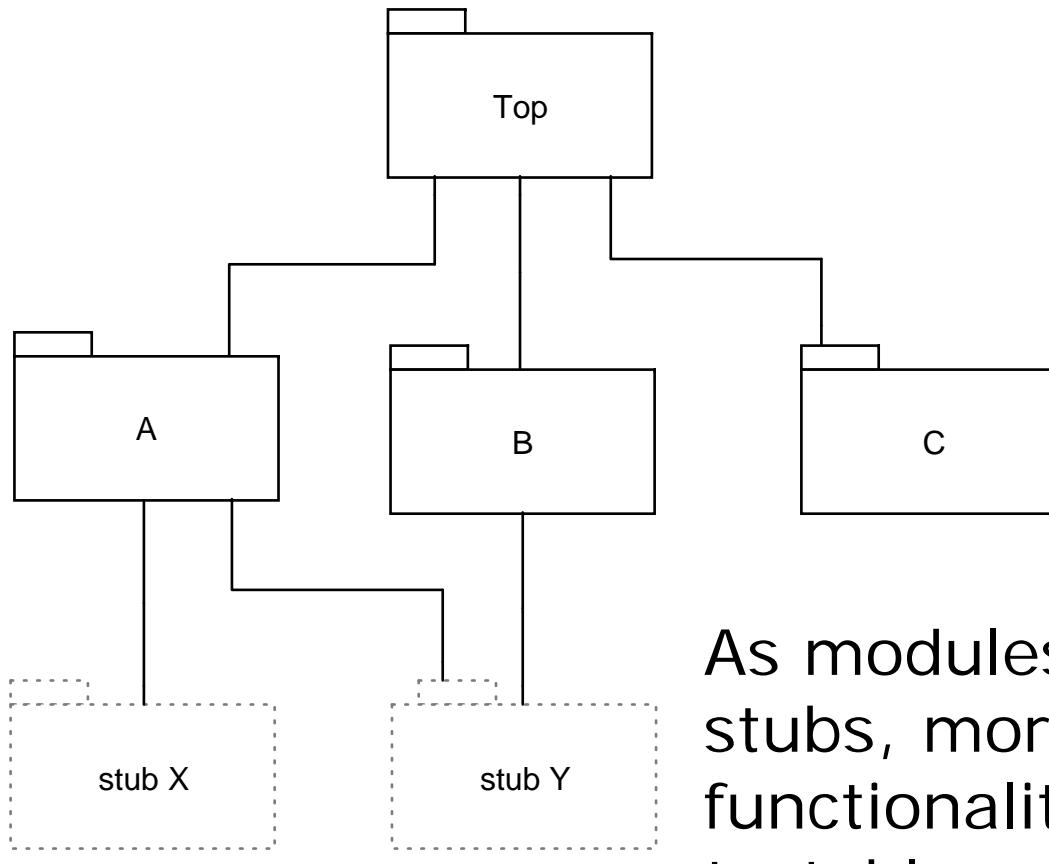
Top down ..



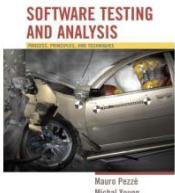
Write stubs of called or used modules at each step in construction



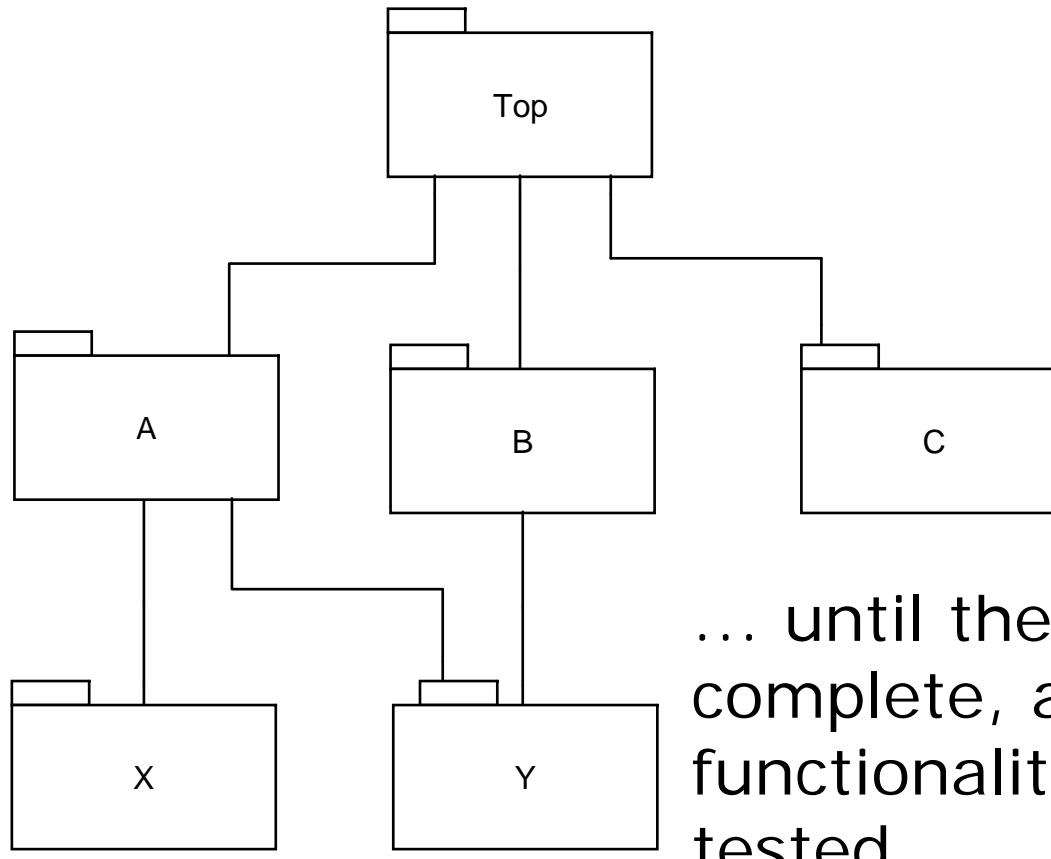
Top down ...



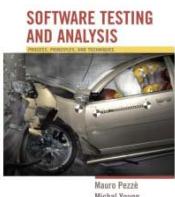
As modules replace
stubs, more
functionality is
testable



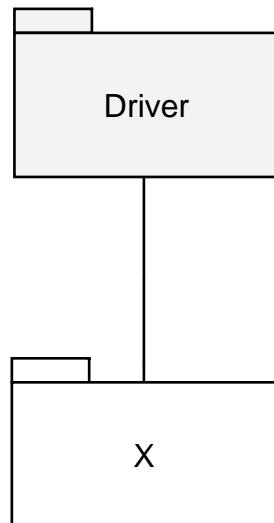
Top down ... complete



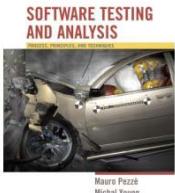
... until the program is complete, and all functionality can be tested



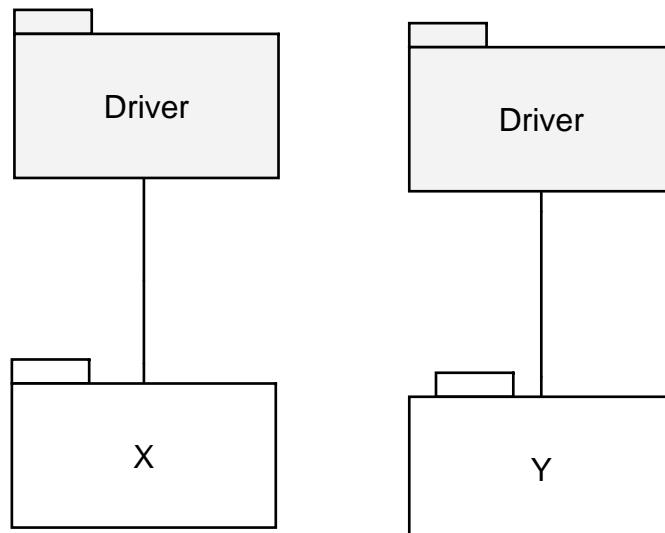
Bottom Up .



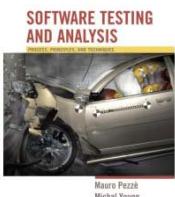
Starting at the leaves of the “uses” hierarchy, we never need stubs



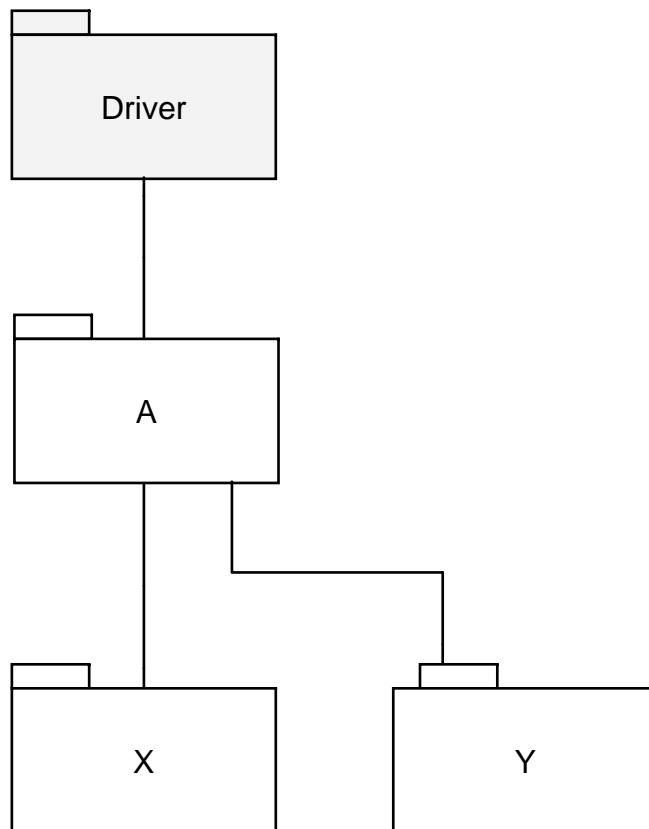
Bottom Up ..



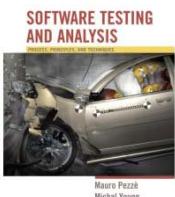
... but we must
construct drivers for
each module (as in
unit testing) ...



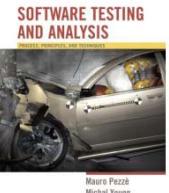
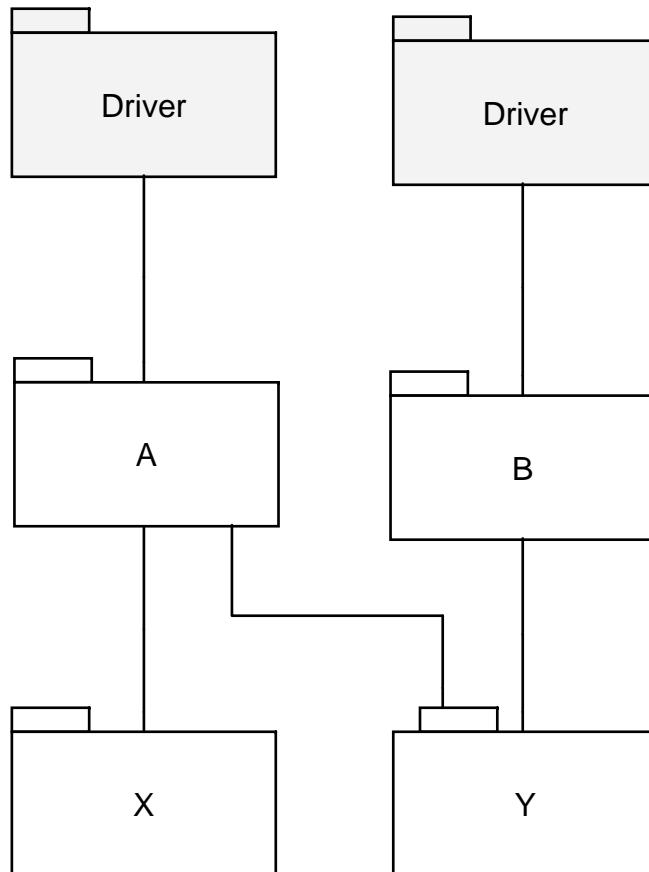
Bottom Up ...



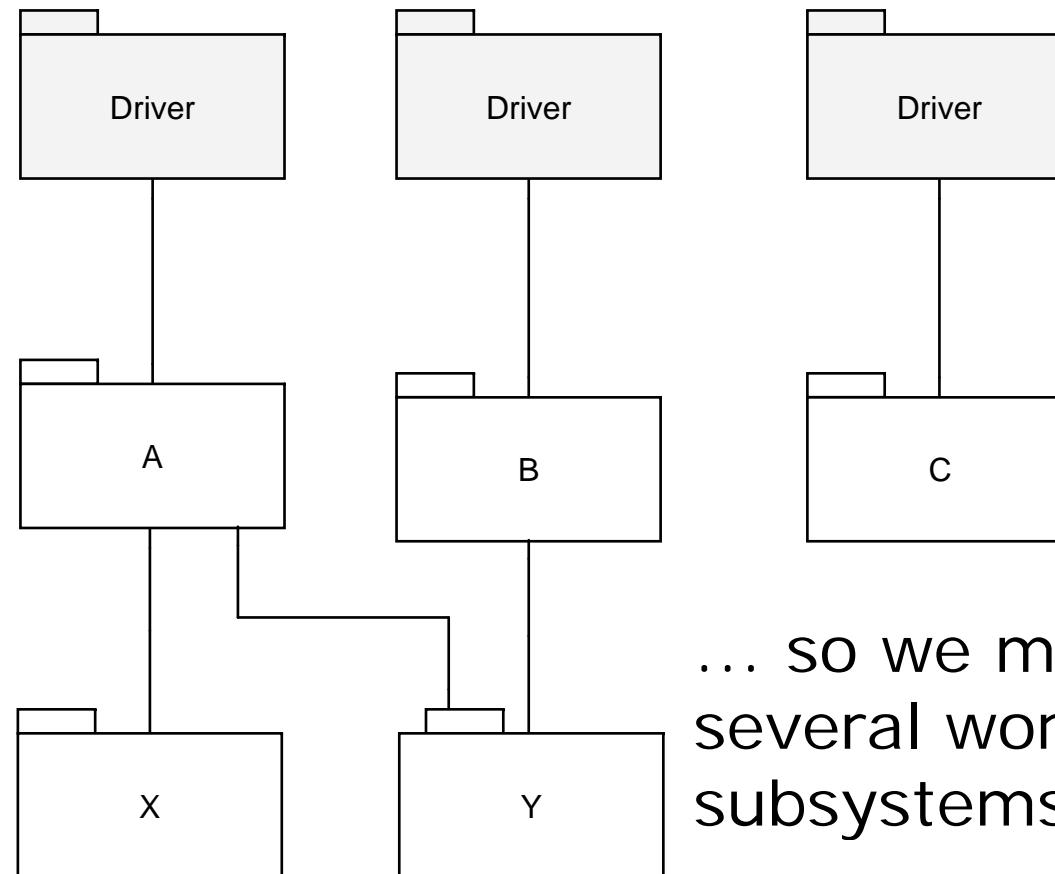
... an intermediate module replaces a driver, and needs its own driver ...



Bottom Up

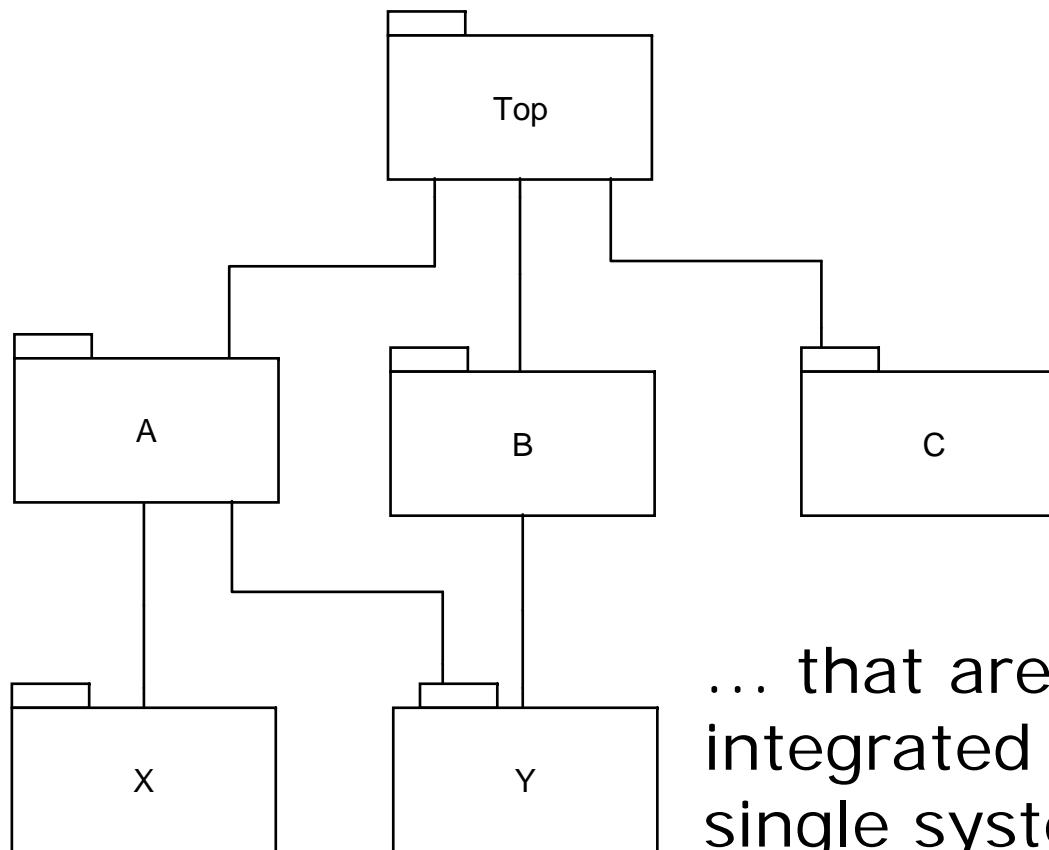


Bottom Up

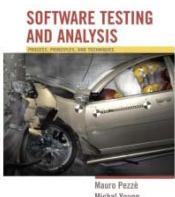


... so we may have
several working
subsystems ...

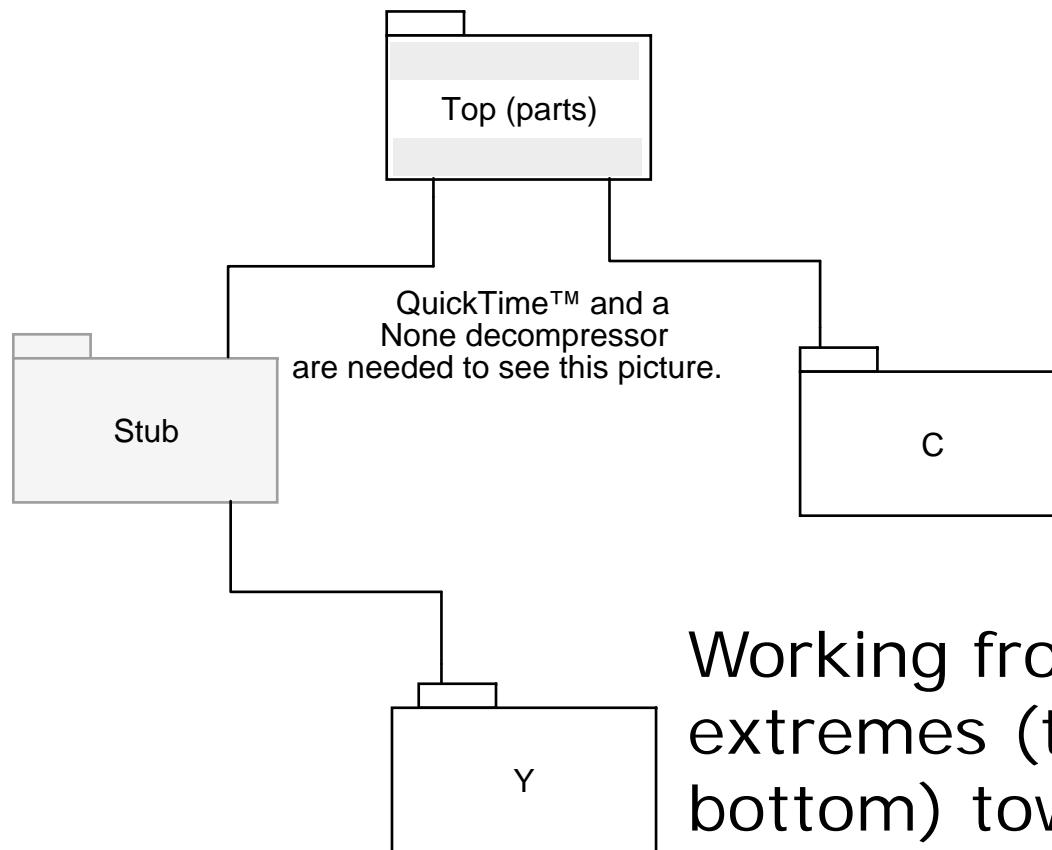
Bottom Up (complete)



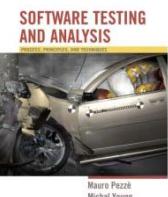
... that are eventually integrated into a single system.



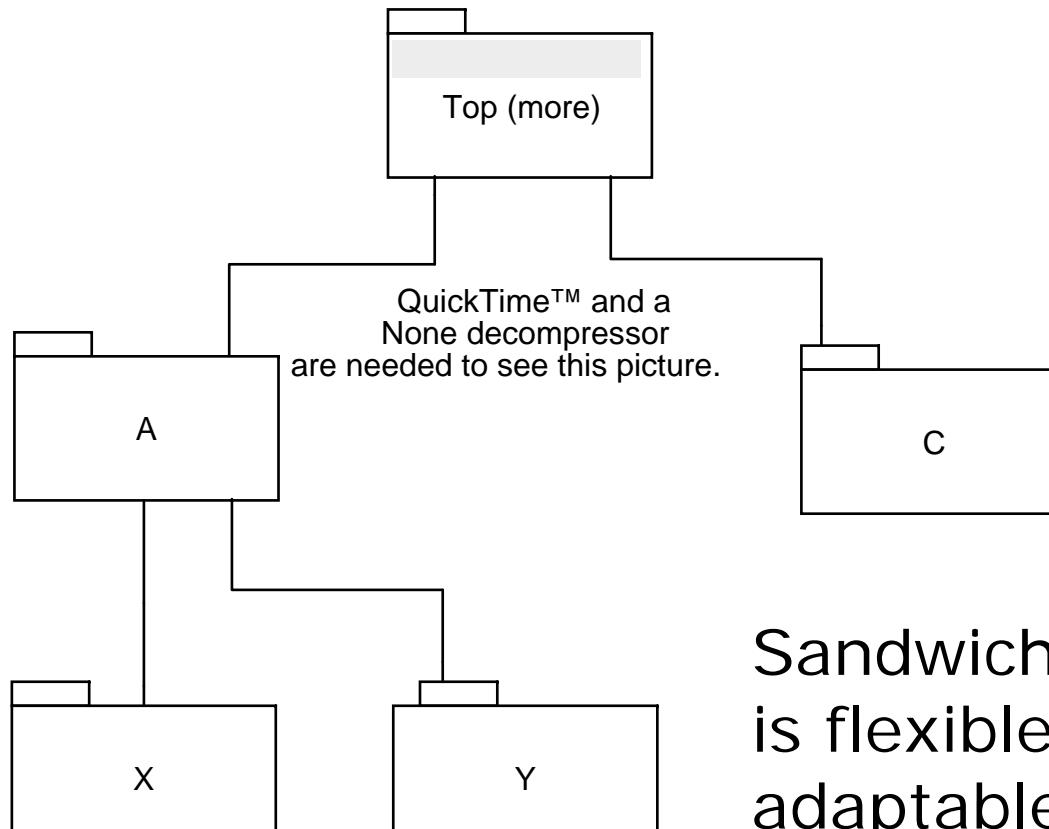
Sandwich .



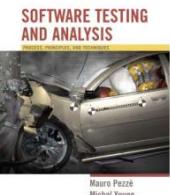
Working from the extremes (top and bottom) toward center, we may use fewer drivers and stubs



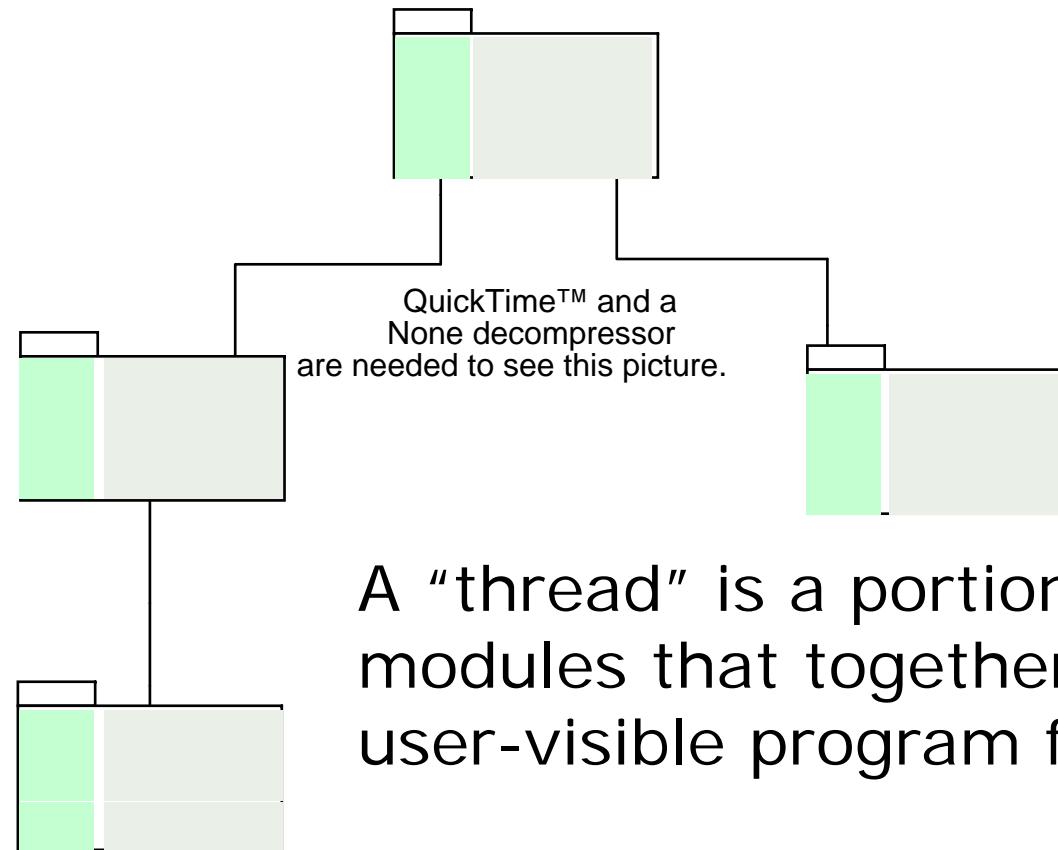
Sandwich ...



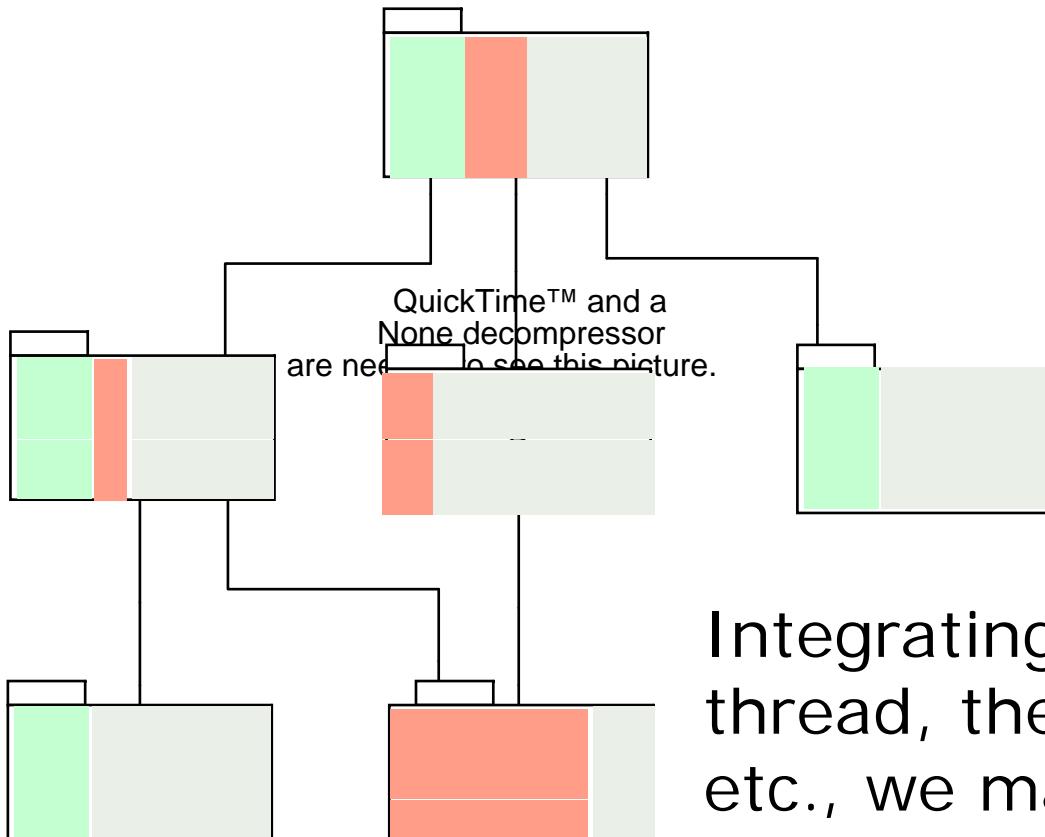
Sandwich integration
is flexible and
adaptable, but
complex to plan



Thread ...

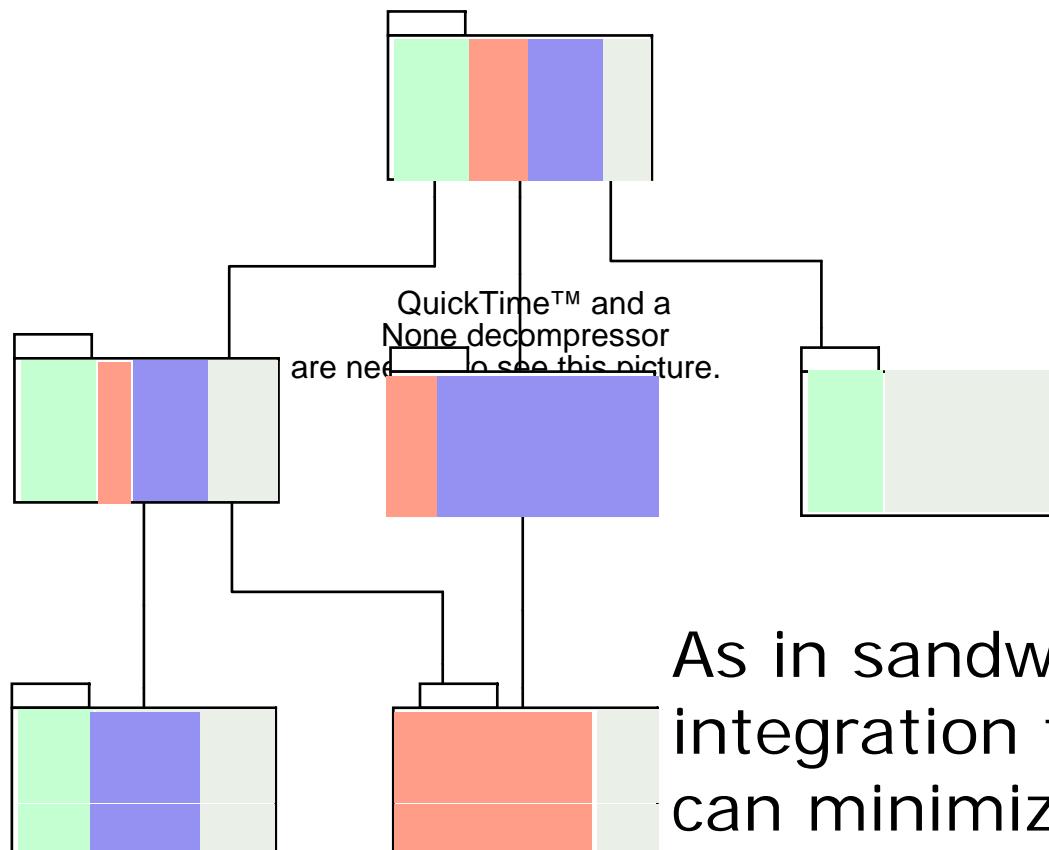


Thread ...

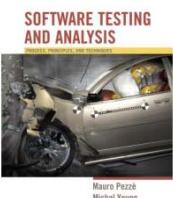


Integrating one thread, then another, etc., we maximize visibility for the user

Thread ...

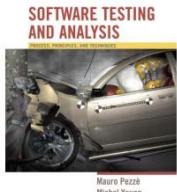


As in sandwich integration testing, we can minimize stubs and drivers, but the integration plan may be complex



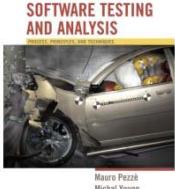
Critical Modules

- Strategy: Start with riskiest modules
 - Risk assessment is necessary first step
 - May include technical risks (is X feasible?), process risks (is schedule for X realistic?), other risks
- May resemble thread or sandwich process in tactics for flexible build order
 - E.g., constructing parts of one module to test functionality in another
- Key point is risk-oriented process
 - Integration testing as a risk-reduction activity, designed to deliver any bad news as early as possible



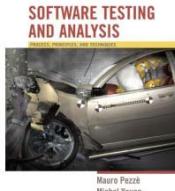
Choosing a Strategy

- Functional strategies require more planning
 - Structural strategies (bottom up, top down, sandwich) are simpler
 - But thread and critical modules testing provide better process visibility, especially in complex systems
- Possible to combine
 - Top-down, bottom-up, or sandwich are reasonable for relatively small components and subsystems
 - Combinations of thread and critical modules integration testing are often preferred for larger subsystems



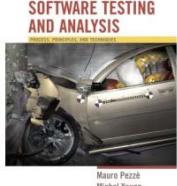
Working Definition of *Component*

- Reusable unit of deployment and composition
 - Deployed and integrated multiple times
 - Integrated by different teams (usually)
 - Component producer is distinct from component user
- Characterized by an *interface or contract*
 - Describes access points, parameters, and all functional and non-functional behavior and conditions for using the component
 - No other access (e.g., source code) is usually available
- Often larger grain than objects or packages
 - Example: A complete database system may be a component



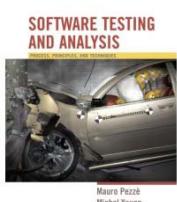
Components — Related Concepts

- Framework
 - Skeleton or micro-architecture of an application
 - May be packaged and reused as a component, with “hooks” or “slots” in the interface contract
- Design patterns
 - Logical design fragments
 - Frameworks often implement patterns, but patterns are not frameworks. Frameworks are concrete, patterns are abstract
- Component-based system
 - A system composed primarily by assembling components, often “Commercial off-the-shelf” (COTS) components
 - Usually includes application-specific “glue code”



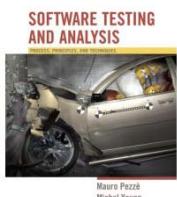
Component Interface Contracts

- Application programming interface (API) is distinct from implementation
 - Example: DOM interface for XML is distinct from many possible implementations, from different sources
- Interface includes *everything* that must be known to use the component
 - More than just method signatures, exceptions, etc
 - May include non-functional characteristics like performance, capacity, security
 - May include dependence on other components



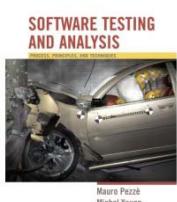
Challenges in Testing Components

- The component builder's challenge:
 - Impossible to know all the ways a component may be used
 - Difficult to recognize and specify all potentially important properties and dependencies
- The component user's challenge:
 - No visibility "inside" the component
 - Often difficult to judge suitability for a particular use and context



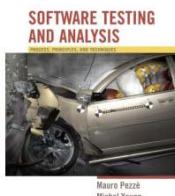
Testing a Component: Producer View

- First: Thorough unit and subsystem testing
 - Includes thorough functional testing based on application program interface (API)
 - Rule of thumb: Reusable component requires at least twice the effort in design, implementation, and testing as a subsystem constructed for a single use (often more)
- Second: Thorough acceptance testing
 - Based on scenarios of expected use
 - Includes stress and capacity testing
 - Find and document the limits of applicability

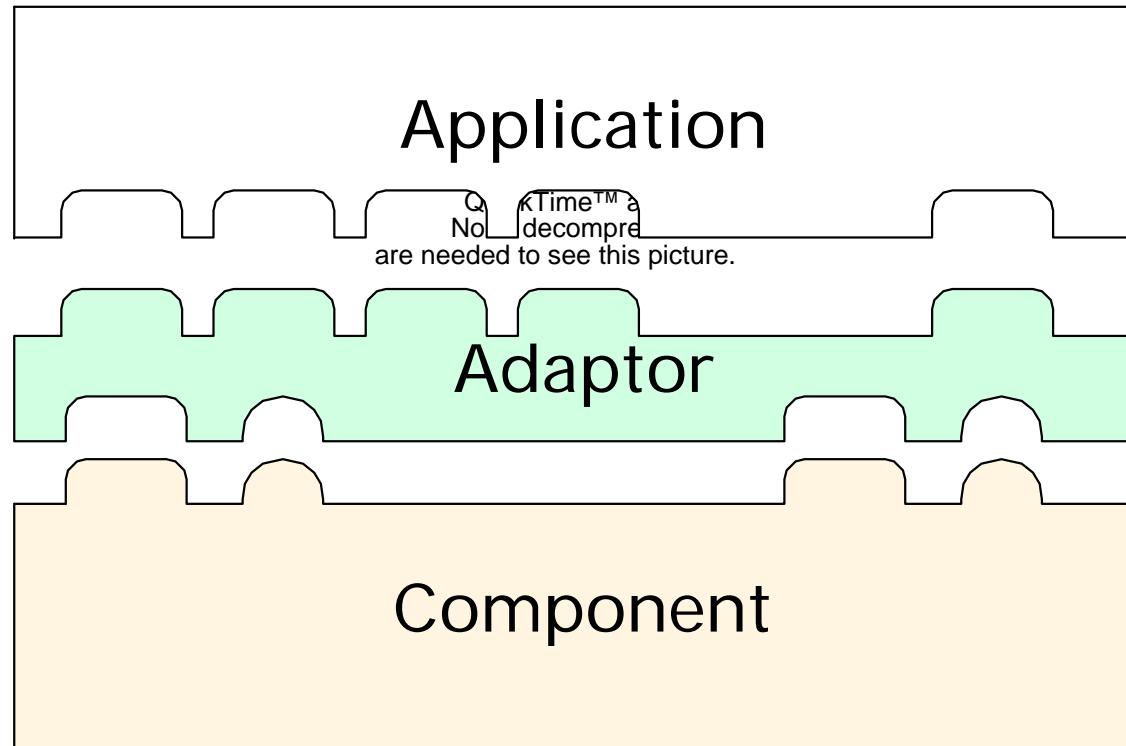


Testing a Component: User View

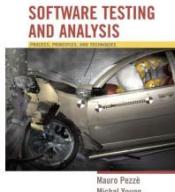
- Not primarily to find faults in the component
- Major question: Is the component suitable for *this* application?
 - Primary risk is not fitting the application context:
 - Unanticipated dependence or interactions with environment
 - Performance or capacity limits
 - Missing functionality, misunderstood API
 - Risk high when using component for first time
- Reducing risk: Trial integration early
 - Often worthwhile to build driver to test model scenarios, long before actual integration



Adapting and Testing a Component

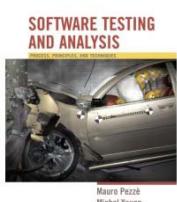


- Applications often access components through an adaptor, which can also be used by a test driver



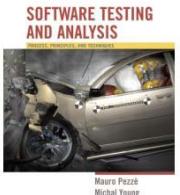
Summary

- Integration testing focuses on interactions
 - Must be built on foundation of thorough unit testing
 - Integration faults often traceable to incomplete or misunderstood interface specifications
 - Prefer prevention to detection, and make detection easier by imposing design constraints
- Strategies tied to project *build order*
 - Order construction, integration, and testing to reduce cost or risk
- Reusable components require special care
 - For component builder, and for component user



Testing Object Oriented Software

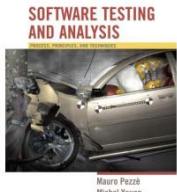
Chapter 15



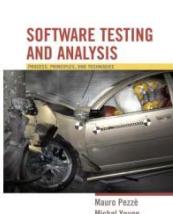
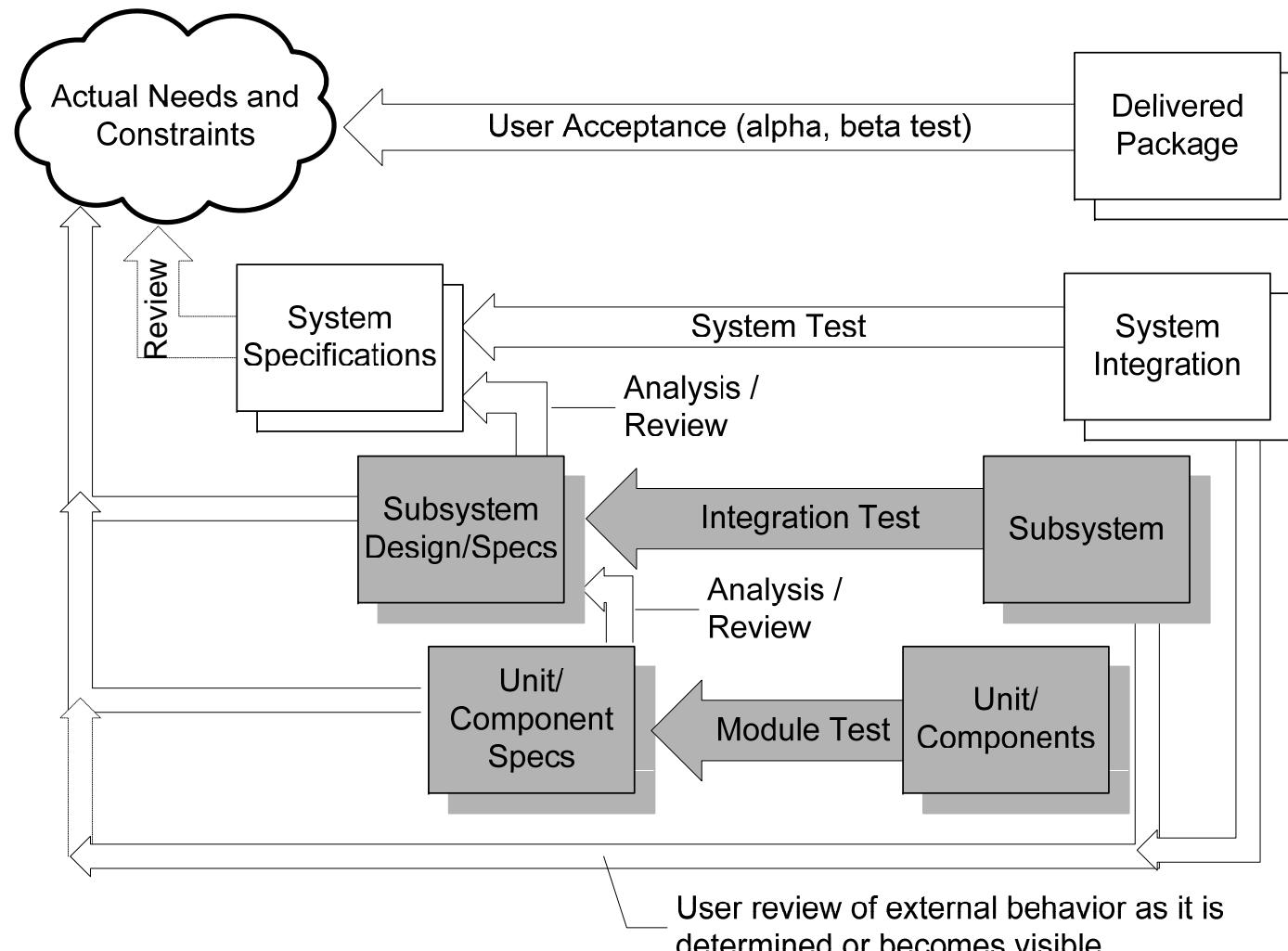
Characteristics of OO Software

Typical OO software characteristics that impact testing

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

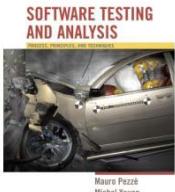


Quality activities and OO SW

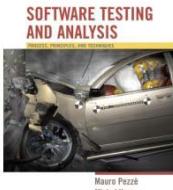
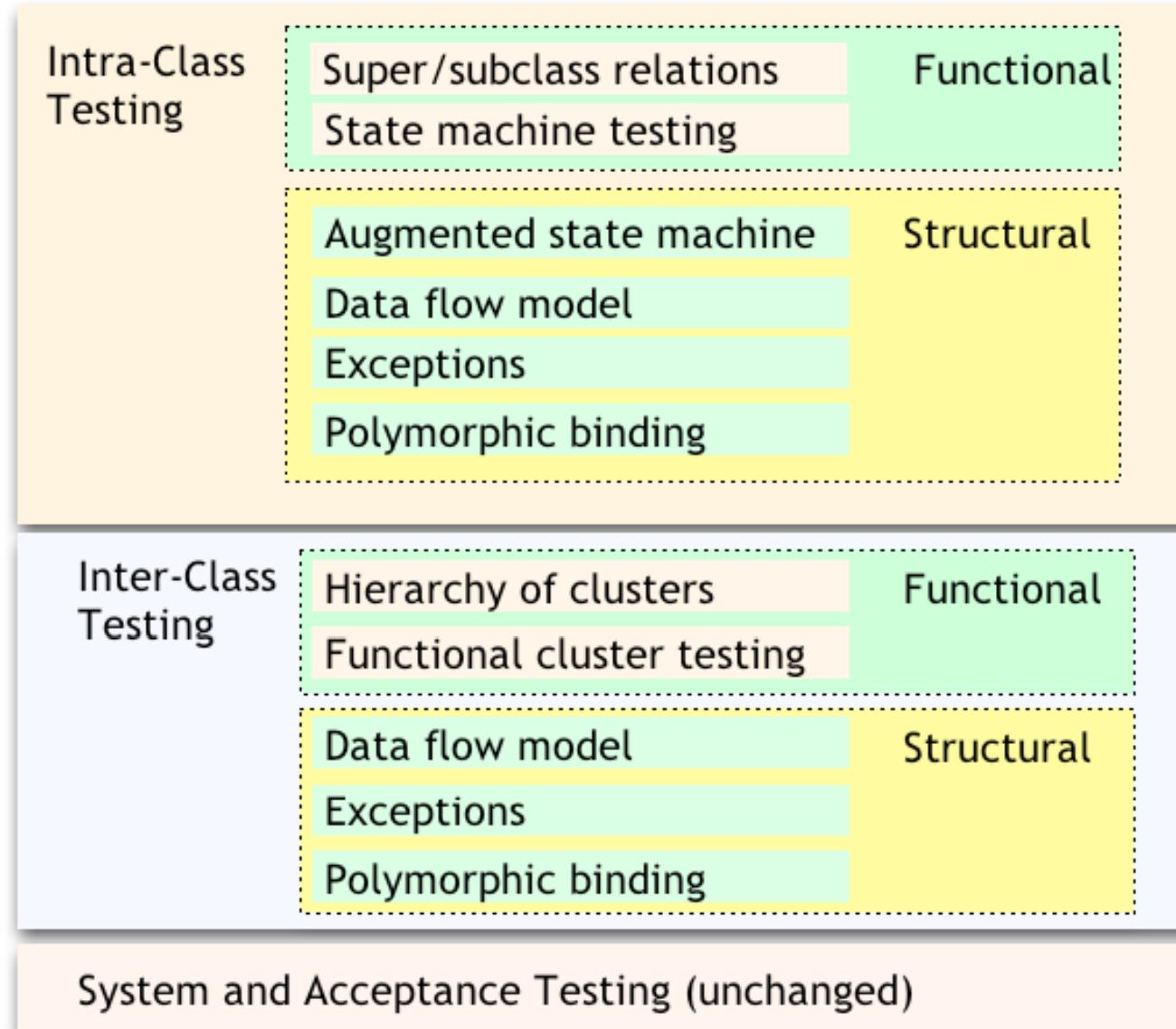


OO definitions of unit and integration testing

- Procedural software
 - unit = single program, function, or procedure
more often: a unit of work that may correspond to one or more intertwined functions or programs
- Object oriented software
 - unit = class or (small) cluster of strongly related classes (e.g., sets of Java classes that correspond to exceptions)
 - unit testing = **intra-class testing**
 - integration testing = **inter-class testing** (cluster of classes)
 - dealing with single methods separately is usually too expensive (complex scaffolding), so methods are usually tested in the context of the class they belong to

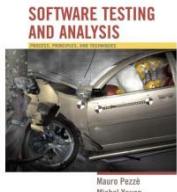


Orthogonal approach: Stages



Intraclass State Machine Testing

- Basic idea:
 - The state of an object is modified by operations
 - Methods can be modeled as state transitions
 - Test cases are sequences of method calls that traverse the state machine model
- State machine model can be derived from specification (functional testing), code (structural testing), or both



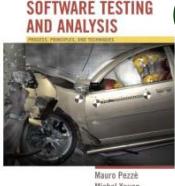
[Later: Inheritance and dynamic binding]

Informal state-full specifications

Slot: represents a slot of a computer model.

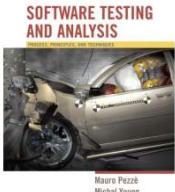
.... slots can be bound or unbound. Bound slots are assigned a compatible component, unbound slots are empty. Class slot offers the following services:

- **Install:** slots can be installed on a model as *required* or *optional*.
...
- **Bind:** slots can be bound to a compatible component.
...
- **Unbind:** bound slots can be unbound by removing the bound component.
- **IsBound:** returns the current binding, if bound; otherwise returns the special value *empty*.

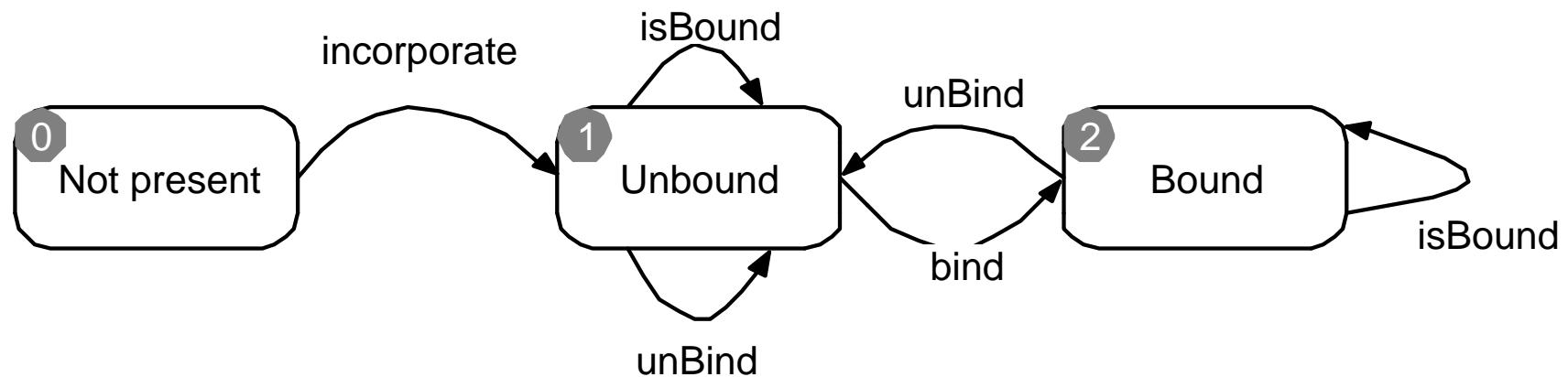


Identifying states and transitions

- From the informal specification we can identify three states:
 - Not_installed
 - Unbound
 - Bound
- and four transitions
 - install: from Not_installed to Unbound
 - bind: from Unbound to Bound
 - unbind: ...to Unbound
 - isBound: does not change state



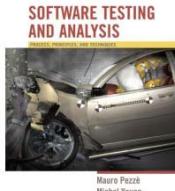
Deriving an FSM and test cases



- TC-1: incorporate, isBound, bind, isBound
- TC-2: incorporate, unBind, bind, unBind, isBound

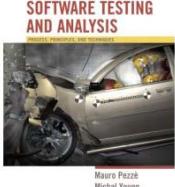
Testing with State Diagrams

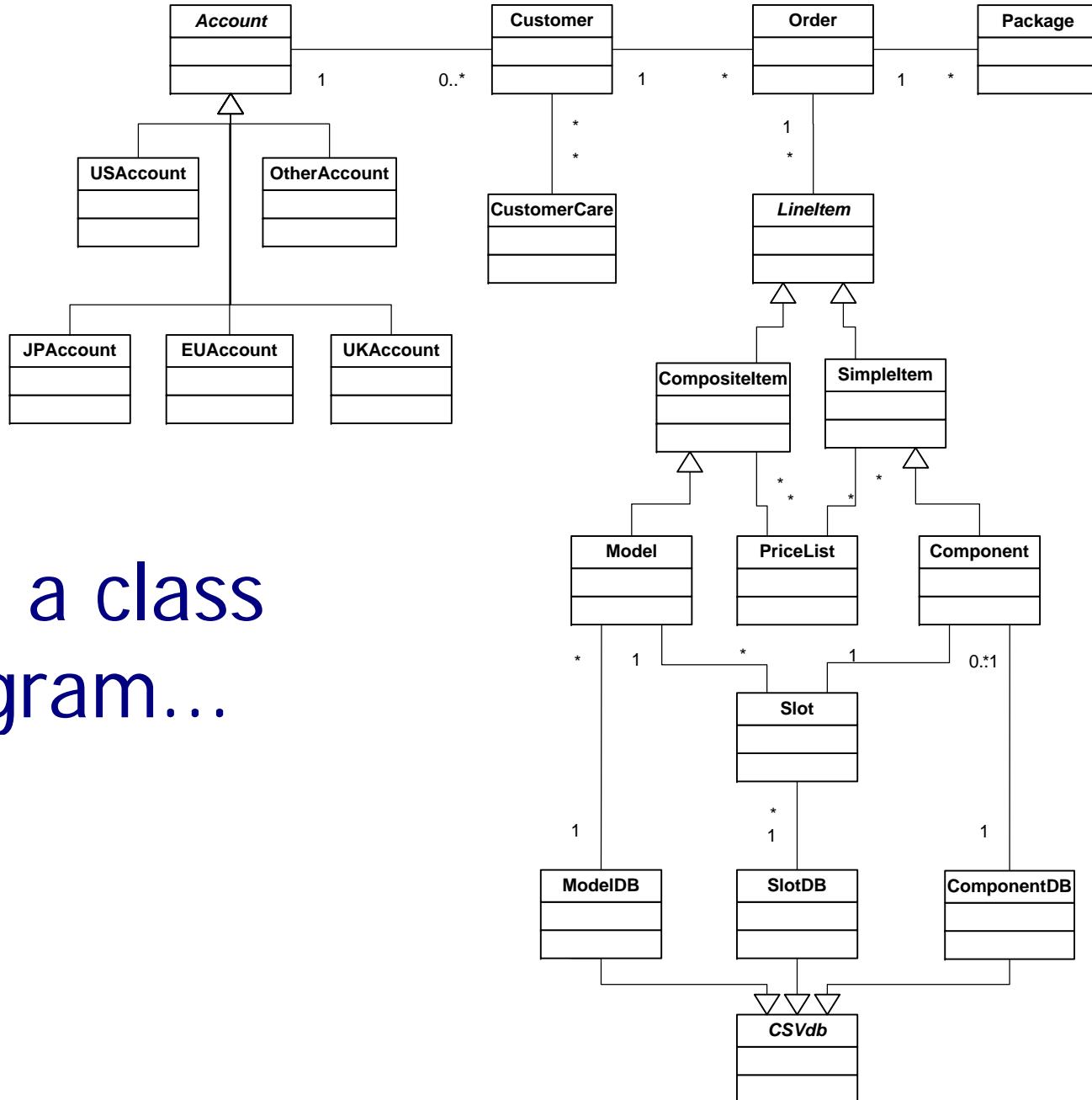
- A statechart (called a “state diagram” in UML) may be produced as part of a specification or design
 - May also be implied by a set of message sequence charts (interaction diagrams), or other modeling formalisms
- Two options:
 - Convert (“flatten”) into standard finite-state machine, then derive test cases
 - Use state diagram model directly



Interclass Testing

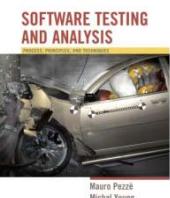
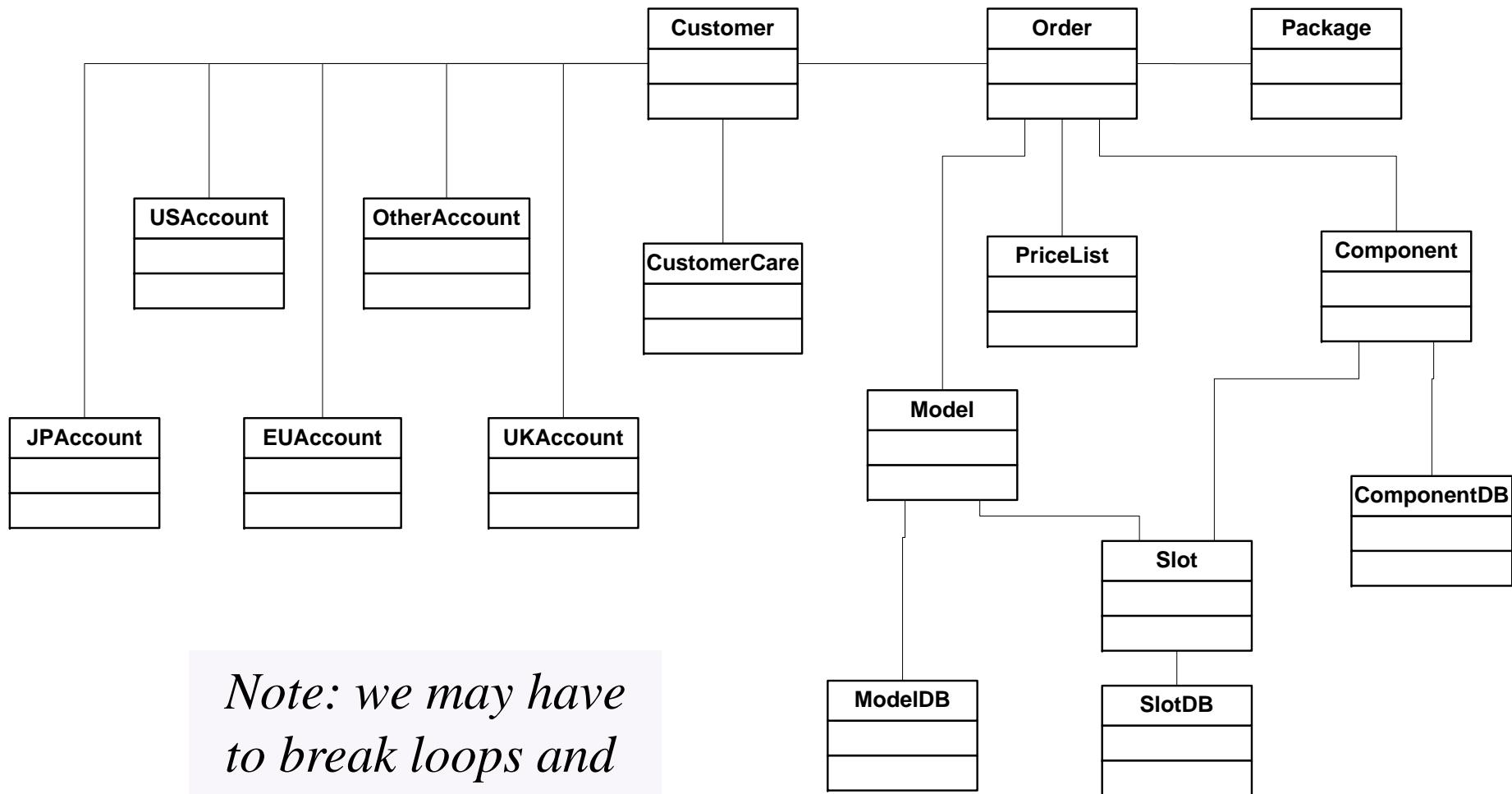
- The first level of *integration testing* for object-oriented software
 - Focus on interactions between classes
- Bottom-up integration according to “depends” relation
 - A depends on B: Build and test B, then A
- Start from use/include hierarchy
 - Implementation-level parallel to logical “depends” 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”





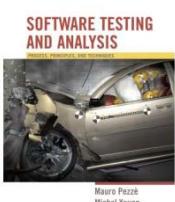
from a class
diagram...

....to a hierarchy

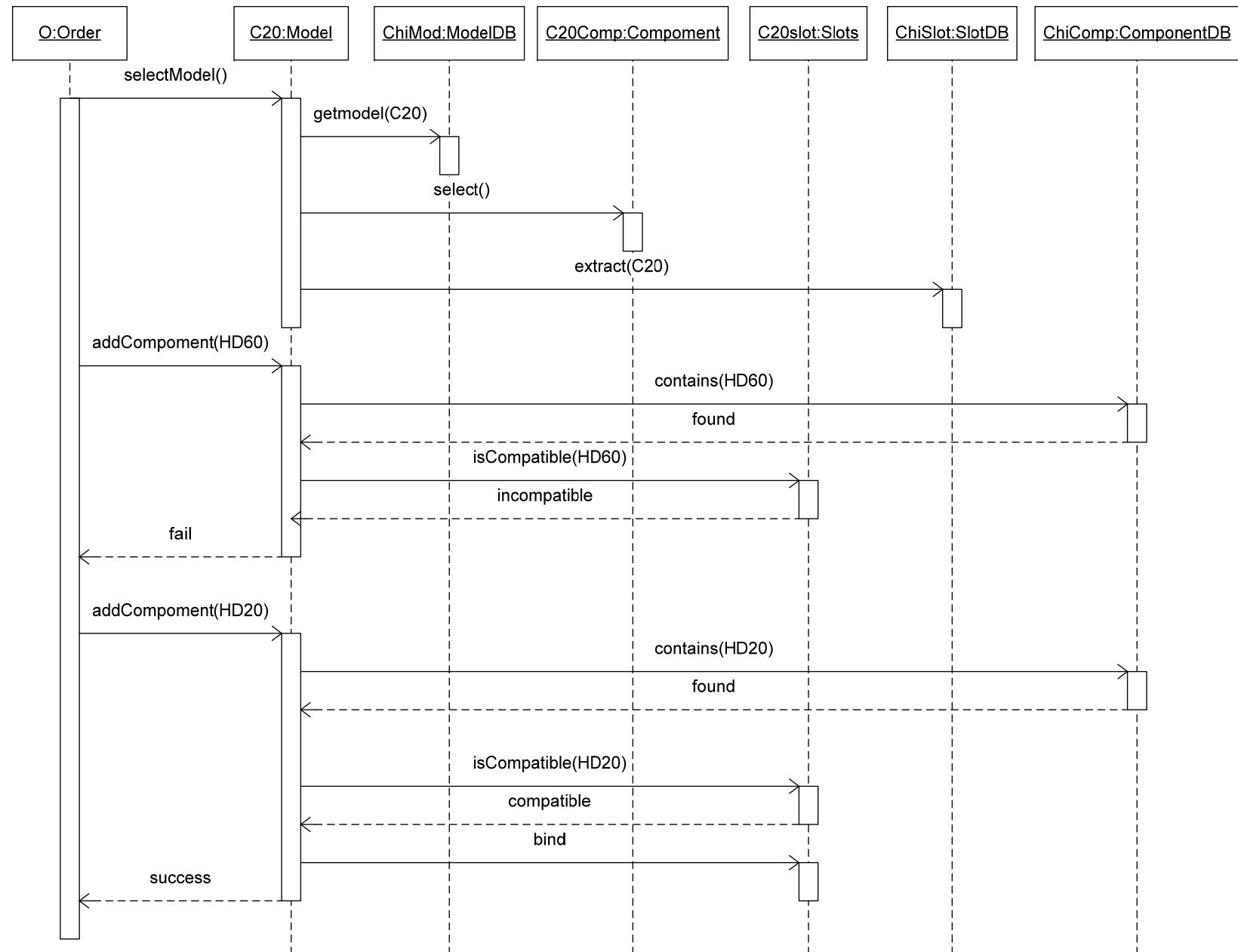


Interactions in Interclass Tests

- Proceed bottom-up
- Consider all combinations of interactions
 - example: a test case for class *Order* includes a call to a method of class *Model*, and the called method calls a method of class *Slot*, exercise all possible relevant states of the different classes
 - problem: combinatorial explosion of cases
 - so select a subset of interactions:
 - arbitrary or random selection
 - plus all significant interaction scenarios that have been previously identified in design and analysis: sequence + collaboration diagrams

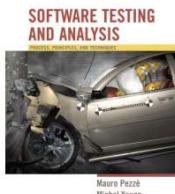


sequence diagram



Using Structural Information

- Start with functional testing
 - As for procedural software, the specification (formal or informal) is the first source of information for testing object-oriented software
 - “Specification” widely construed: Anything from a requirements document to a design model or detailed interface description
- Then add information from the code (structural testing)
 - Design and implementation details not available from other sources

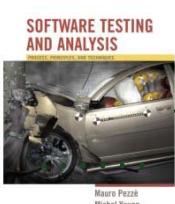


From the implementation ...

```
public class Model extends Orders.CompositeItem {  
    ....  
    private boolean legalConfig = false; // memoized  
    ....  
    public boolean isLegalConfiguration() {  
        if (! legalConfig) {  
            checkConfiguration();  
        }  
        return legalConfig;  
    }  
    ....  
    private void checkConfiguration() {  
        legalConfig = true;  
        for (int i=0; i < slots.length; ++i) {  
            Slot slot = slots[i];  
            if (slot.required && ! slot.isBound()) {  
                legalConfig = false;  
            }  
        }  
    }  
}
```

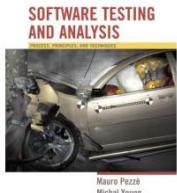
private instance variable

private method



Intraclass 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

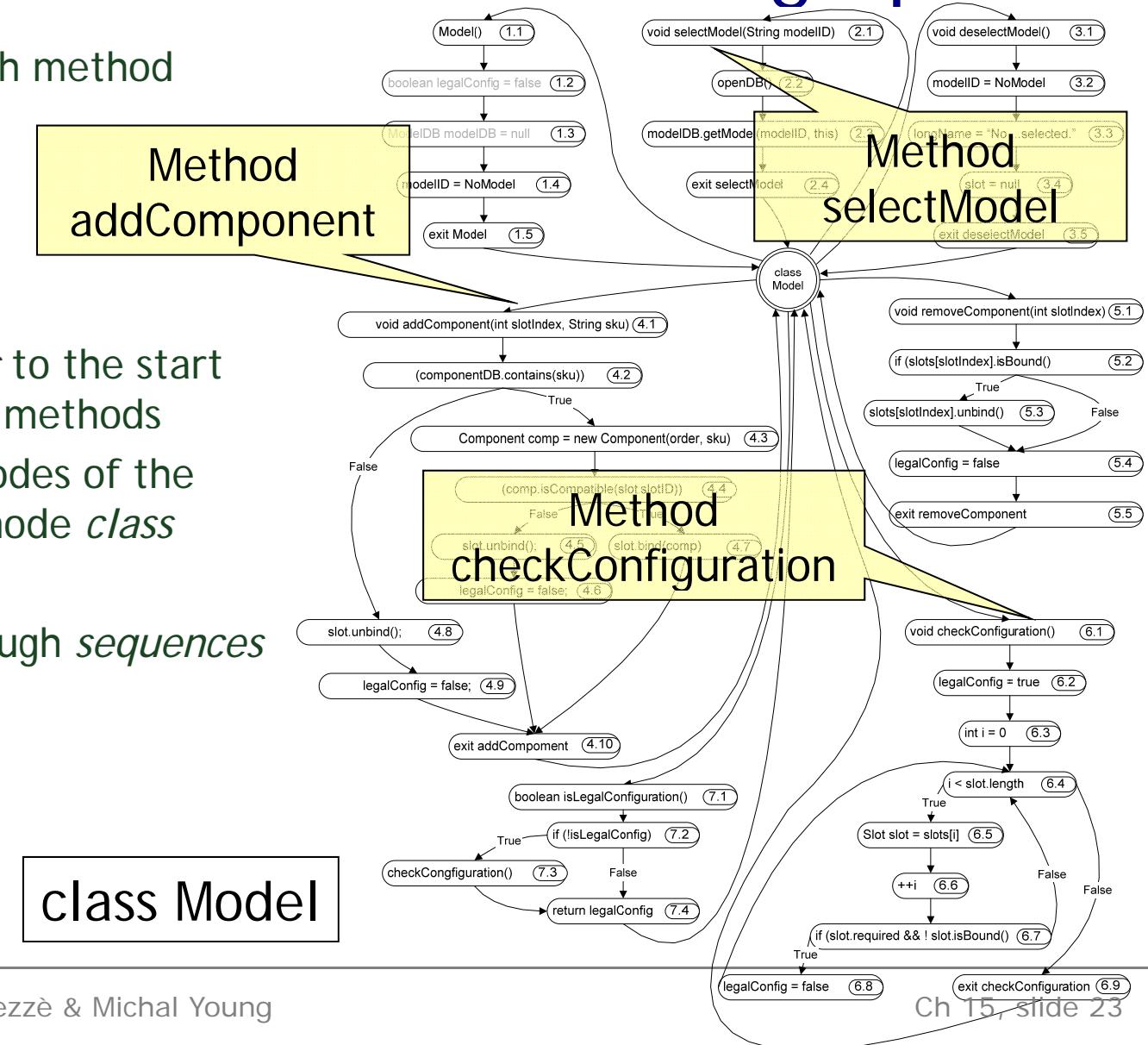
Control flow for each method

+
node for class
+

edges

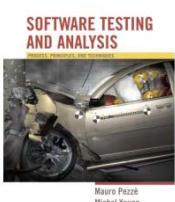
from node *class* to the start
nodes of the methods
from the end nodes of the
methods to node *class*

=> control flow through *sequences*
of method calls



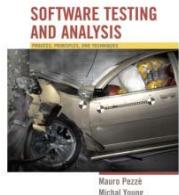
Interclass structural testing

- Working “bottom up” in dependence hierarchy
 - Dependence is not the same as class hierarchy; not always the same as call or inclusion relation.
 - May match bottom-up build order
 - Starting from leaf classes, then classes that use leaf classes, ...
- Summarize effect of each method: Changing or using object state, or both
 - Treating a whole object as a variable (not just primitive types)

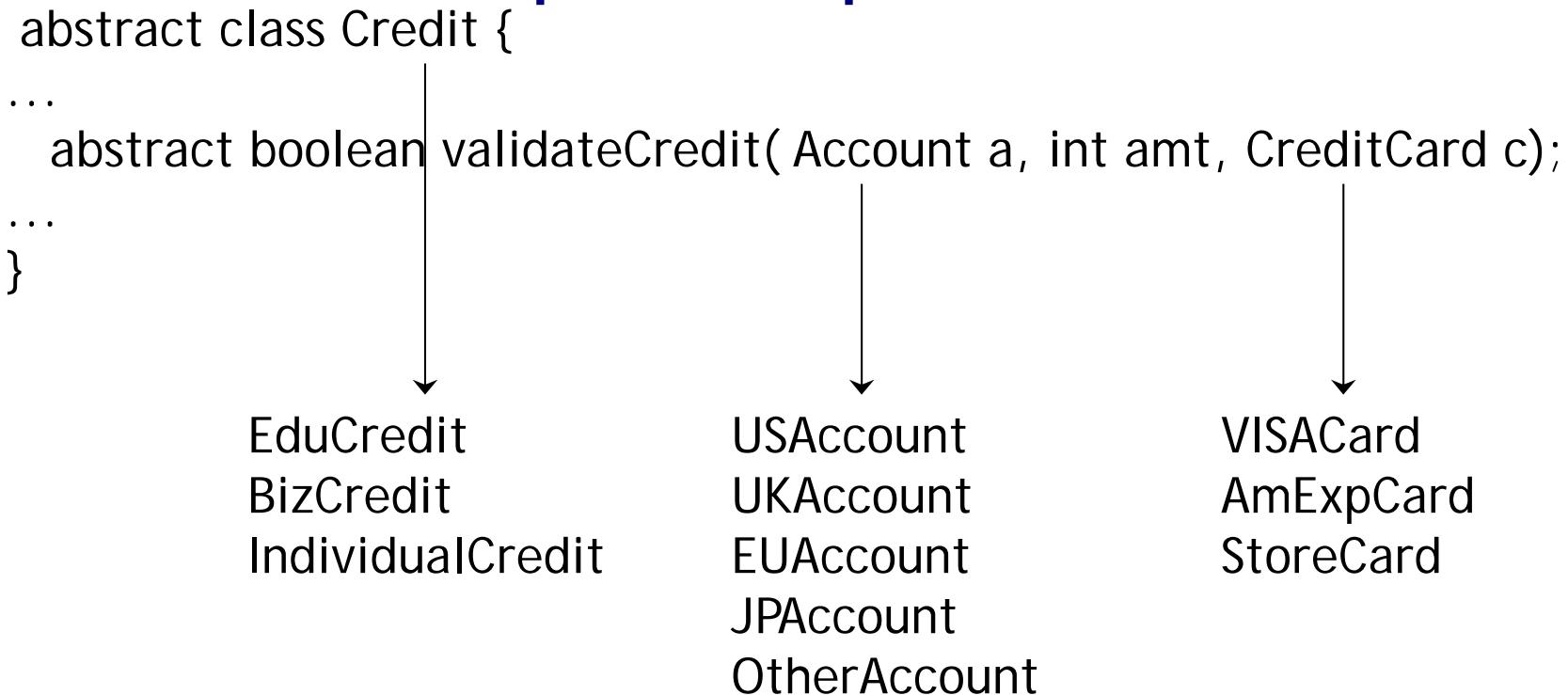


Polymorphism and dynamic binding

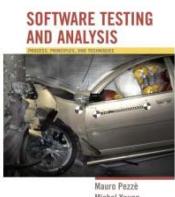
One variable potentially bound to
methods of different (sub-)classes



“Isolated” calls: the combinatorial explosion problem



The combinatorial problem: $3 \times 5 \times 3 = 45$ possible combinations of dynamic bindings (just for this one method!)



The combinatorial approach

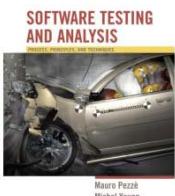
Identify a set of combinations that cover all pairwise combinations of dynamic bindings

Same motivation as pairwise specification-based testing

Account	Credit	creditCard
USAccount	EduCredit	VISACard
USAccount	BizCredit	AmExpCard
USAccount	individualCredit	ChipmunkCard
UKAccount	EduCredit	AmExpCard
UKAccount	BizCredit	VISACard
UKAccount	individualCredit	ChipmunkCard
EUAccount	EduCredit	ChipmunkCard
EUAccount	BizCredit	AmExpCard
EUAccount	individualCredit	VISACard
JPAccount	EduCredit	VISACard
JPAccount	BizCredit	ChipmunkCard
JPAccount	individualCredit	AmExpCard
OtherAccount	EduCredit	ChipmunkCard
OtherAccount	BizCredit	VISACard
OtherAccount	individualCredit	AmExpCard

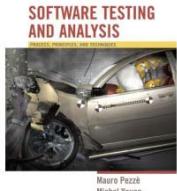
Inheritance

- When testing a subclass ...
 - We would like to re-test only what has not been thoroughly tested in the parent class
 - for example, no need to test hashCode and getClass methods inherited from class Object in Java
 - But we should test any method whose behavior may have changed
 - even accidentally!

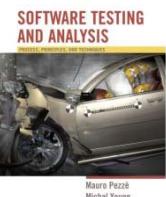
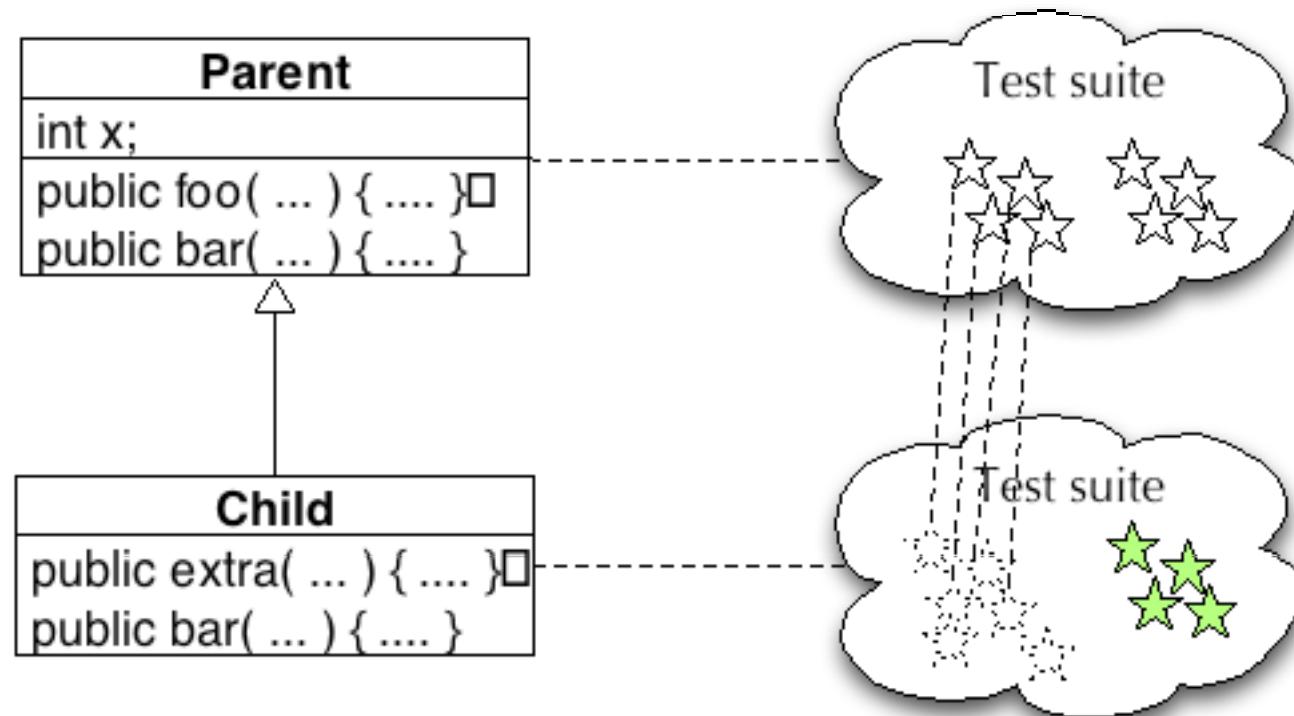


Reusing Tests with the Testing History Approach

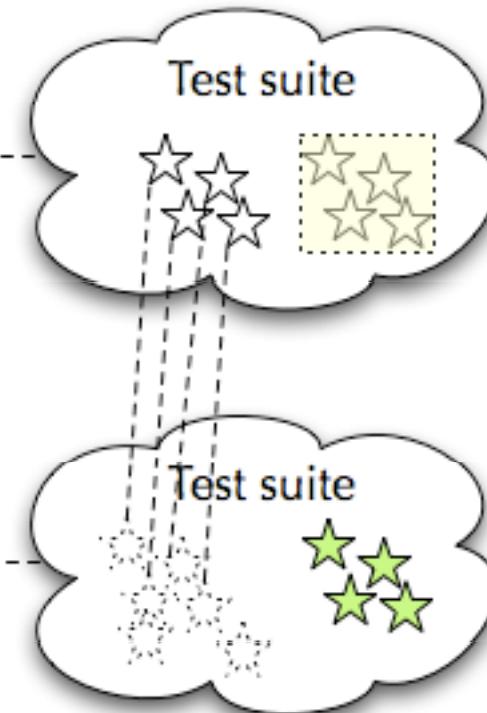
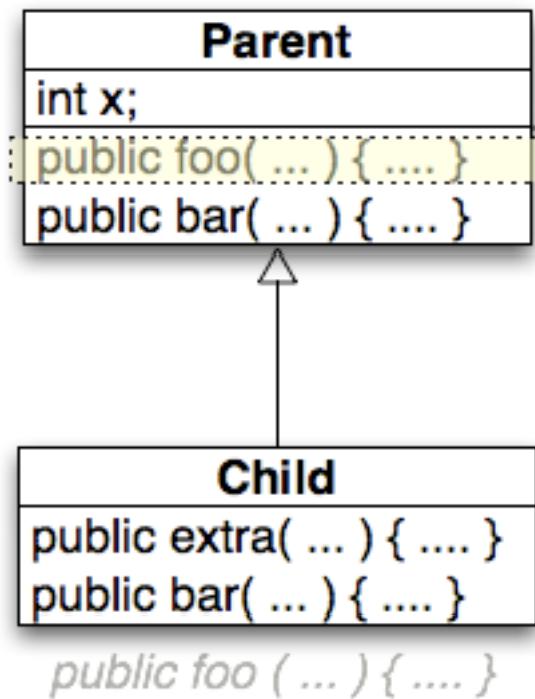
- Track test suites and test executions
 - determine which new tests are needed
 - determine which old tests must be re-executed
- New and changed behavior ...
 - new methods must be tested
 - redefined methods must be tested, but we can partially reuse test suites defined for the ancestor
 - other inherited methods do not have to be retested



Testing history

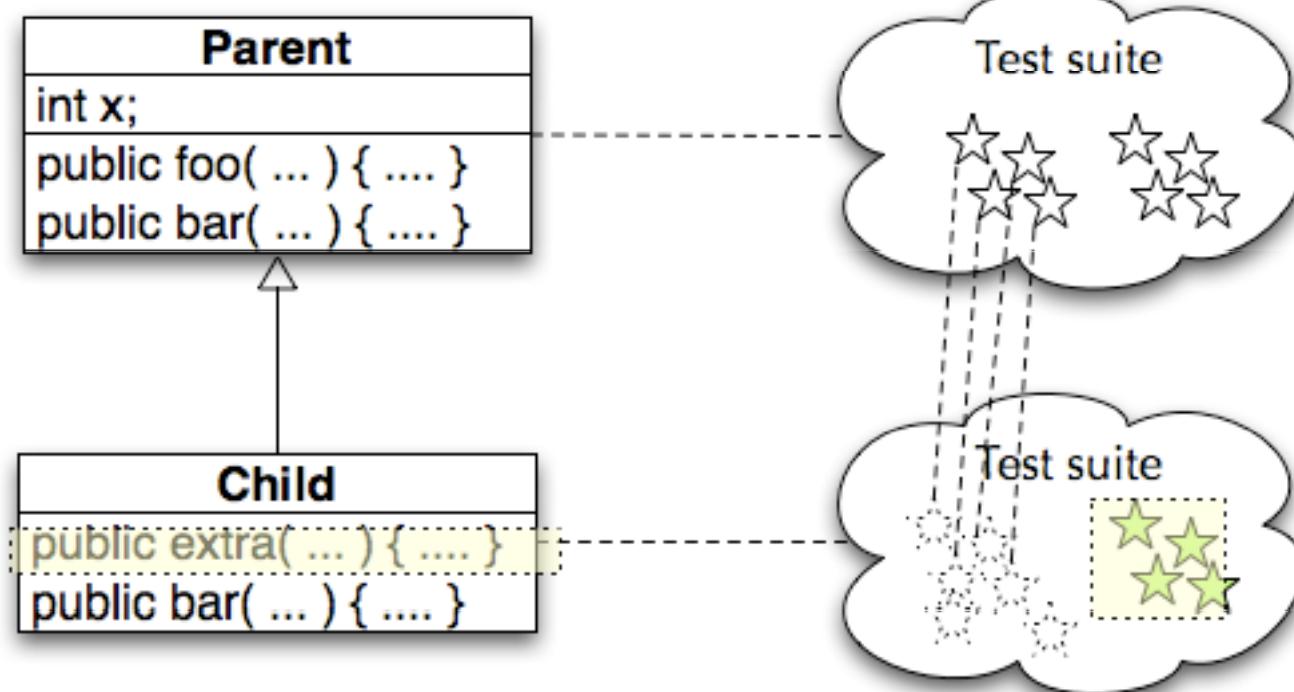


Inherited, unchanged



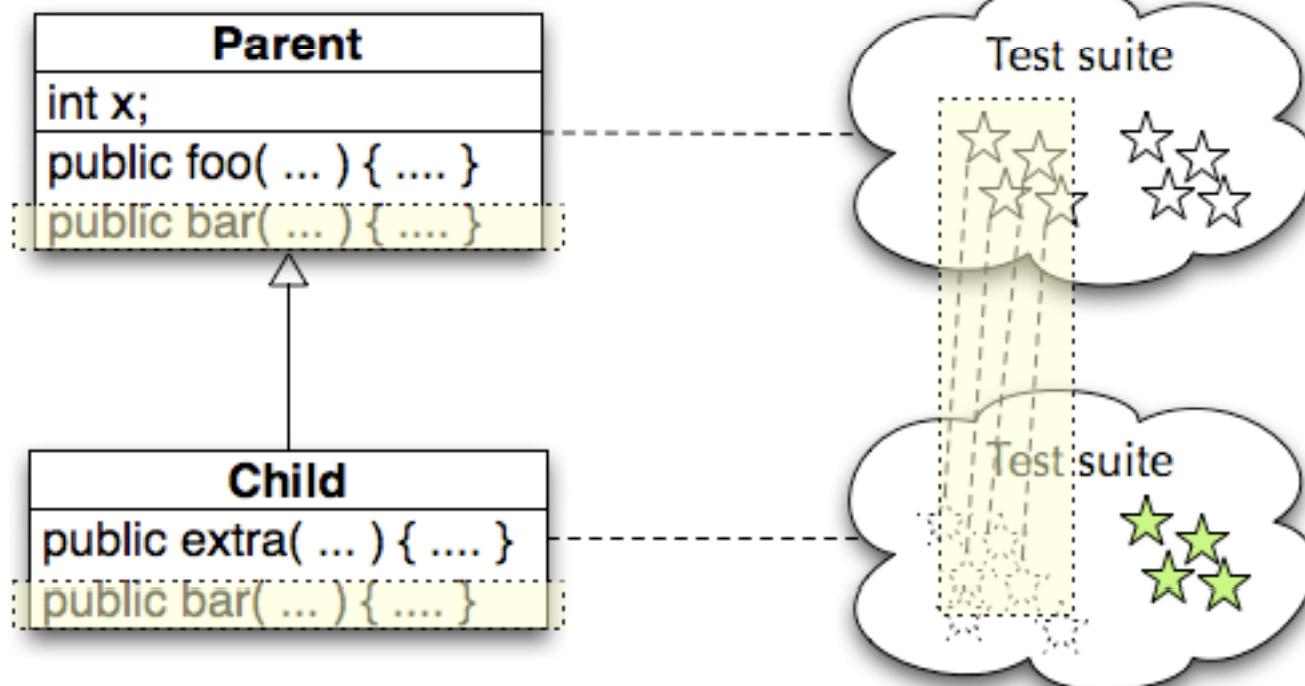
Inherited, unchanged ("recursive"):
No need to re-test

Newly introduced methods



New:
Design and execute new test cases

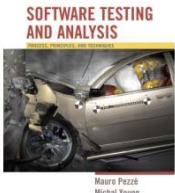
Overridden methods



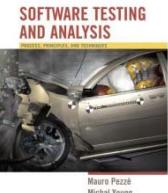
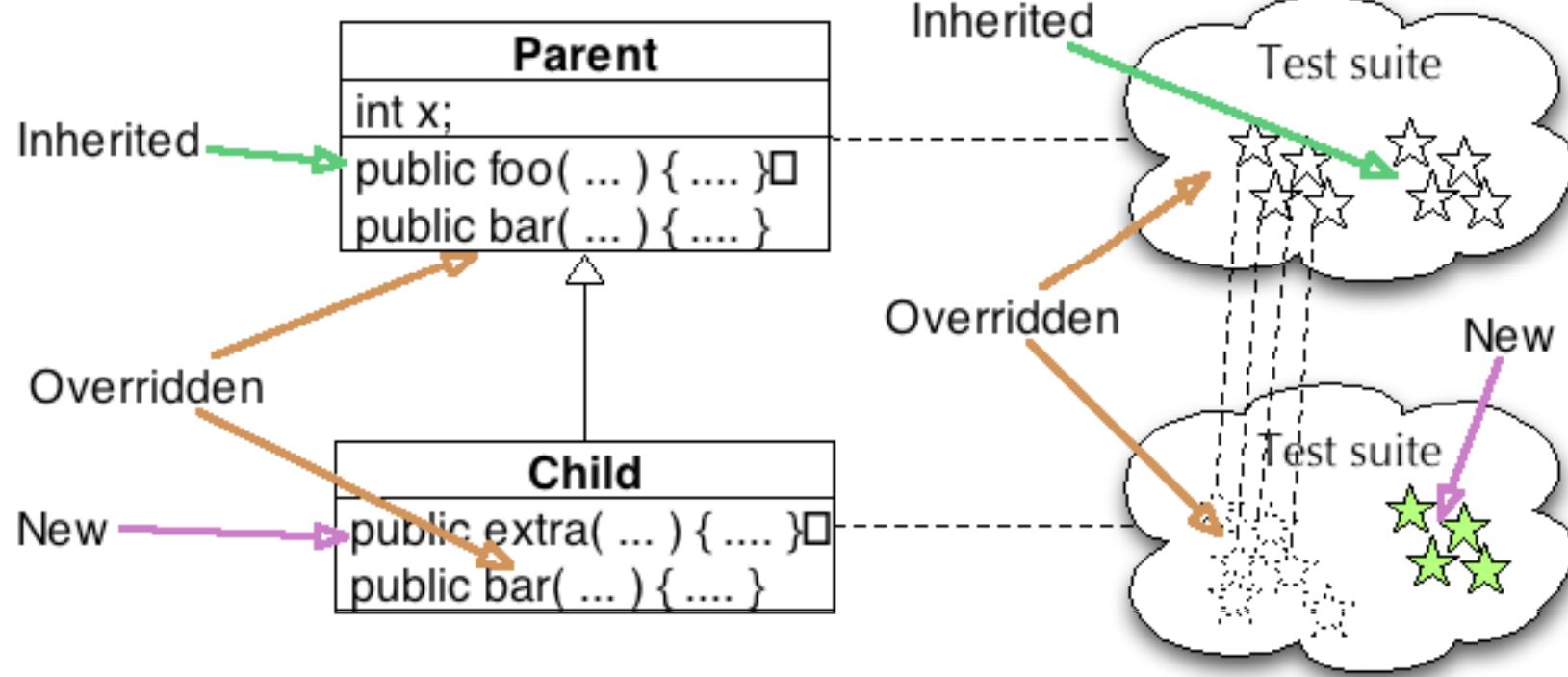
Overridden:
Re-execute test cases from parent,
add new test cases as needed

Testing History – some details

- Abstract methods (and classes)
 - Design test cases when abstract method is introduced (even if it can't be executed yet)
- Behavior changes
 - Should we consider a method “redefined” if another new or redefined method changes its behavior?
 - The standard “testing history” approach does not do this
 - It might be reasonable combination of data flow (structural) OO testing with the (functional) testing history approach

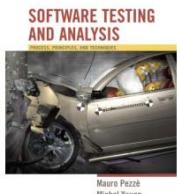


Testing History - Summary



Does testing history help?

- Executing test cases should (usually) be cheap
 - It may be simpler to re-execute the full test suite of the parent class
 - ... but still add to it for the same reasons
- But sometimes execution is not cheap ...
 - Example: Control of physical devices
 - Or very large test suites
 - Ex: Some Microsoft product test suites require more than one night (so daily build cannot be fully tested)
 - Then some use of testing history is profitable



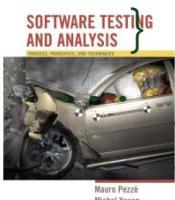
Exception handling

```

void addCustomer(Customer theCust) {
    customers.add(theCust);
}
public static Account
newAccount(...) {
    throws InvalidRegionException
{
    Account thisAccount = null;
    String regionAbbrev = Regions.regionOfCountry(
        mailAddress.getCountry());
    if (regionAbbrev == Regions.US) {
        thisAccount = new USAccount();
    } else if (regionAbbrev == Regions.UK) {
        ....
    } else if (regionAbbrev == Regions.Invalid) {
        throw new
    InvalidRegionException(mailAddress.getCountry());
}

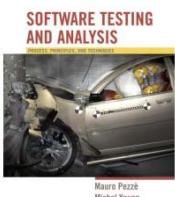
```

exceptions
create implicit
control flows
and may be
handled by
different
handlers



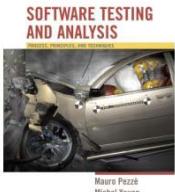
Testing exception handling

- Impractical to treat exceptions like normal flow
 - too many flows: every array subscript reference, every memory allocation, every cast, ...
 - multiplied by matching them to every handler that could appear immediately above them on the call stack.
 - many actually impossible
- So we separate testing exceptions
 - and ignore program error exceptions (test to prevent them, not to handle them)
- What we do test: Each exception handler, and each explicit throw or re-throw of an exception



Summary

- Several features of object-oriented languages and programs impact testing
 - from encapsulation and state-dependent structure to generics and exceptions
 - but only at unit and subsystem levels
 - and fundamental principles are still applicable
- Basic approach is orthogonal
 - Techniques for each major issue (e.g., exception handling, generics, inheritance, ...) can be applied incrementally and independently



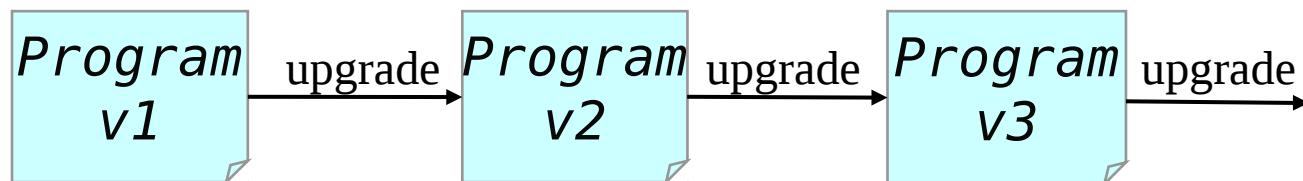
Regression Testing

Ajitha Rajan

Evolving Software

Large software systems are usually built incrementally:

- **Maintenance** - fixing errors and flaws, hardware changes
- **Enhancements** - new functionality, improved efficiency, extension, new regulations



Regressions

- Ideally, software should *improve* over time.
- But changes can both
 - **Improve** software, adding features and fixing bugs
 - **Break** software, introducing new bugs
- We call such breaking changes *regressions*

Regression Testing

Version 1

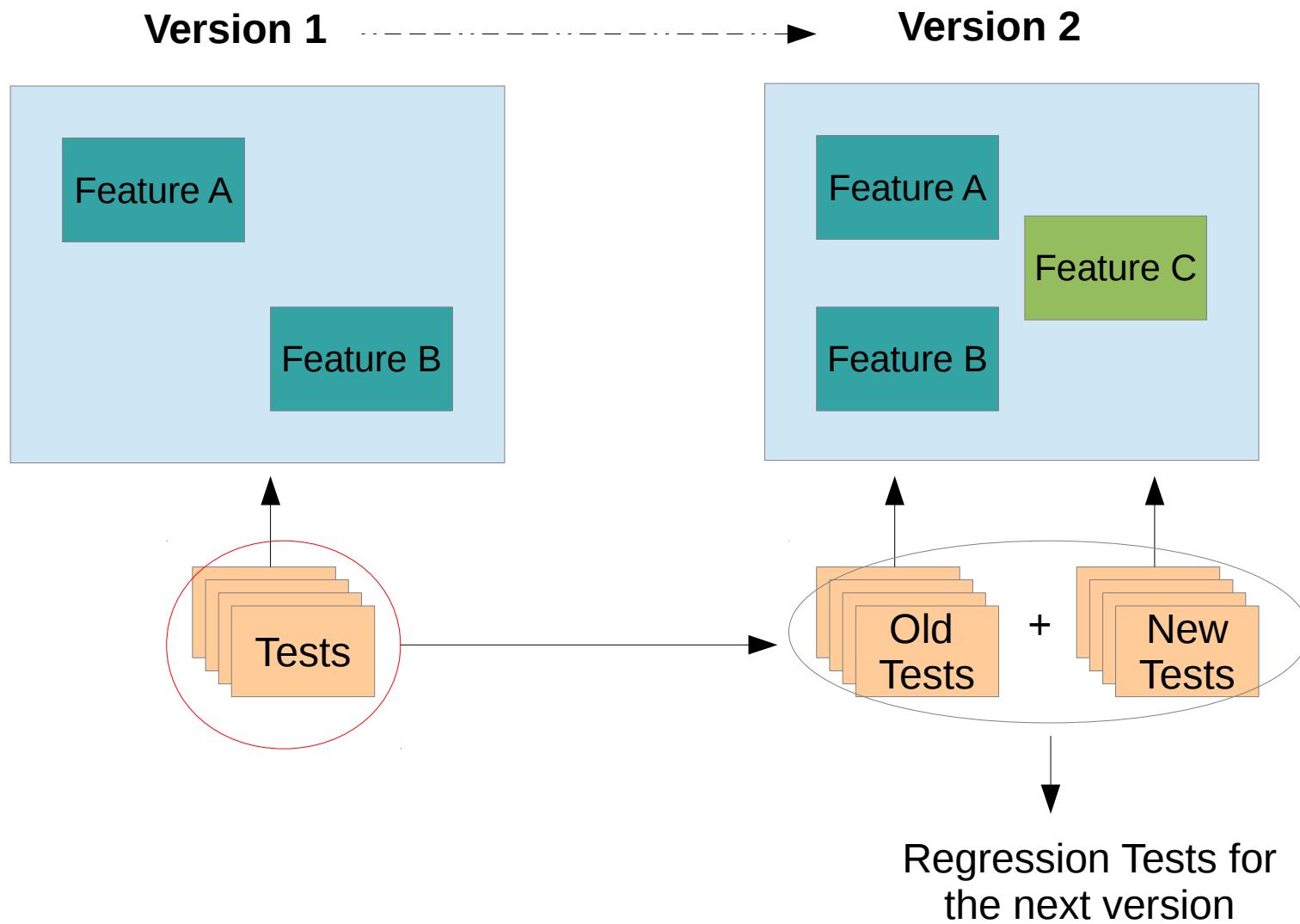
1. Develop P
2. Test P
3. Release P



Version 2

4. Modify P to P'
5. Test P' for *new functionality* or *bug fixing*
6. Perform **regression testing** on P'
7. Release P'

Example



Consequences of Poor Regression Testing

- Thousands of 1-800 numbers disabled by a poorly tested software upgrade (December 1991)
- Fault in an SS7 software patch causes extensive phone outages (June 1991)
- Fault in a 4ESS upgrade causes massive breakdown in the AT&T network (January 1990)

AT&T Network Outage, Jan 1990

```
1 While (ring receive buffer | empty and side buffer | empty)
2 {
3   Initialize pointer to first message in side buffer or ring received buffer
4   Get a copy of buffer
5   Switch (message) {
6     Case incoming message: if (sending switch = out of service)
7       {
8         if (ring write buffer = empty)
9           Send in service to states map manager;
10        Else
11          Break;
12        }
13      Process incoming message, set up pointers to optional parameters
14      Break;
15      :
16    }
17    Do optional parameter work
18 }
```

Regression

- Yesterday it worked, today it doesn't.
 - I was fixing X, and accidentally broke Y
- Tests must be re-run after any change
 - Adding new features
 - Changing, adapting software to new conditions
 - Fixing other bugs
- Regression testing can be a major cost of software maintenance
 - Sometimes much more than making the change

Regression Testing takes too long

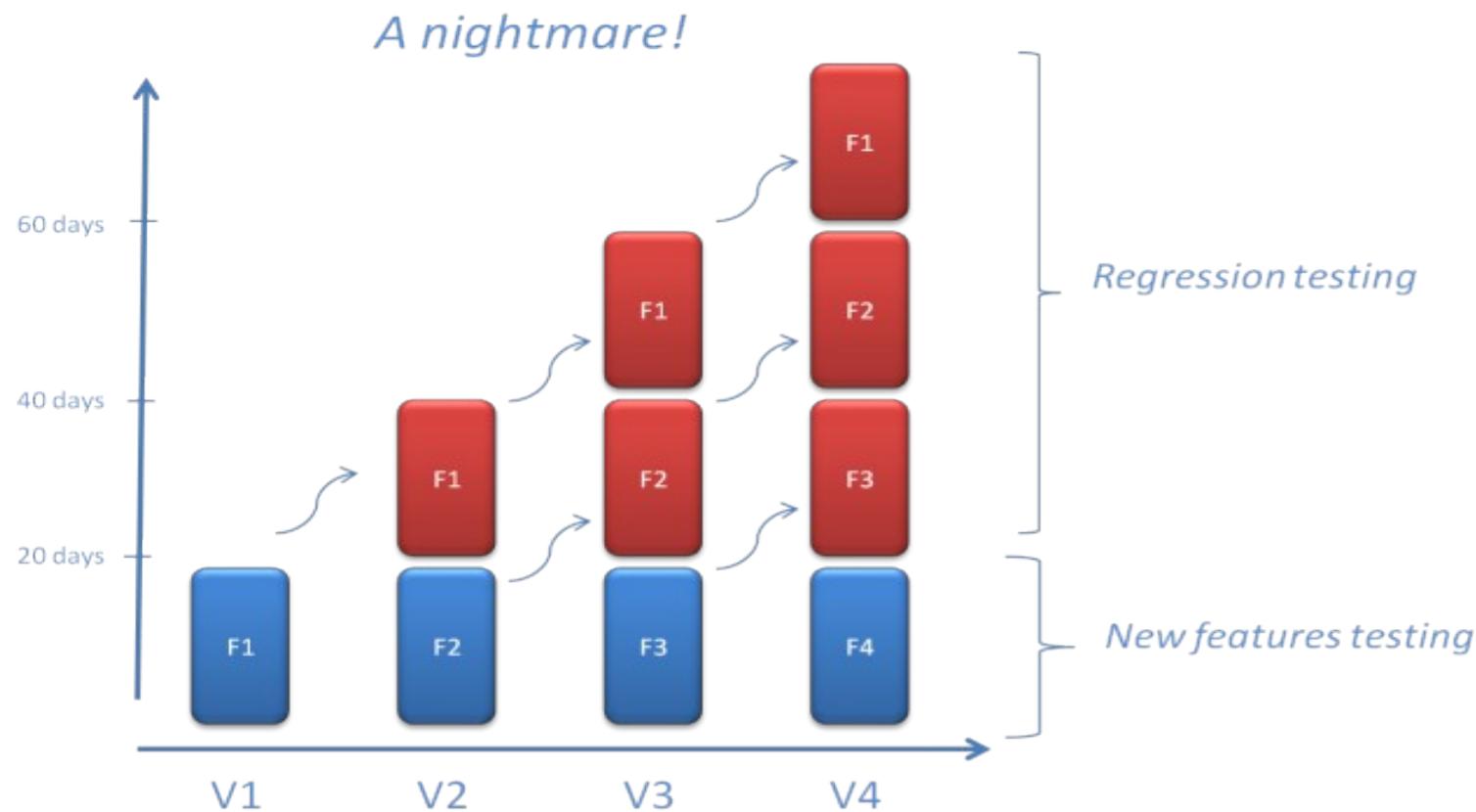


Image from <http://blog.kalistick.com/tools/improving-regression-testing-effectiveness/>

Basic Problems of Regression Test

- **Maintaining test suite**
 - If I change feature X, how many test cases must be revised because they use feature X?
 - Which test cases should be removed or replaced? Which test cases should be added?
- **Cost of re-testing**
 - Often proportional to product size, not change size
 - Big problem if testing requires manual effort
 - Possible problem even for automated testing, when the test suite and test execution time grows beyond a few hours

Test Case Maintenance

Some maintenance is inevitable

If feature X has changed, test cases for feature X will require updating

Some maintenance should be avoided

Example: Trivial changes to user interface or file format should not invalidate large numbers of test cases

Test suites should be modular!

Avoid unnecessary dependence

Generating concrete test cases from test case specifications can help

Obsolete and Redundant

- **Obsolete:** A test case that is no longer valid
 - Should be removed from the test suite
- **Redundant:** A test case that does not differ significantly from others
 - Unlikely to find a fault missed by similar test cases
 - Has some cost in re-execution
 - May or may not be removed, depending on costs

Regression Test Optimization

- Re-test All
- Regression Test Selection
- Regression Test Set Minimisation
- Regression Test Set Prioritisation

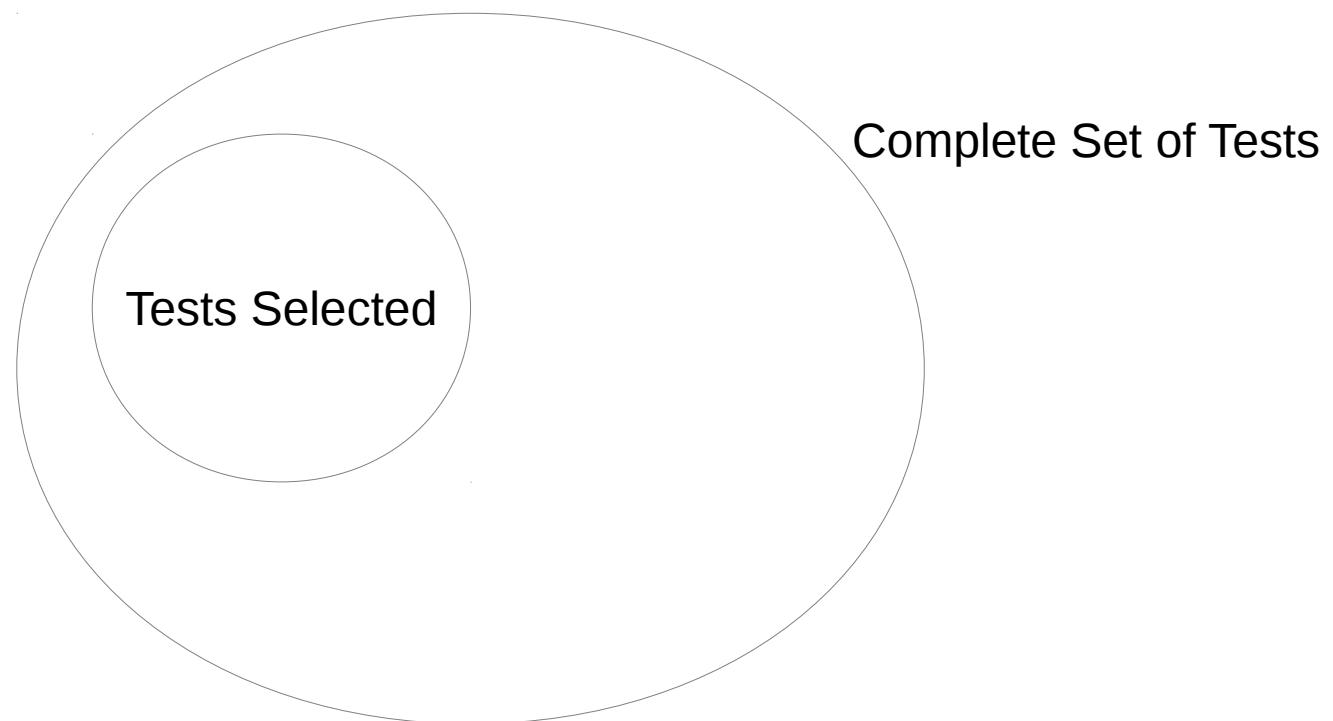
Re-test All Approach

- Traditional Approach – **Select All**
- The test-all approach is good when you want to be certain that the new version works on all tests developed for the previous version.
- What if you only have limited resources to run tests and have to meet a deadline?
- Those on which the **new and the old programs produce different outputs** (Undecidable)



Regression Test Selection

From the entire test suite, only select subset of test cases whose execution is relevant to changes



Code-based Regression Test Selection

- **Observation:** A test case can't find a fault in code it doesn't execute
 - In a large system, many parts of the code are untouched by many test cases
- **So:** Only execute test cases that execute changed or new code

Control-flow and Data-flow Regression Test Selection

- Same basic idea as code-based selection
 - Re-run test cases only if they include changed elements
 - Elements may be modified control flow nodes and edges, or definition-use (DU) pairs in data flow
- To automate selection:
 - Tools record elements touched by each test case
 - Stored in database of regression test cases
 - Tools note changes in program
 - Check test-case database for overlap

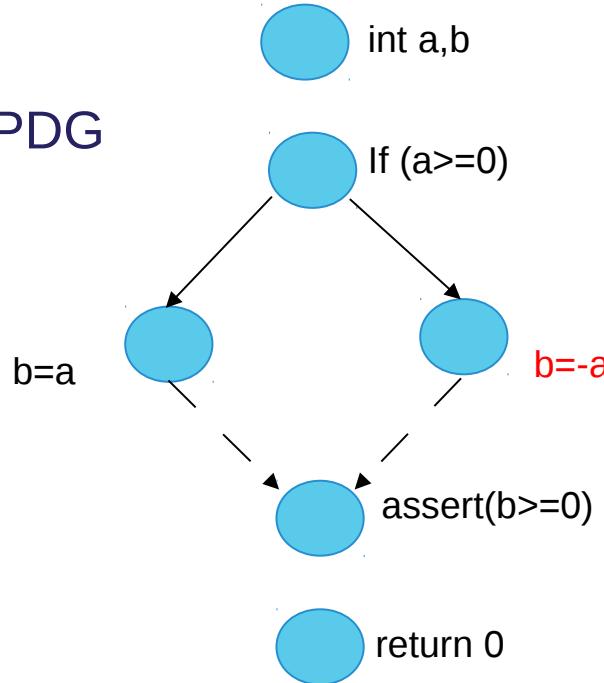
Specification-based Regression Test Selection

- Like code-based and structural regression test case selection
 - Pick test cases that test new and changed functionality
- Difference: No guarantee of independence
 - A test case that isn't "for" changed or added feature X might find a bug in feature X anyway
- Typical approach: Specification-based prioritization
 - Execute all test cases, but start with those that related to changed and added features

Example

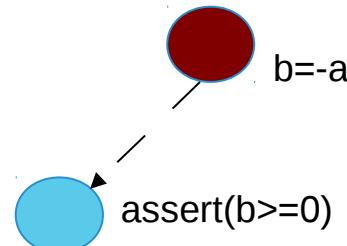
```
int main()
{
    int a, b;
    if (a>=0)
        b = a;
    else
        b = -a;
    assert(b >= 0);
    return 0;
}
```

PDG



Forward Slice

Depth first traversal from node `b = -a;`



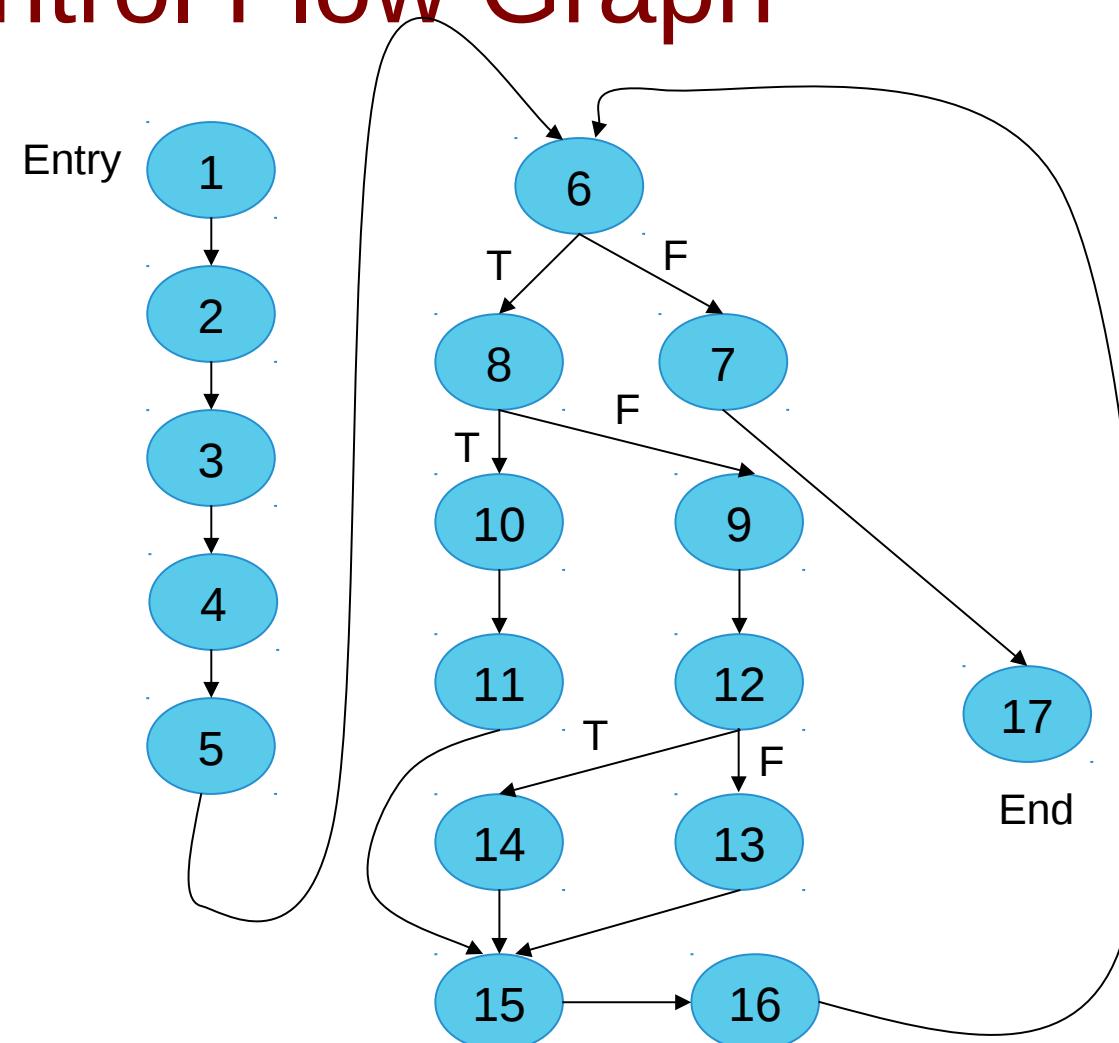
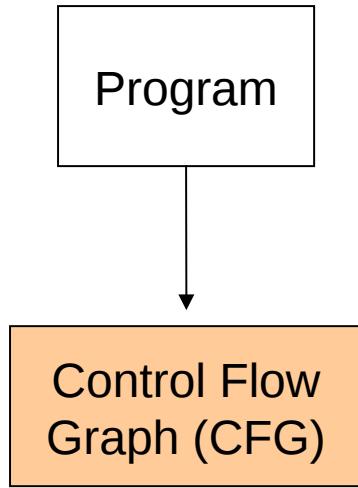
Slicing procedure

Program

Computing the greatest in an array of integers

```
int main(int argc, char* argv[]) {  
    unsigned int num[5] = {12, 23, 4, 78, 34};  
    unsigned int largest, counter = 0;  
    while (counter < 5) {  
        if (counter == 0)  
            largest = num[counter];  
        else if(largest > num[counter])  
            largest = num[counter];  
        ++counter;  
    }  
    for (counter = 0; counter < 5; counter++)  
        assert(largest >= num[counter]);  
}
```

Construct Control Flow Graph



Build a PDG

- Build a Program Dependence Graph (PDG) that captures **control** and **data** dependencies between nodes in CFG

Sample Data Dependency

For *counter* variable

1 → 2,3,4,5,6,7

7 → 2,3,4,5,6,7

```
int main(int argc, char* argv[]) {  
    1 unsigned int num[5] = {12, 23, 4, 78, 34},  
    largest, counter = 0;  
    2 while (counter <5) {  
        3     if (counter ==0)  
        4         largest = num[counter];  
        5     else if(largest < num[counter])  
        6         largest = num[counter];  
        7     counter = counter +1;  
    }  
}
```

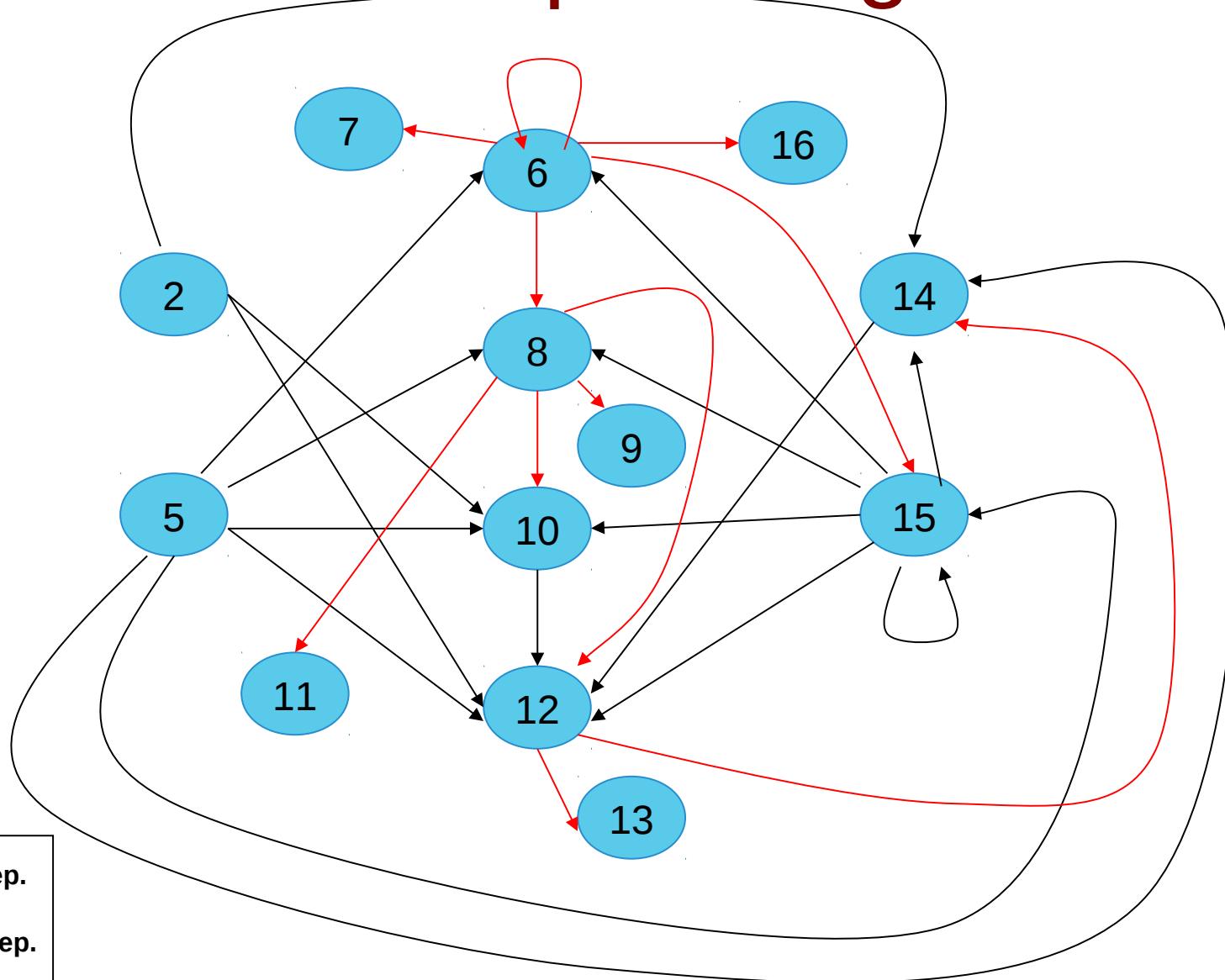
Sample Control Dependency

Conditional in statement 3

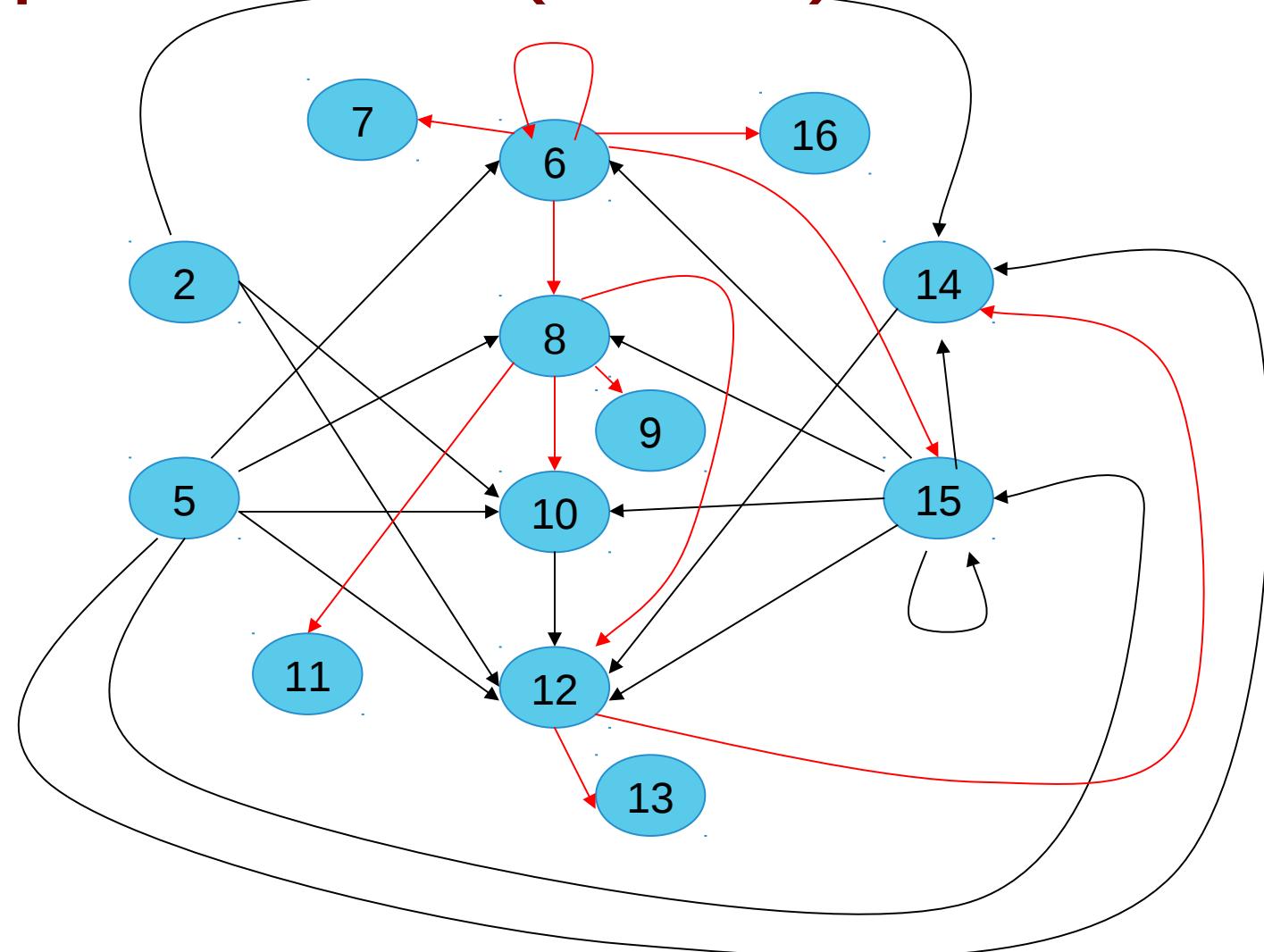
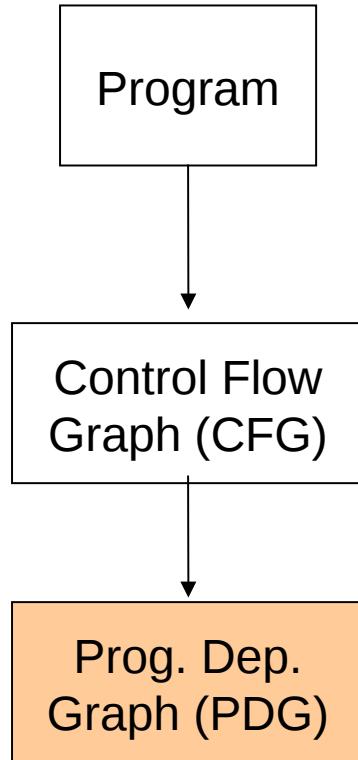
3 → 4, 5

```
int main(int argc, char* argv[]) {
    1 unsigned int num[5] = {12, 23, 4, 78, 34},
    largest, counter = 0;
    2 while (counter <5) {
        3     if (counter ==0)
            4         largest = num[counter];
        5     else if(largest < num[counter])
            6         largest = num[counter];
        7     counter = counter +1;
    }
}
```

PDG for Example Program



Slicing procedure (so far)



Slight change in the example

```
int main(int argc, char* argv[]) {  
    unsigned int num[5] = {12, 23, 4, 78, 34},  
        largest, counter = 0;  
    while (counter < 5) {  
        if (counter == 0)  
            largest = num[counter];  
        else if(largest > num[counter])  
            largest = num[counter];  
        ++counter;  
    }  
    for (counter = 0; counter < 5;  
         counter++)  
        assert(largest >= num[counter]);  
}
```

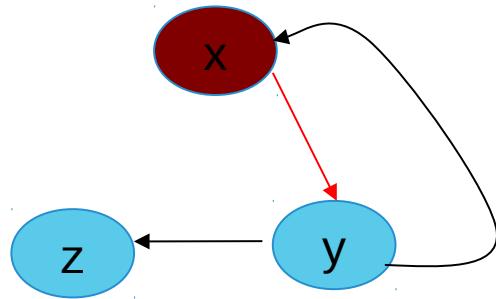
Changed program

Forward Slicing from Changes

- Compute the nodes corresponding to changed statements in the PDG, and
- Compute a transitive closure over all forward dependencies (control + data) from these nodes.

Forward Slice

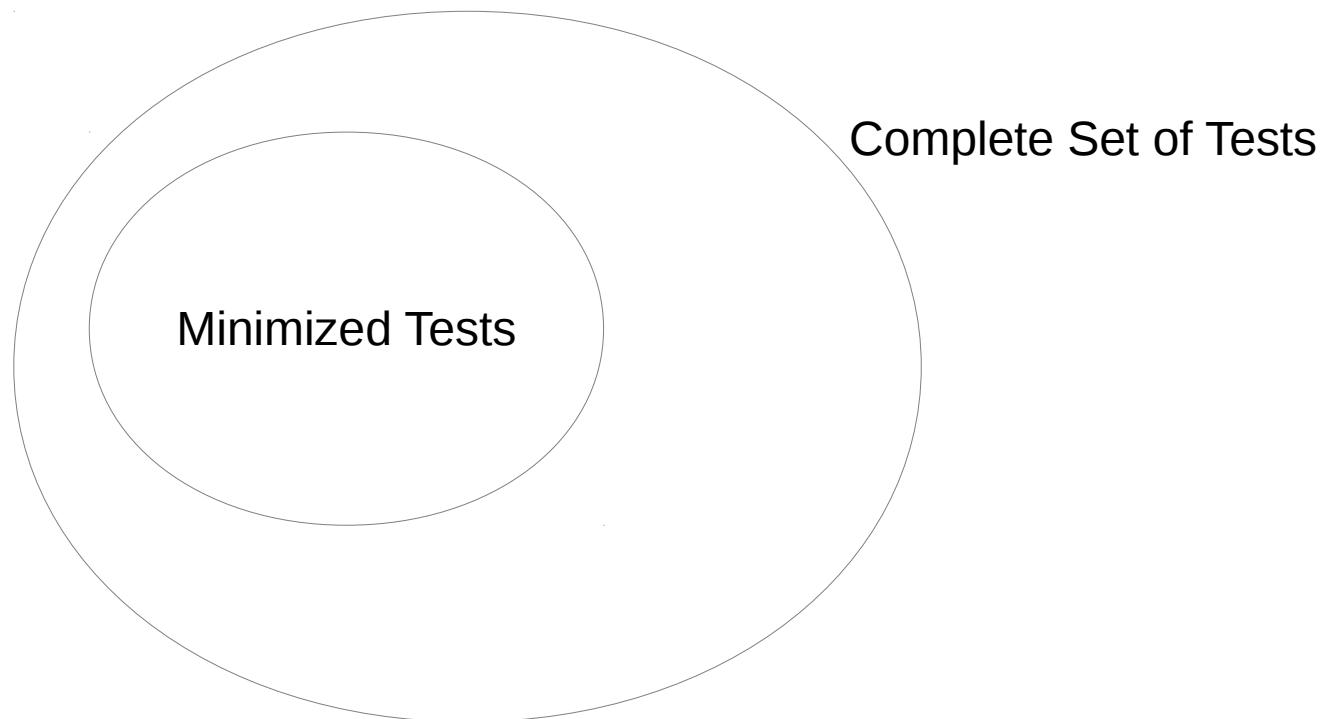
Depth first traversal from changed node



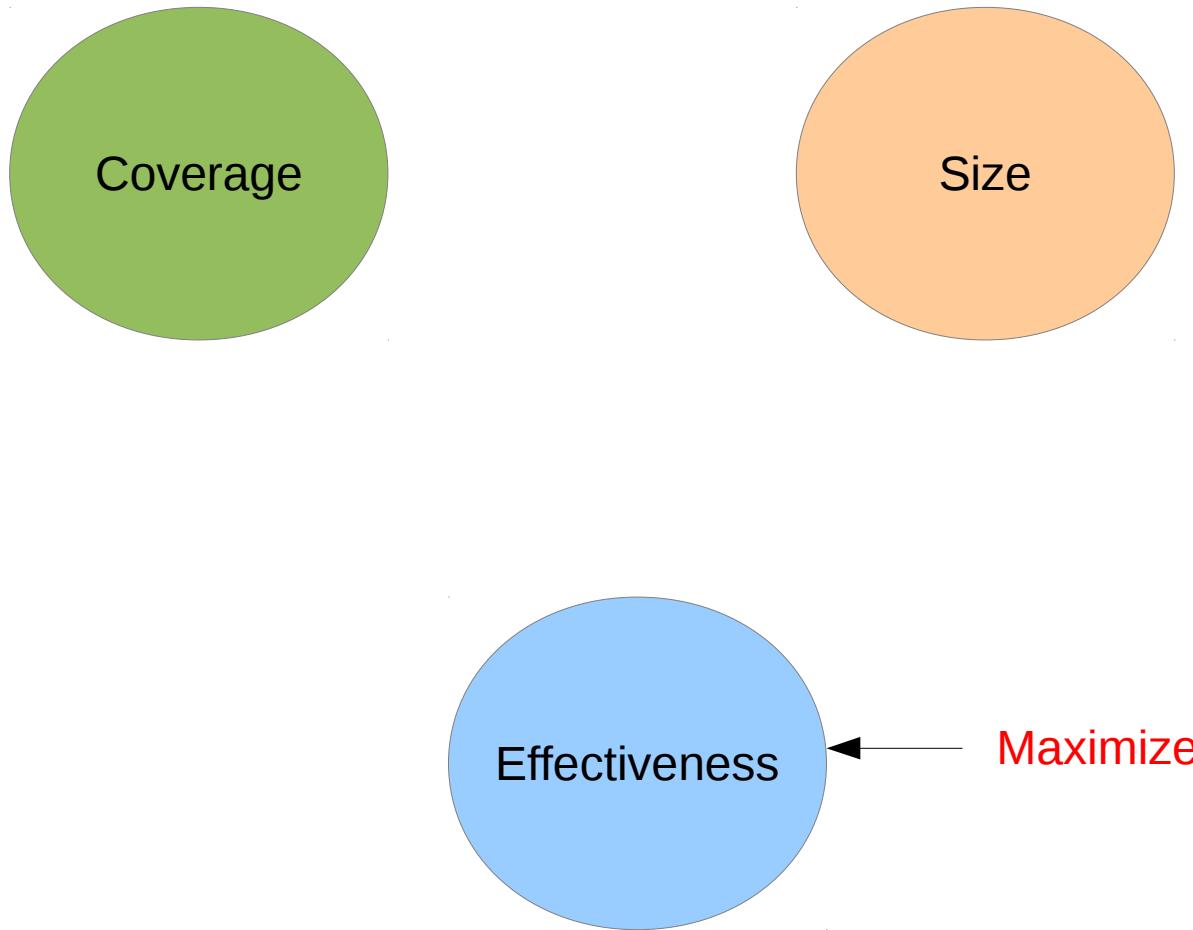
```
else if(largest < num[counter])  
    largest = num[counter];  
assert (largest >= num[counter]);
```

Test Set Minimization

Identify test cases that are **redundant** and remove them from the test suite to reduce its size.



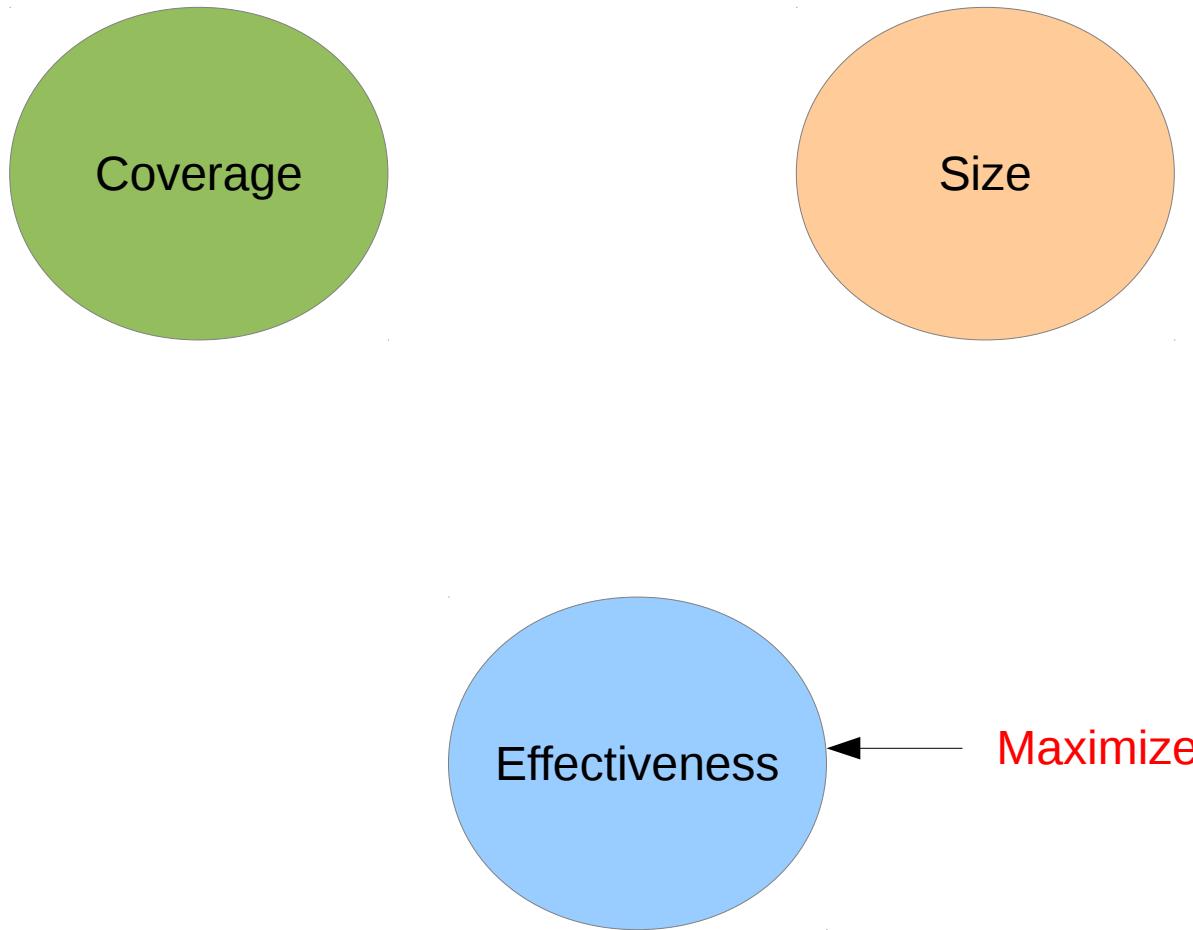
Test Set Attributes



Structural Coverage

- (In)adequacy criteria
 - If significant parts of program structure are not tested, testing is surely inadequate
- Control flow coverage criteria
 - Statement (node, basic block) coverage
 - Branch (edge) coverage
 - Condition coverage
 - Path coverage
 - Data flow (syntactic dependency) coverage
- Attempted compromise between the impossible and the inadequate

Test Set Attributes



Test Set Attributes

- Higher Coverage -----> Better Fault Detection
- Bigger Size -----> Better Fault Detection

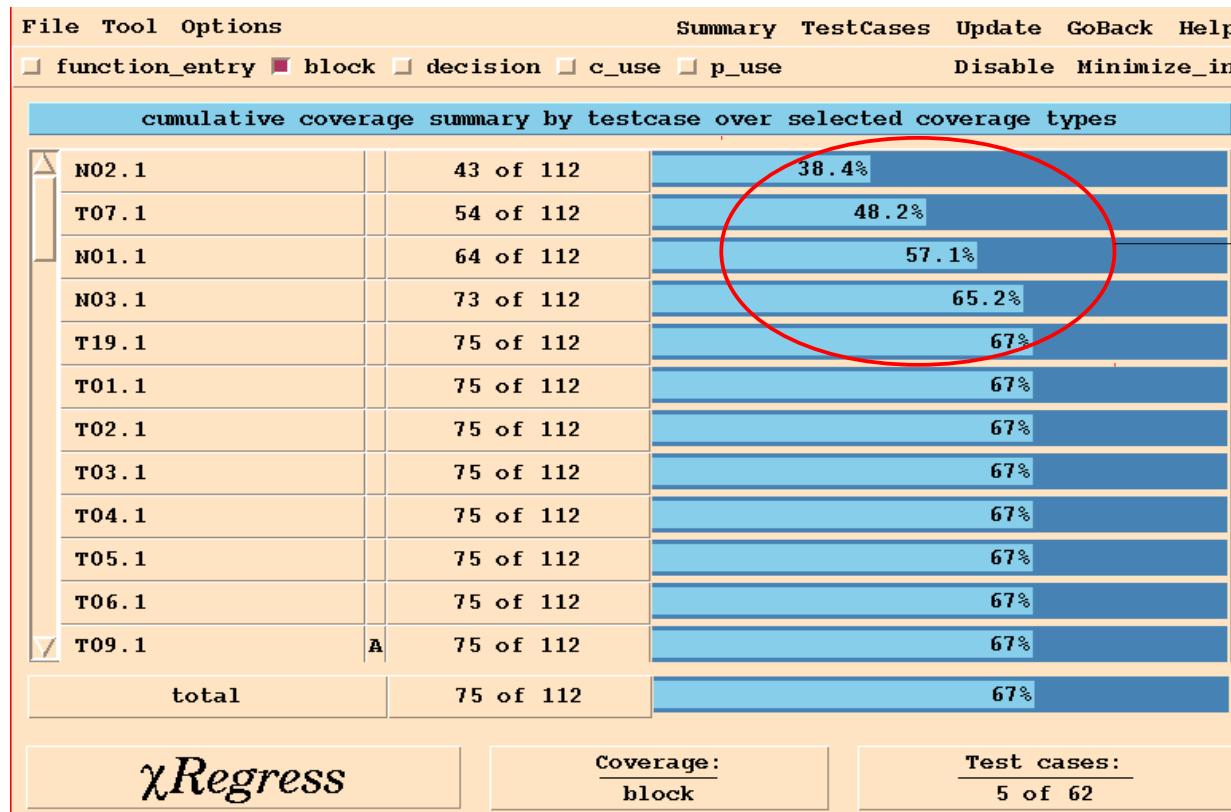
Better Correlated!

Test Set Minimization

- Maximize coverage with minimum number of test cases
- The minimization algorithm can be exponential in time
- Does not occur in our experience
 - Some examples
 - an object-oriented language compiler (100 KLOC)
 - a provisioning application (353 KLOC) with 32K regression tests
 - a database application with 50 files (35 KLOC)
 - a space application (10 KLOC)
- Stop after a pre-defined number of iterations
- Obtain an approximate solution by using a greedy heuristic

Example

Sort test cases in order of increasing cost per additional coverage



Only 5 of the 62 test cases are included in the minimized subset which has the same block coverage as the original test set.

Test Set Prioritisation

- Sort test cases in order of **increasing cost per additional coverage**
- Select the first test case
- Repeat the above two steps until n test cases are selected or max cost is reached (whichever is first)

Example

- Individual decision coverage and cost per test case

```
$ atac -K -md main,atac wc,atac wordcount,trace
```

cost	% decisions	test
120	57(20/35)	wordcount.1
50	11(4/35)	wordcount.2
20	49(17/35)	wordcount.3
10	11(4/35)	wordcount.4
40	71(25/35)	wordcount.5
60	60(21/35)	wordcount.6
80	11(4/35)	wordcount.7
20	66(23/35)	wordcount.8
10	66(23/35)	wordcount.9
70	60(21/35)	wordcount.10
50	60(21/35)	wordcount.11
50	60(21/35)	wordcount.12
50	20(7/35)	wordcount.13
40	14(5/35)	wordcount.14
60	60(21/35)	wordcount.15
20	26(9/35)	wordcount.16
150	54(19/35)	wordcount.17
900	100(35)	== all ==

Example

- Prioritized cumulative decision coverage and cost per test case

```
$ atac -Q -md main,atac wc,atac wordcount,trace
```

cost (cum)	% decisions (cumulative)	test	Cost per additional coverage
10	66(23/35)	wordcount.9	$10/23 = 0.43$
30	77(27/35)	wordcount.3	$(30-10)/(27-23) = 20/4 = 5.00$
40	83(29/35)	wordcount.4	$(40-30)/(29-27) = 10/2 = 5.00$
60	89(31/35)	wordcount.8	$(60-40)/(31-29) = 20/2 = 10.00$
100	91(32/35)	wordcount.5	$(100-60)/(32-31) = 40/1 = 40.00$
140	94(33/35)	wordcount.14	
200	97(34/35)	wordcount.15	
280	100(35)	wordcount.7	
300	100(35)	wordcount.16	
350	100(35)	wordcount.2	
400	100(35)	wordcount.12	
450	100(35)	wordcount.11	
500	100(35)	wordcount.13	
560	100(35)	wordcount.6	
630	100(35)	wordcount.10	
750	100(35)	wordcount.1	
900	100(35)	wordcount.17	

Increasing Order

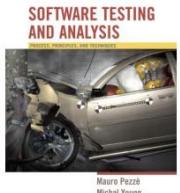
Prioritized Rotating Selection

- Basic idea:
 - Execute all test cases, eventually
 - Execute some sooner than others
- Possible priority schemes:
 - Round robin: Priority to least-recently-run test cases
 - Track record: Priority to test cases that have detected faults before
 - They probably execute code with a high fault density
 - Structural: Priority for executing elements that have not been recently executed
 - Can be coarse-grained: Features, methods, ...

Summary

- Regression testing is an essential phase of software product development.
- In a situation where test resources are limited and deadlines are to be met, execution of all tests might not be feasible.
- One can make use of different techniques for selecting a subset of all tests to reduce the time and cost for regression testing.

System, Acceptance, and Regression Testing

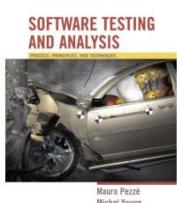


(c) 2007 Mauro Pezzè & Michal Young

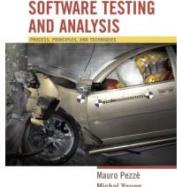
Ch 22, slide 1

Learning objectives

- Distinguish system and acceptance testing
 - How and why they differ from each other and from unit and integration testing
- Understand basic approaches for quantitative assessment (reliability, performance, ...)
- Understand interplay of validation and verification for usability and accessibility
 - How to continuously monitor usability from early design to delivery
- Understand basic regression testing approaches
 - Preventing accidental changes

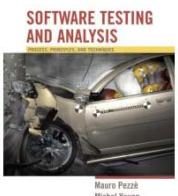


	System	Acceptance	Regression
Test for ...	Correctness, completion	Usefulness, satisfaction	Accidental changes
Test by ...	Development test group	Test group with users	Development test group
Verification		<i>Validation</i>	Verification



22.2

➤ System testing



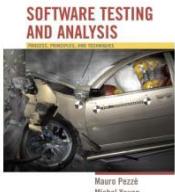
(c) 2007 Mauro Pezzè & Michal Young

Ch 22, slide 4

System Testing

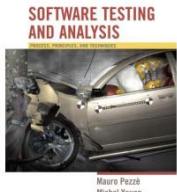
- Key characteristics:
 - Comprehensive (the whole system, the whole spec)
 - Based on specification of observable behavior
 - Verification against a requirements specification, not validation, and not opinions
 - Independent of design and implementation

Independence: Avoid repeating software design errors in system test design



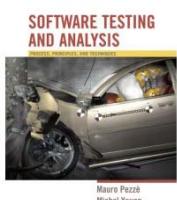
Independent V&V

- *One strategy for maximizing independence:* System (and acceptance) test performed by a different organization
 - Organizationally isolated from developers (no pressure to say “ok”)
 - Sometimes outsourced to another company or agency
 - Especially for critical systems
 - Outsourcing for independent judgment, not to save money
 - May be *additional* system test, not replacing internal V&V
 - Not all outsourced testing is IV&V
 - Not *independent* if controlled by development organization



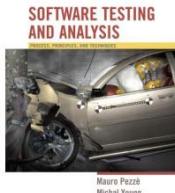
Independence without changing staff

- If the development organization controls system testing ...
 - Perfect independence may be unattainable, but we can reduce undue influence
- Develop system test cases early
 - As part of requirements specification, before major design decisions have been made
 - Agile “test first” and conventional “V model” are both examples of designing system test cases before designing the implementation
 - An opportunity for “design for test”: Structure system for critical system testing early in project



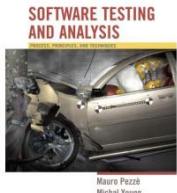
Incremental System Testing

- System tests are often used to measure progress
 - System test suite covers all features and scenarios of use
 - As project progresses, the system passes more and more system tests
- Assumes a “threaded” incremental build plan:
Features exposed at top level as they are developed



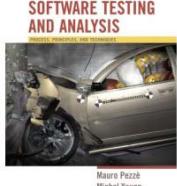
Global Properties

- 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



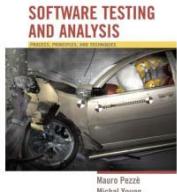
Context-Dependent Properties

- 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



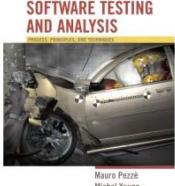
Establishing an Operational Envelope

- When a property (e.g., performance or real-time response) is parameterized by use ...
 - requests per second, size of database, ...
- Extensive stress testing is required
 - varying parameters within the envelope, near the bounds, and beyond
- Goal: A well-understood model of how the property varies with the parameter
 - How sensitive is the property to the parameter?
 - Where is the “edge of the envelope”?
 - What can we expect when the envelope is exceeded?



Stress Testing

- Often requires extensive simulation of the execution environment
 - With systematic variation: What happens when we push the parameters? What if the number of users or requests is 10 times more, or 1000 times more?
- Often requires more resources (human and machine) than typical test cases
 - Separate from regular feature tests
 - Run less often, with more manual control
 - Diagnose deviations from expectation
 - Which may include difficult debugging of latent faults!



Capacity Testing

- **When:** systems that are intended to cope with high volumes of data should have their limits tested and we should consider how they fail when capacity is exceeded
- **What/How:** usually we will construct a harness that is capable of generating a very large volume of simulated data that will test the capacity of the system or use existing records
- **Why:** we are concerned to ensure that the system is fit for purpose say ensuring that a medical records system can cope with records for all people in the UK (for example)
- **Strengths:** provides some confidence the system is capable of handling high capacity
- **Weaknesses:** simulated data can be unrepresentative; can be difficult to create representative tests; can take a long time to run

Security Testing

- **When:** most systems that are open to the outside world and have a function that should not be disrupted require some kind of security test. Usually we are concerned to thwart malicious users.
- **What/How:** there are a range of approaches. One is to use league tables of bugs/errors to check and review the code (e.g. SANS top twenty-five security-related programming errors). We might also form a team that attempts to break/break into the system.
- **Why:** some systems are essential and need to keep running, e.g. the telephone system, some systems need to be secure to maintain reputation.
- **Strengths:** this is the best approach we have most of the effort should go into design and the use of known secure components.
- **Weaknesses:** we only cover known ways in using checklists and we do not take account of novelty using a team to try to break does introduce this.

Performance Testing

- **When:** many systems are required to meet performance targets laid down in a service level agreement (e.g. does your ISP give you 2Mb/s download?).
- **What/How:** there are two approaches - modelling/simulation, and direct test in a simulated environment (or in the real environment).
- **Why:** often a company charges for a particular level of service - this may be disputed if the company fails to deliver. E.g. the VISA payments system guarantees 5s authorisation time delivers faster and has low variance. Customers would be unhappy with less.
- **Strengths:** can provide good evidence of the performance of the system, modelling can identify bottlenecks and problems.
- **Weaknesses:** issues with how representative tests are.

Compliance Testing

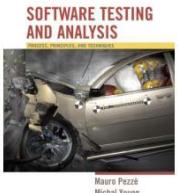
- **When:** we are selling into a regulated market and to sell we need to show compliance. E.g. if we have a C compiler we should be able to show it correctly compiles ANSI C.
- **What/How:** often there will be standardised test sets that constitute good coverage of the behaviour of the system (e.g. a set of C programs, and the results of running them).
- **Why:** we can identify the problem areas and create tests to check that set of conditions.
- **Strengths:** regulation shares the cost of tests across many organisations so we can develop a very capable test set.
- **Weaknesses:** there is a tendency for software producers to orient towards the compliance test set and do much worse on things outside the compliance test set.

Documentation Testing

- **When:** most systems that have documentation should have it tested and should be tested against the real system. Some systems embed test cases in the documentation and using the doc tests is an essential part of a new release.
- **What/How:** test set is maintained that verifies the doc set matches the system behaviour. Could also just get someone to do the tutorial and point out the errors.
- **Why:** the user gets really confused if the system does not conform to the documentation.
- **Strengths:** ensures consistency.
- **Weaknesses:** not particularly good on checking consistency of narrative rather than examples.

22.3

➤ Acceptance testing

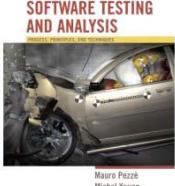


(c) 2007 Mauro Pezzè & Michal Young

Ch 22, slide 13

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

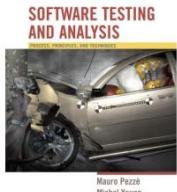


Definitions

- **Reliability:** Survival Probability
 - When function is critical during the mission time.
- **Availability:** The fraction of time a system meets its specification.
 - Good when continuous service is important but it can be delayed or denied
- **Failsafe:** System fails to a known safe state
- **Dependability:** Generalisation - System does the right thing at right time

Statistical Sampling

- We need a valid *operational profile* (model)
 - Sometimes from an older version of the system
 - Sometimes from operational environment (e.g., for an embedded controller)
 - *Sensitivity testing* reveals which parameters are most important, and which can be rough guesses
- And a clear, precise definition of what is being measured
 - Failure rate? Per session, per hour, per operation?
- And many, many random samples
 - Especially for high reliability measures



System Reliability

- The reliability, $R_F(t)$ of a system is the probability that no fault of the class F occurs (i.e. system survives) during time t.

$$R_F(t) = P(t_{init} \leq t < t_f \forall f \in F)$$

where t_{init} is time of introduction of the system to service,
 t_f is time of occurrence of the first failure f drawn from F.

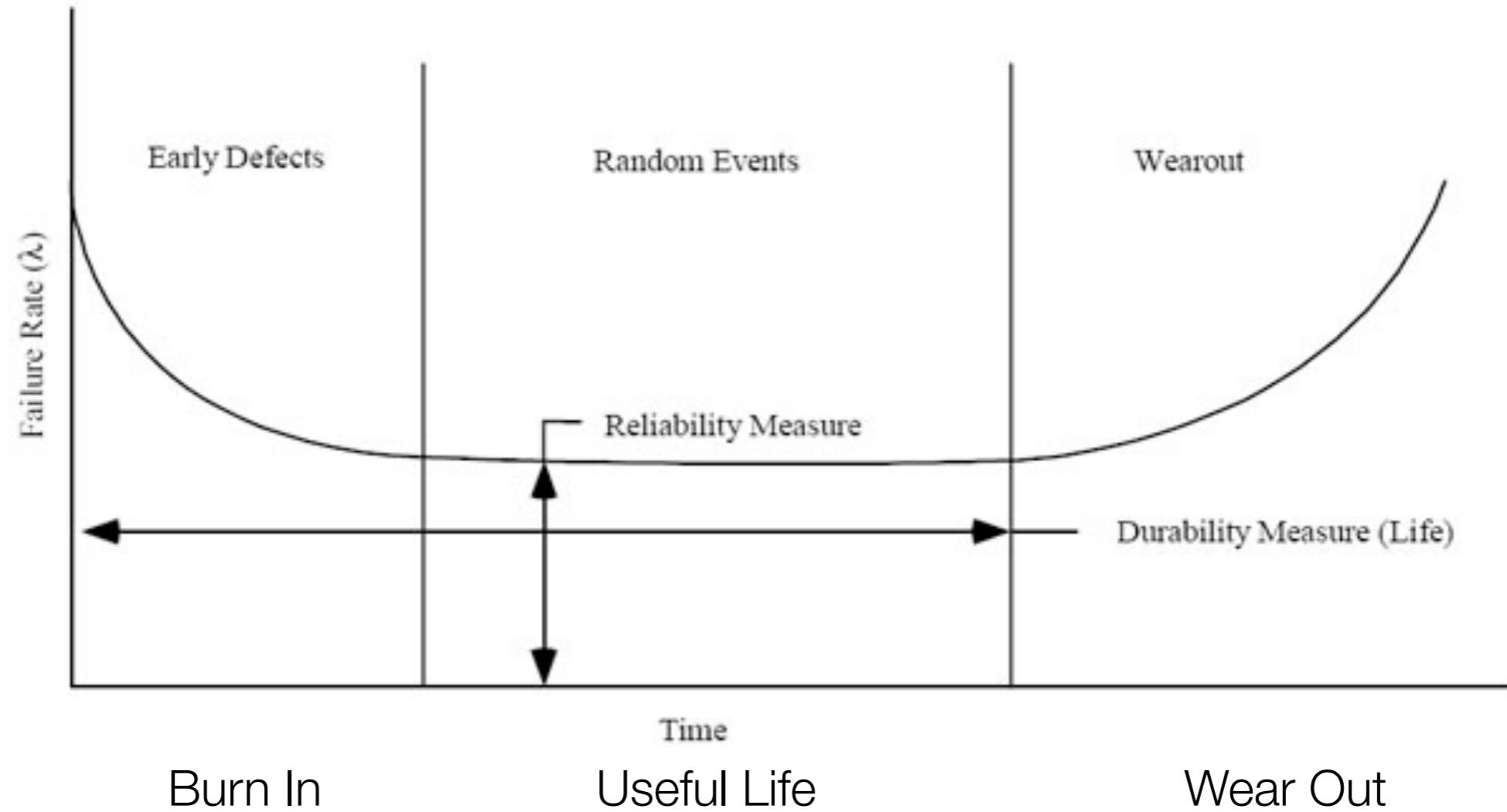
- Failure Probability, $Q_F(t)$ is complementary to $R_F(t)$

$$R_F(t) + Q_F(t) = 1$$

- We can take off the F subscript from $R_F(t)$ and $Q_F(t)$

- When the lifetime of a system is exponentially distributed, the reliability of the system is: $R(t) = e^{-\lambda t}$ where the parameter λ is called the failure rate

Component Reliability Model



During useful life, components exhibit a constant failure rate λ . Reliability of a device can be modelled using an exponential distribution $R(t) = e^{-\lambda t}$

Component Failure Rate

- Failure rates often expressed in failures / million operating hours

Automotive Embedded System Component	Failure Rate λ
Military Microprocessor	0.022
Typical Automotive Microprocessor	0.12
Electric Motor Lead/Acid battery	16.9
Oil Pump	37.3
Automotive Wiring Harness (luxury)	775

MTTF: Mean Time To Failure

- **MTTF:** Mean Time to Failure or Expected Life
- **MTTF:** Mean Time To (first) Failure is defined as the expected value of t_f

$$MTTF = E(t_f) = \int_0^{\infty} R(t)dt = \frac{1}{\lambda}$$

where λ is the failure rate.

- **MTTF** of a system is the expected time of the first failure in a sample of identical initially perfect systems.
- **MTTR:** Mean Time To Repair is defined as the expected time for repair.
- **MTBF:** Mean Time Between Failure

Serial System Reliability

- Serially Connected Components
- $R_k(t)$ is the reliability of a single component k: $R_k(t) = e^{-\lambda_k t}$
- Assuming the failure rates of components are statistically independent.
- The overall system reliability $R_{ser}(t)$

$$R_{ser}(t) = R_1(t) \times R_2(t) \times R_3(t) \times \dots \times R_n(t)$$

$$R_{ser}(t) = \prod_{i=1}^n R_i(t)$$

- No redundancy: Overall system reliability depends on the proper working of each component

$$R_{ser}(t) = e^{-t(\sum_{i=1}^n \lambda_i)}$$

- Serial failure rate

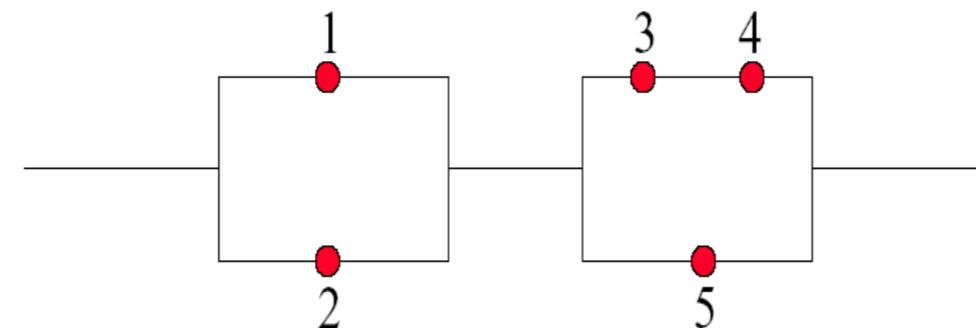
$$\lambda_{ser} = \sum_{i=1}^n \lambda_i$$

System Reliability

- Building a reliable serial system is extraordinarily difficult and expensive.
- For example: if one is to build a serial system with 100 components each of which had a reliability of 0.999, the overall system reliability would be

$$0.999^{100} = 0.905$$

- Reliability of System of Components



- Minimal Path Set:
Minimal set of components whose functioning ensures the functioning of the system: {1,3,4} {2,3,4} {1,5} {2,5}

Parallel System Reliability

- Parallel Connected Components
- $Q_k(t)$ is $1 - R_k(t)$: $Q_k(t) = 1 - e^{-\lambda_k t}$
- Assuming the failure rates of components are statistically independent.

$$Q_{par}(t) = \prod_{i=1}^n Q_i(t)$$

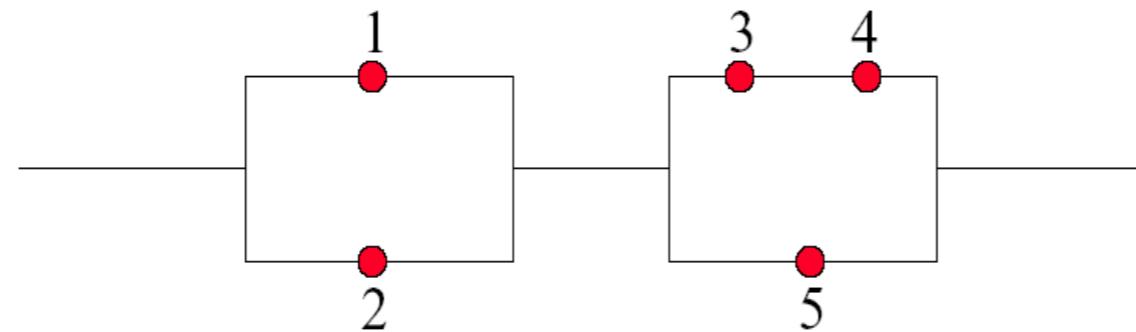
- Overall system reliability: $R_{par}(t) = 1 - \prod_{i=1}^n (1 - R_i(t))$

Example

- Consider 4 identical modules are connected in parallel
- System will operate correctly provided at least one module is operational. If the reliability of each module is 0.95.
- The overall system reliability is $1 - (1 - 0.95)^4 = 0.99999375$

Parallel-Serial Reliability

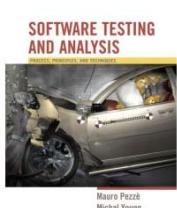
- Parallel and Serial Connected Components



- Total reliability is the reliability of the first half, in serial with the second half.
- Given $R_1=0.9$, $R_2=0.9$, $R_3=0.99$, $R_4=0.99$, $R_5=0.87$
- $R_t = (1 - (1 - 0.9)(1 - 0.9))(1 - (1 - 0.87)(1 - (0.99 \times 0.99))) = 0.987$

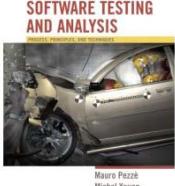
Is Statistical Testing Worthwhile?

- Necessary for ...
 - Critical systems (safety critical, infrastructure, ...)
- But difficult or impossible when ...
 - Operational profile is unavailable or just a guess
 - Often for new functionality involving human interaction
 - But we may factor critical functions from overall use to obtain a good model of only the critical properties
 - Reliability requirement is very high
 - Required sample size (number of test cases) might require years of test execution
 - Ultra-reliability can seldom be demonstrated by testing



Process-based Measures

- Less rigorous than statistical testing
 - Based on similarity with prior projects
- System testing process
 - Expected history of bugs found and resolved
- Alpha, beta testing
 - Alpha testing: Real users, controlled environment
 - Beta testing: Real users, real (uncontrolled) environment
 - May statistically sample users rather than uses
 - Expected history of bug reports



Usability Testing

- **When:** where the system has a significant user interface and it is important to avoid user error — e.g. this could be a critical application e.g. cockpit design in an aircraft or a consumer product that we want to be an enjoyable system to use or we might be considering efficiency (e.g. call-centre software).
- **What/How:** we could construct a simulator in the case of embedded systems or we could just have many users try the system in a controlled environment. We need to structure the test with clear objectives (e.g. to reduce decision time,...) and have good means of collecting and analysing data.
- **Why:** there may be safety issues, we may want to produce something more useable than competitors' products...
- **Strengths:** in well-defined contexts this can provide very good feedback – often underpinned by some theory e.g. estimates of cognitive load.
- **Weaknesses:** some usability requirements are hard to express and to test, it is possible to test extensively and then not know what to do with the data.

Reliability Testing

- **When:** we may want to guarantee some system will only fail very infrequently (e.g. nuclear power control software we might claim no more than one failure in 10,000 hours of operation). This is particularly important in telecommunications.
- **What/How:** we need to create a representative test set and gather enough information to support a statistical claim (system structured modelling supports demonstrating how overall failure rate relates to component failure rate).
- **Why:** we often need to make guarantees about reliability in order to satisfy a regulator or we might know that the market leader has a certain reliability that the market expects.
- **Strengths:** if the test data is representative this can make accurate predictions.
- **Weaknesses:** we need a lot of data for high-reliability systems, it is easy to be optimistic.

Availability/Reparability Testing

- **When:** we are interested in avoiding long down times we are interested in how often failure occurs and how long it takes to get going again. Usually this is in the context of a service supplier and this is a Key Performance Indicator.
- **What/How:** similar to reliability testing – but here we might seed errors or cause component failures and see how long they take to fix or how soon the system can return once a component is repaired.
- **Why:** in providing a critical service we may not want long interruptions (e.g. 999 service).
- **Strengths:** similar to reliability.
- **Weaknesses:** similar to reliability – in the field it may be much faster to fix common problems because of learning.

Summary

- There are a very wide range of potential tests that should be applied to a system.
- Not all systems require all tests.
- Managing the test sets and when they should be applied is a very complex task.
- The quality of test sets is critical to the quality of a running implementation.

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.

Functional vs non-functional security requirements

- Functional requirements – *What shall the software do?*
- Non-functional requirements – *How should it be done?*
- Regular functional requirement example (Webmail system):
It should be possible to use HTML formatting in e-mails
- Functional security requirement example:
The system should check upon user registration that passwords are at least 8 characters long
- Non-functional security requirement example:
All user input must be sanitized before being used in database queries

How would you write a unit test for this?

Software Vulnerabilities

Common software vulnerabilities include

- **Memory safety violations**
 - Buffer overflows
 - Dangling pointers
- **Input Validation errors**
 - Code injection
 - Cross site scripting in web applications
 - Email injection
 - Format string attacks
 - HTTP header injection
- **Race conditions**
 - Symlink races
 - Time of check to time of use bugs
 - SQL injection
- **Privilege confusion**
 - Clickjacking
 - Cross-site request forgery
 - FTP bounce attack
- **Side-channel attack**
 - Timing attack

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

Penetration testing

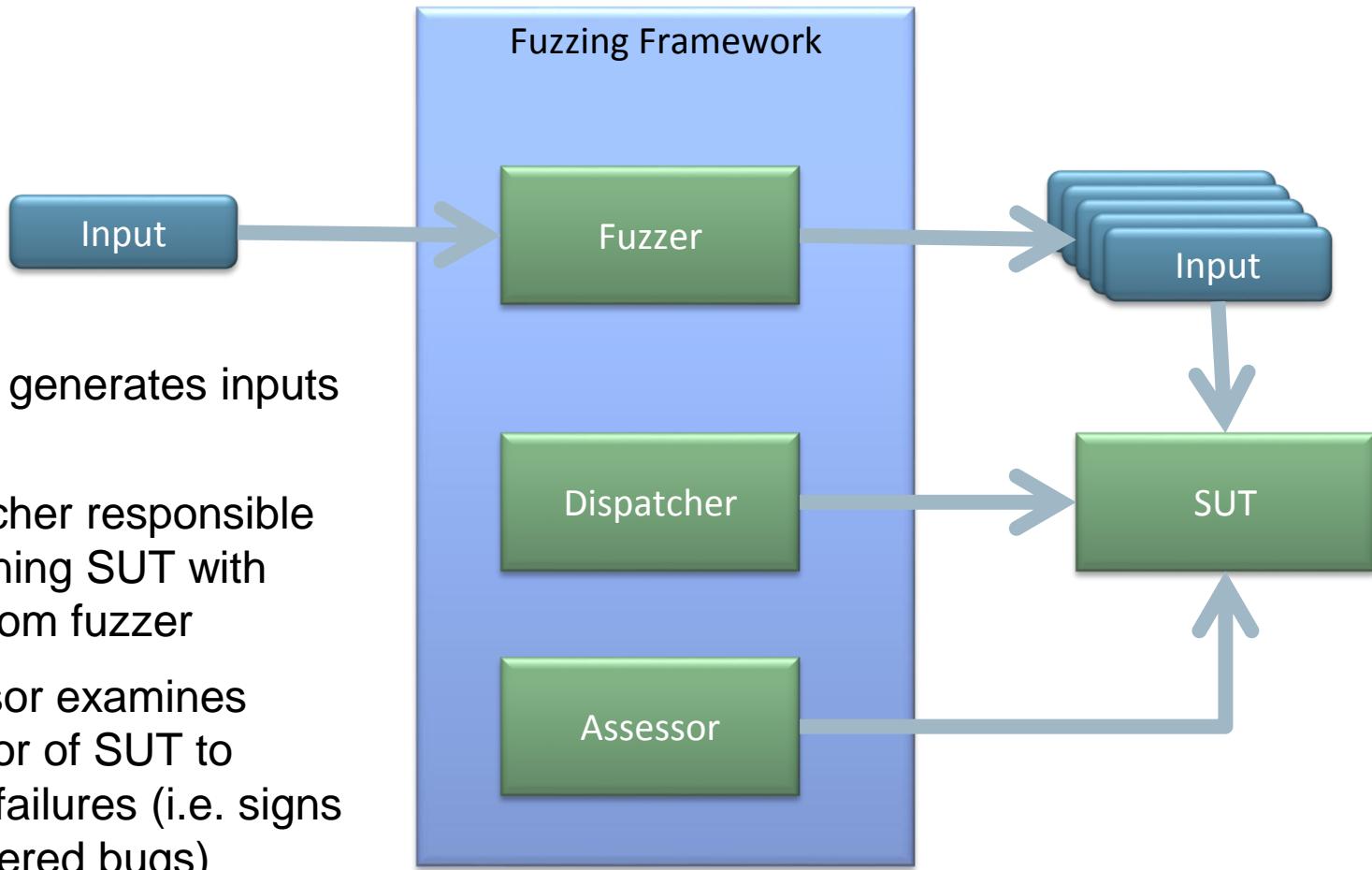
- Manually try to “break” software
- Relies on human intuition and experience.
- Typically involves looking for known common problems.
- Can uncover problems that are impossible or difficult to find using automated methods
 - ...but results completely dependent on skill of tester!

Fuzz testing

Idea: Send semi-valid input to a program and observe its behavior.

- Black-box testing – *System Under Test* (SUT) treated as a “black-box”
- The only feedback is the output and/or externally observable behavior of SUT.
- First proposed in a 1990 paper where completely random data was sent to 85 common Unix utilities in 6 different systems.
24 – 33% crashed.
 - Remember: Crash implies memory protection errors.
 - Crashes are often signs of exploitable flaws in the program!

Fuzz testing architecture



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

Mutation based fuzzing

Start with a valid seed input, and “mutate” it.

- Flip some bits, change value of some bytes.
- Programs that have highly structured input, e.g. XML, may require “smarter” mutations.

Challenge: How to select appropriate seed input?

- If official test suites are available, these can be used.

Generally mostly used for programs that take files as input.

- Trickier to do when interpretation of inputs depends on program state, e.g. network protocol parsers.
(The way a message is handled depends on previous messages.)

Mutation based fuzzing – Pros and Cons

- 😊 Easy to get started, no (or little) knowledge of specific input format needed.
- 😢 Typically yields low code coverage, inputs tend to deviate too much from expected format – rejected by early sanity checks.

```
int parse_input(char* data, size_t size)
{
    int saved_checksum, computed_checksum;

    if(size < 4) return ERR_CODE;

    // First four bytes of 'data' is CRC32 checksum
    saved_checksum = *((int*)data);

    // Compute checksum for rest of 'data'
    computed_checksum = CRC32(data + 4, size - 4);

    // Error if checksums don't match
    if(computed_checksum != saved_checksum)
        return ERR_CODE;

    // Continue processing of 'data'
    ...
}
```

Mutated inputs will always be rejected here!

Mutation based fuzzing – Pros and Cons

- 😊 Easy to get started, no (or little) knowledge of specific input format needed.
- 😢 Typically yields low code coverage, inputs tend to deviate too much from expected format – rejected by early sanity checks.
- 😢 Hard to reach “deeper” parts of programs by random guessing

```
int parse_record(char* data, int type)
{
    switch(type) {
        case 0xFF0001:
            parse_type_A(data);
            break;

        case 0xFF0002:
            parse_type_B(data);
            break;

        case 0xFF0003:
            parse_type_C(data);
            break;
    ...
}
```

*Very unlikely to guess
“magic constants”
correctly.*

*If seed only contains
Type A records,
parse_type_B will
likely never be tested.*

Generation based fuzzing

Idea: Use a specification of the input format (e.g. a grammar) to automatically generate semi-valid inputs

Usually combined with various fuzzing heuristics that are known to trigger certain vulnerability types.

- Very long strings, empty strings
- Strings with format specifiers, “extreme” format strings
 - %n%n%n%n%n%n%n%n%n%n%n%n%n%n%
 - %s%s%s%s%s%s%s%s%s%s%s%s%
 - %5000000.x
- Very large or small values, values close to max or min for data type
0x0, 0xffffffff, 0x7fffffff, 0x80000000, 0xfffffff
- Negative values where positive ones are expected

Generation based fuzzing – Pros and Cons

- 😊 Input is much closer to the expected, much better coverage
- 😊 Can include models of protocol state machines to send messages in the sequence expected by SUT.

- 😢 Requires input format to be known.
- 😢 May take considerable time to write the input format grammar/specification.

Examples of generation based fuzzers (open source)

- SPIKE (2001):
 - Early successful generation based fuzzer for network protocols.
 - Designed to fuzz input formats consisting of blocks with fixed or variable size. E.g. [type][length][data]
- Peach (2006):
 - Powerful fuzzing framework which allows specifying input formats in an XML format.
 - Can represent complex input formats, but relatively steep learning curve.
- Sulley (2008):
 - More modern fuzzing framework designed to be somewhat simpler to use than e.g. Peach.
- Also several commercial fuzzers, e.g.
Codenomicon DEFENSICS, BeStorm, etc.

Fuzzing components: The Dispatcher

Responsible for running the SUT on each input generated by fuzzer module.

- Must provide suitable environment for SUT.
 - E.g. implement a “client” to communicate with a SUT using the fuzzed network protocol.
- SUT may modify environment (file system, etc.)
 - Some fuzzing frameworks allow running SUT inside a virtual machine and restoring from known good snapshot after each SUT execution.

Fuzzing components: The Assessor

Must automatically assess observed SUT behavior to determine if a fault was triggered.

- For C/C++ programs: Monitor for memory access violations, e.g. out-of-bounds reads or writes.
- Simplest method: Just check if SUT crashed.
- Problem: SUT may catch signals/exceptions to gracefully handle e.g. segmentation faults
 - ⇒ Difficult to tell if a fault, (which could have been exploitable with carefully crafted input), have occurred

Improving fault detection

One solution is to attach a programmable debugger to SUT.

- Can catch signals/exceptions prior to being delivered to application.
- Can also help in manual diagnosis of detected faults by recording stack traces, values of registers, etc.

However: All faults do not result in failures, i.e. a crash or other observable behavior (e.g. Heartbleed).

- An out-of-bounds read/write or use-after-free may e.g. not result in a memory access violation.
- Solution: Use a dynamic-analysis tool that can monitor what goes on “under the hood”
 - Can potentially catch more bugs, but SUT runs (considerably) slower.
 - ⇒ Need more time for achieving the same level of coverage

Memory error checkers

Two open source examples

AddressSanitizer

- Applies instrumentation during compilation: Additional code is inserted in program to check for memory errors.
- Monitors all calls to malloc/new/free/delete – can detect if memory is freed twice, used after free, out of bounds access of heap allocated memory, etc.
- Inserts checks that stack buffers are not accessed out of bounds
- Detects use of uninitialized variables
- etc...

Valgrind/Memcheck

- Applies instrumentation directly at the binary level during runtime – does not need source code!
- Can detect similar problems as AddressSanitizer
- Applying instrumentation at the machine code level has some benefits – works with any build environment, can instrument third-party libraries without source code, etc.
- But also comes at a cost; Runs slower than e.g. AddressSanitizer and can generally not detect out-of-bounds access to buffers on stack.
 - Size of stack buffers not visible in machine code

Limitations of fuzz testing

- Many programs have an infinite input space and state space - Combinatorial explosion!
- Conceptually a simple idea, but many subtle practical challenges
- Difficult to create a truly generic fuzzing framework that can cater for all possible input formats.
 - For best results often necessary to write a custom fuzzer for each particular SUT.
- (Semi)randomly generated inputs are very unlikely to trigger certain faults.

Limitations of fuzz testing

Example from first lecture on vulnerabilities

```
char buffer[100];
if(strlen(input) > 100)
{
    printf("String too long!");
    exit(1);
}
strcpy(buffer, input);
```

The off-by-one error will only
be detected if
`strlen(input) == 100`

Very unlikely to trigger this
bug using black-box fuzz
testing!

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)

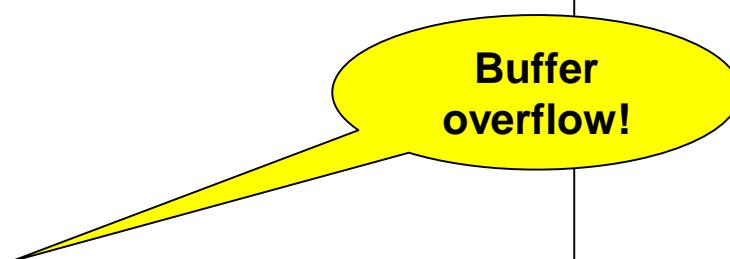
⇒ Complete – Reported bugs are always real bugs

Concolic testing – small example

```
void wheather(int temperature, unsigned int precipitation)
{
    char forecast[5];

    if(temperature > 0) {
        if(precipitation > 0)
            strcpy(forecast, "rain");
        else
            strcpy(forecast, "nice");
    } else {
        if(precipitation == 0)
            strcpy(forecast, "cold");
        else if(precipitation > 20)
            strcpy(forecast, "blizzard");
        else
            strcpy(forecast, "snow");
    }
}
```

...



Buffer overflow!

Concolic testing – small example

```
void wheather(int temperature, unsigned int precipitation )  
{  
    char forecast[5];  
  
    if(temperature > 0) {  
        if(precipitation > 0)  
            strcpy(forecast, "rain");  
        else  
            strcpy(forecast, "nice");  
    } else {  
        if(precipitation == 0)  
            strcpy(forecast, "cold");  
        else if(precipitation > 20)  
            strcpy(forecast, "blizzard");  
        else  
            strcpy(forecast, "snow");  
    }  
    ...  
}
```

First round:

Concrete inputs (arbitrary):
temperature=1, precipitation=1

Recorded path constraints:
temperature > 0 \wedge precipitation > 0

New path constraint:
temperature > 0 \wedge \neg (precipitation > 0)

Solution from solver:
temperature=1, precipitation=0

Concolic testing – small example

```
void wheather(int temperature, unsigned int precipitation )  
{  
    char forecast[5];  
  
    if(temperature > 0) {  
        if(precipitation > 0)  
            strcpy(forecast, "rain");  
        else  
            strcpy(forecast, "nice");  
    } else {  
        if(precipitation == 0)  
            strcpy(forecast, "cold");  
        else if(precipitation > 20)  
            strcpy(forecast, "blizzard");  
    }  
};
```

Note: 'precipitation' is unconstrained – no need to care about later branch conditions when negating an earlier condition.

Second round:

Concrete inputs:
temperature=1, precipitation=0

Recorded path constraints:
 $\text{temperature} > 0 \wedge \neg (\text{precipitation} > 0)$

New path constraint:
 $\neg (\text{temperature} > 0)$

Solution from solver:
temperature=0, precipitation=0

Concolic testing – small example

```
void wheather(int temperature, unsigned int precipitation )  
{  
    char forecast[5];  
  
    if(temperature > 0) {  
        if(precipitation > 0)  
            strcpy(forecast, "rain");  
        else  
            strcpy(forecast, "nice");  
    } else {  
        if(precipitation == 0)  
            strcpy(forecast, "cold");  
        else if(precipitation > 20)  
            strcpy(forecast, "blizzard");  
        else  
            strcpy(forecast, "snow");  
    }  
    ...  
}
```

Third round:

Concrete inputs:

temperature=0, precipitation=0

Recorded path constraints:

$\neg(\text{temperature} > 0) \wedge \text{precipitation} = 0$

New path constraint:

$\neg(\text{temperature} > 0) \wedge \neg(\text{precipitation} = 0)$

Solution from solver:

temperature=0, precipitation=1

Concolic testing – small example

```
void wheather(int temperature, unsigned int precipitation )  
{  
    char forecast[5];  
  
    if(temperature > 0) {  
        if(precipitation > 0)  
            strcpy(forecast, "rain");  
        else  
            strcpy(forecast, "nice");  
    } else {  
        if(precipitation == 0)  
            strcpy(forecast, "cold");  
        else if(precipitation > 20)  
            strcpy(forecast, "blizzard");  
        else  
            strcpy(forecast, "snow");  
    }  
    ...
```

Fourth round:

Concrete inputs:

temperature=0, precipitation=1

Recorded path constraints:

$\neg(\text{temperature} > 0) \wedge \neg(\text{precipitation} = 0)$
 $\wedge \neg(\text{precipitation} > 20)$

New path constraint:

$\neg(\text{temperature} > 0) \wedge \neg(\text{precipitation} = 0)$
 $\wedge \text{precipitation} > 20$

Solution from solver:

temperature=0, precipitation=21

Concolic testing – small example

```
void wheather(int temperature, unsigned int precipitation )  
{  
    char forecast[5];  
  
    if(temperature > 0) {  
        if(precipitation > 0)  
            strcpy(forecast, "rain");  
        else  
            strcpy(forecast, "nice");  
    } else {  
        if(precipitation == 0)  
            strcpy(forecast, "cold");  
        else if(precipitation > 20)  
            strcpy(forecast, "blizzard");  
        else  
            strcpy(forecast, "snow");  
    }  
  
    ...
```

Fifth round:

Concrete inputs:

temperature=0, precipitation=21

Recorded path constraints:

$\neg(\text{temperature} > 0) \wedge \neg(\text{precipitation} = 0)$
 $\wedge \text{precipitation} > 20$

Bug found!

Challenges with concolic testing: Path explosion

Number of paths increase exponentially with number of branches

- Most real-world programs have an infinite state space!
 - For example, number of loop iterations may depend on size of input

Not possible to explore all paths

- ⇒ Need a strategy to explore “interesting” parts of the program

Challenges with concolic testing: Path explosion

Depth-first search (as in the first example) will easily get “stuck” in one part of the program

- May e.g. keep exploring the same loop with more and more iterations

Breadth-first search will take a very long time to reach “deep” states

- May take “forever” to reach the buggy code

Try “smarter” ways of exploring the program state space

- May want to try to run loops many times to uncover possible buffer overflows
- ...but also want to maximize coverage of different parts of the program

Generational search (“whitebox fuzzing”)

The Microsoft SAGE system implements “whitebox fuzzing”

- Performs concolic testing, but prioritizes paths based on how much they improve coverage
- Results can be assessed similar to black-box fuzzing (with dynamic analysis tools, etc.)

Search algorithm outline:

1. Run program on concrete seed input and record path constraint
2. For each branch condition:
 - I. Negate condition and keep the earlier conditions unchanged.
 - II. Have SMT solver generate new satisfying input and run program with that input.
 - III. Assign input a score based on how much it improves coverage
(i.e. how much previously unseen code will be executed with that input)
 - IV. Store input in a global worklist **sorted on score**
3. Pick input at head of worklist and repeat.

Rationale for “whitebox fuzzing”

- Coverage based heuristic avoids getting “stuck” as in DFS
 - If one test case executes the exact same code as in a previous test case its score will be 0 → Moved to end of worklist → Probably never used again
 - Gives sparse but more diverse search of the paths of the program
- Implements a form of greedy search heuristic
 - For example, loops may only be executed once!
 - Only generates input “close” to the seed input (cf. mutation-based fuzzing)
 - ⇒ Will try to explore the “weird corner cases” in code exercised by seed input
 - ⇒ Choice of seed input important for good coverage

Rationale for “whitebox fuzzing”

Has proven to work well in practice

- Used in production at Microsoft to test e.g. Windows, Office, etc. prior to release
 - Has uncovered many serious vulnerabilities that was missed by other approaches (black-box fuzzing, static analysis, etc.)

Interestingly, SAGE works directly at the machine-code level

- Note: Source code not needed for concolic execution – sufficient to collect constraints from one concrete sequence of machine-code instructions.
- Avoids hassle with different build environments, third-party libraries, programs written in different languages etc.
- ...but sacrifices some coverage due to additional approximations needed when working on machine code

Limitations of concolic testing

- The success of concolic testing is due to the massive improvement in SAT/SMT solvers during the last two decades.
 - Main bottleneck is still often the solvers.
 - Black-box fuzzing can perform a much larger number of test cases per time unit – may be more time efficient for “shallow” bugs.
- Solving SAT/SMT problems is NP-complete.
 - Solvers like e.g. Z3 use various “tricks” to speed up common cases
 - ...but may take unacceptably long time to solve certain path constraints.
- Solving SMT queries with non-linear arithmetic (multiplication, division, modulo, etc.) is an undecidable problem in general.
 - But since programs typically use fixed-precision data types, non-linear arithmetic is decidable in this special case.

Limitations of concolic testing

If program uses any kind of cryptography, symbolic execution will typically fail.

- Consider previous checksum example:
- CRC32 is linear and reversible – solver can “repair” checksum if rest of data is modified.

```
....  
    // Compute checksum for rest of 'data'  
    computed_checksum = CRC32(data + 4, size - 4);  
  
    // Error if checksums don't match  
    if(computed_checksum != saved_checksum)  
        return ERR_CODE;
```

What if program used e.g. md5 or SHA256 here instead?

Solver would get “stuck” trying to solve this constraint!

Generation-based fuzzing could handle this without problem!

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.

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

Mutational fuzzer would need to guess correct values of all four conditions in one go to reach bug!

Greybox fuzzing

- Instead of recording full path constraint (as in whitebox fuzzing), record light-weight coverage information to guide fuzzing.
- Use evolutionary algorithms to “learn” input format.
- American Fuzzy Lop (AFL) is considered the current state-of-the art in fuzzing.
 - Performs “regular” mutation-based fuzzing (using several different strategies) and measures code coverage.
 - Every generated input that resulted in any new coverage is saved and later re-fuzzed
 - This extremely simple evolutionary algorithm allows AFL to gradually “learn” how to reach deeper parts of the program.
 - Also highly optimized for speed – can reach several thousand test cases per second.
 - This often beats smarter (and slower) methods like whitebox fuzzing!
 - Has found hundreds of serious vulnerabilities in open-source programs!

Conclusions

Fuzzing – Black-box testing method.

- Semi-random input generation.
- Despite being a brute-force, somewhat ad-hoc approach to security testing, experience has shown that it improves security in practice.

Concolic testing – White-box testing method.

- Input generated from control-structure of code to systematically explore different paths of the program.
- Some in-house and academic implementations exist, but some challenges left to solve before widespread adoption.

Greybox fuzzing

- Coverage-guided semi-random input generation.
- High speed sometimes beats e.g. concolic testing, but shares some limitations with mutation-based fuzzing (e.g. magic constants, checksums)

Conclusions

- Test cases generated automatically, either semi-randomly or from structure of code
 - Test cases not based on requirements
 - Pro: Not “biased” by developers’ view of “how things should work”, can uncover unsound assumptions or corner cases not covered by specification.
 - Con: Fuzzing and concolic testing mostly suited for finding *implementation* errors, e.g. buffer overflows, arithmetic overflows, etc.
- Generally hard to test for high-level errors in requirements and design using these methods

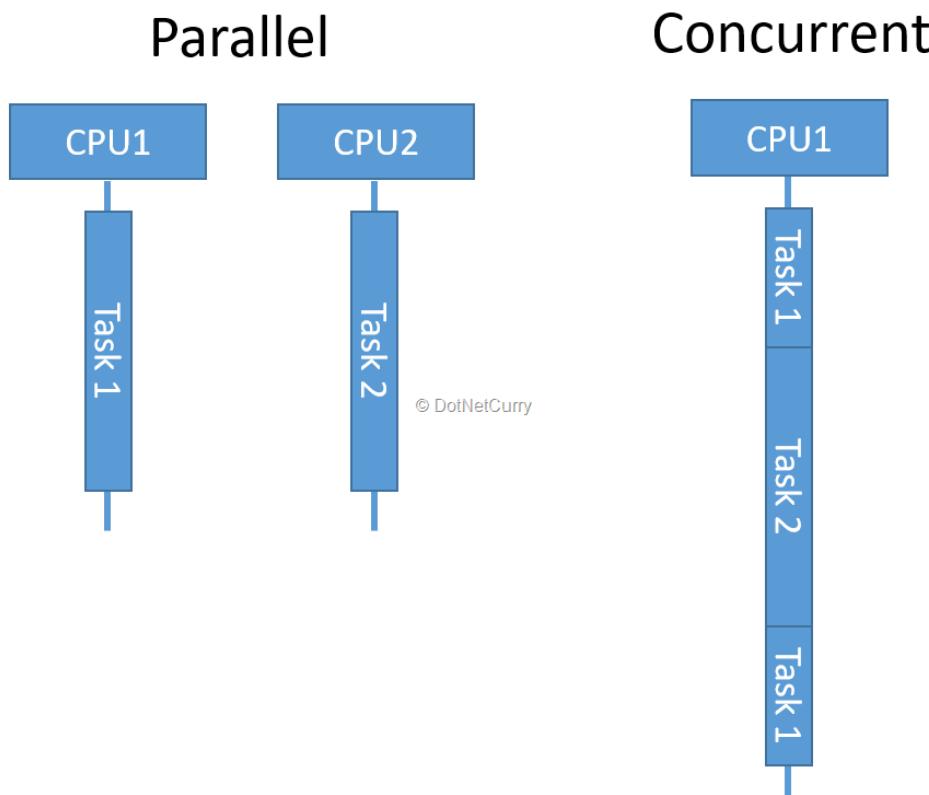
Conclusions

- Different methods are good at finding different kinds of bugs, but none is a silver bullet.
 - Fuzzing cannot be the only security assurance method used!
 - Static analysis
 - Manual reviews (code, design documents, etc.)
 - Regular unit/integration/system testing!
 - ...

Concurrency Bugs

Concurrent Programs

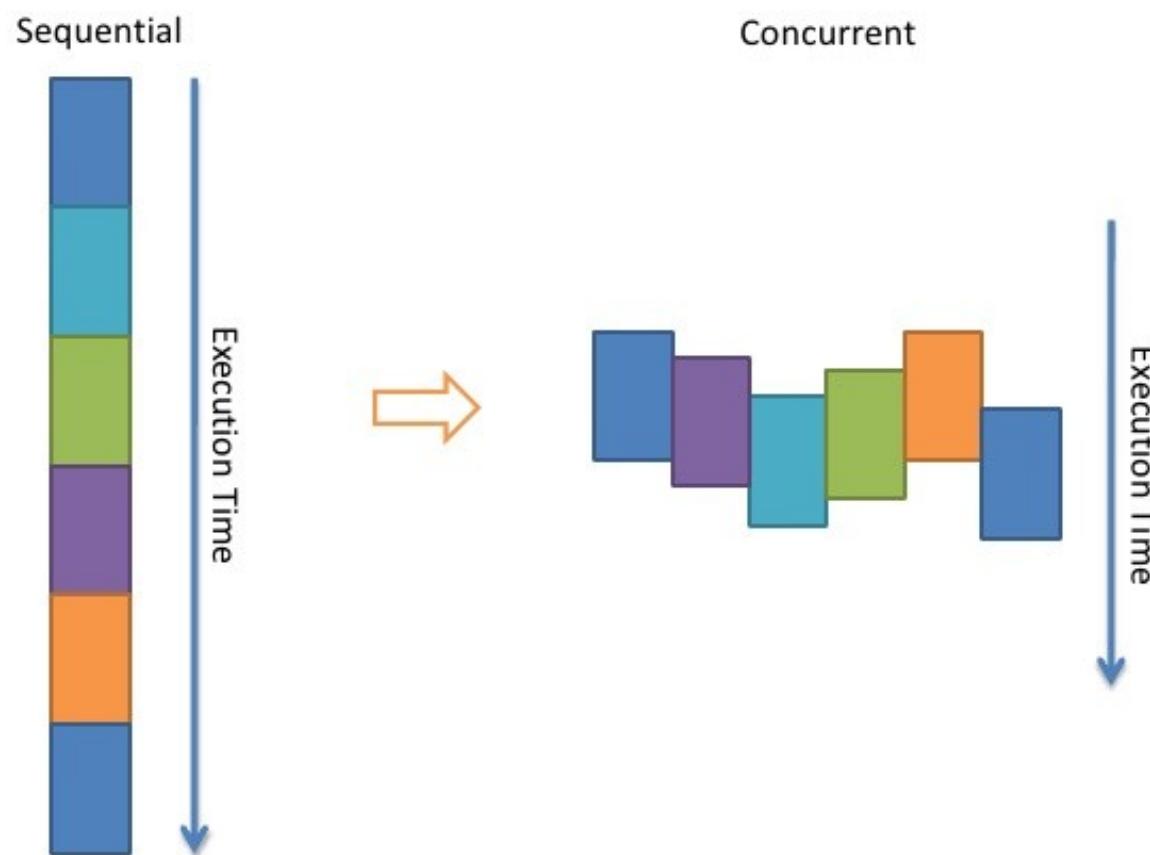
- Multi-core computers are common
- More programmers are having to write concurrent programs
- Concurrent programs have different bugs than sequential programs



Multi-Threaded programs

🏠 > Patterns > Multi-threading

Multi-threaded programming



Concurrency Bugs

- Knowing the types of concurrent bugs that actually occur in software will:
 - Help create better bug detection schemes
 - Inform the testing process software goes through
 - Provide information to program language designers
- Repeating concurrent bugs is difficult
- Testing is critical to find concurrency bugs

Concurrency Bug Types

- **Data race** : Occurs when two conflicting accesses to one shared variable are executed without proper synchronization, e.g., not protected by a common lock.
- **Deadlock** : Occurs when two or more operations circularly wait for each other to release the acquired resource (e.g., locks).
- **Atomicity Violation bugs** : Bugs which are caused by concurrent execution unexpectedly violating the atomicity of a certain code region.
- **Order Violation bugs** : Bugs that don't follow the programmer's intended order.

Data Race bug example

```
for (int i = 0; i <= 3; i++)  
    a[i] = a[i+1] + b[i];
```

Thread 1

```
a[0] = a[1] + b[0]  
a[1] = a[2] + b[1]
```

Thread 2

```
a[2] = a[3] + b[2]  
a[3] = a[4] + b[3]
```

$a[2]$ is updated in the second thread before $a[1]$ uses it in the first thread. Wrong $a[1]$ gets generated.

Preventing data races

- Using Locks or Atomic operations on shared variables.

Orig. Code with data race

```
public class Counter {  
    int counter;  
  
    public void increment() {  
        counter++;  
    }  
}
```

Data race on *counter* shared variable.
counter++ is a combination of 3 operations: reading the value, incrementing and writing the updated value.

Using Locks to prevent data race

```
public class SafeCounterWithLock {  
    private volatile int counter;  
  
    public synchronized void increment() {  
        counter++;  
    }  
}
```

Volatile keyword for reference visibility among threads.
The **synchronized** keyword acts as a lock and ensures that only one thread can enter the method at one time.

Preventing data races

- Using Atomic shared variables.

The most commonly used atomic variable classes in Java are **AtomicInteger**, **AtomicLong**, **AtomicBoolean**, and **AtomicReference**. These classes represent an int, long, boolean and object reference respectively which can be atomically updated. Methods exposed by these classes are *get()*, *set()*, *lazySet()*, *compareAndSet()*.

Using Atomics to prevent data race

```
public class SafeCounterWithAtomic {  
    private final AtomicInteger counter =  
        new AtomicInteger(0);  
    public int getValue() {  
        return counter.get();  
    }  
    public void increment() {  
        while(true) {  
            int existingValue = getValue();  
            int newValue = existingValue + 1;  
            if(counter.compareAndSet  
                (existingValue, newValue)) {  
                return;  
            }  
        }  
    }  
}
```

Deadlock example

Figure - 1

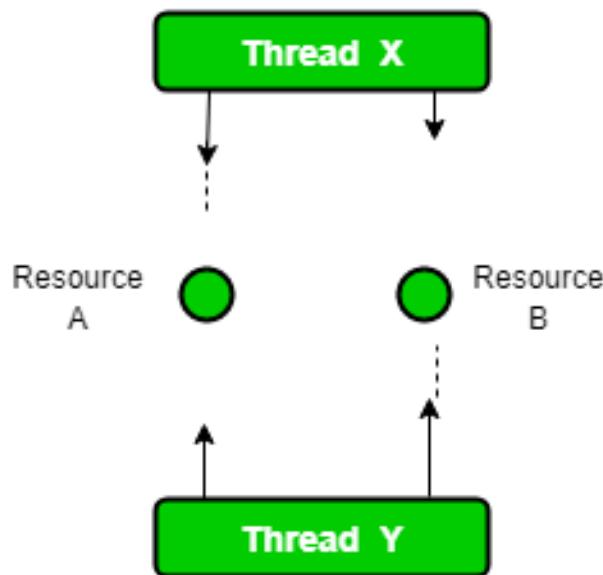
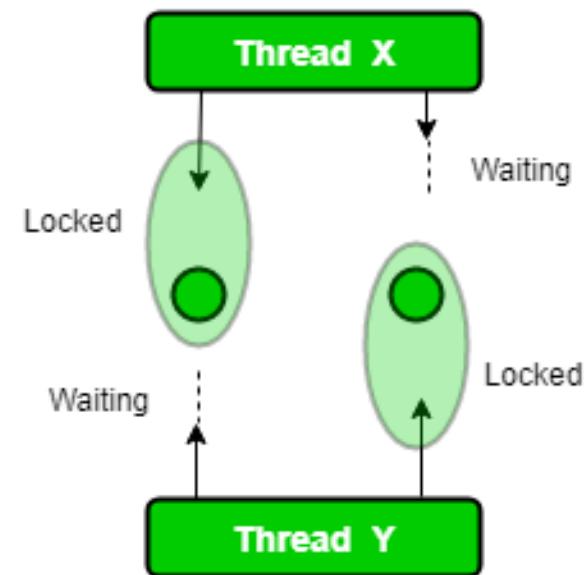


Figure - 2



Deadlock (cont.)

Deadlock

thread 1	thread 2
void f1()	void f2()
{	{
get(A);	get(B);
get(B);	get(A);
release(B);	release(A);
release(A);	release(B);
}	}

Deadlock prevention

- **Lock Ordering:** Deadlock occurs when multiple threads need the same locks but obtain them in different order. If you make sure that all locks are always taken in the same order by any thread, deadlocks cannot occur.
- **Lock Timeout:** Another deadlock prevention mechanism is to put a timeout on lock attempts meaning a thread trying to obtain a lock will only try for so long before giving up. If a thread does not succeed in taking all necessary locks within the given timeout, it will backup, free all locks taken, wait for a random amount of time and then retry. The random amount of time waited serves to give other threads trying to take the same locks a chance to take all locks, and thus let the application continue running without locking.

Atomicity violation example

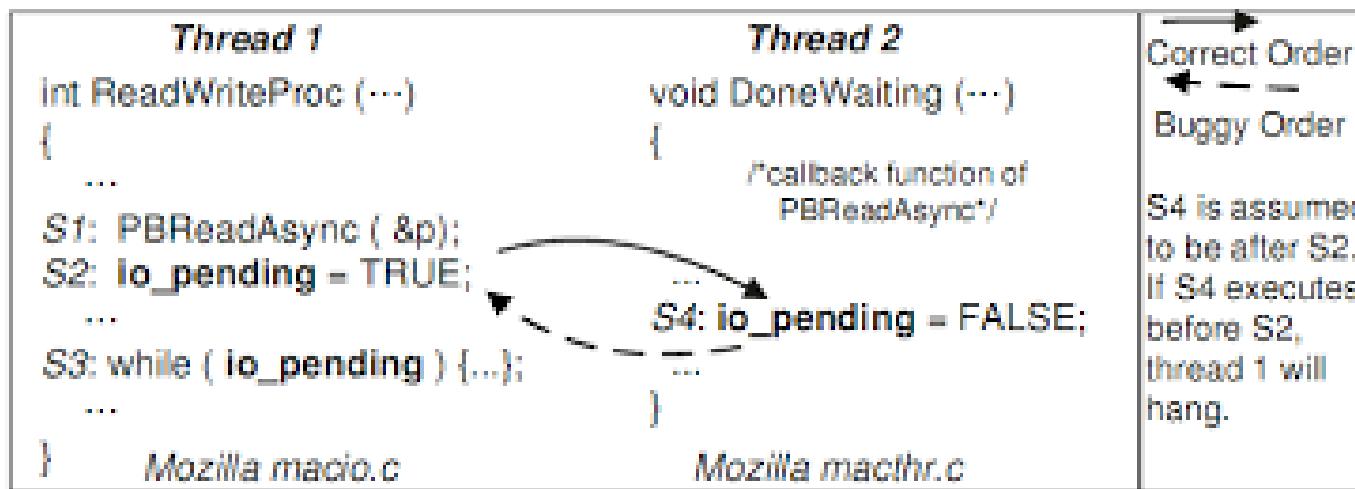
Atomicity violation detection techniques check if an observed execution is equivalent to any serial execution.

Case 1	Case 1
thread-1	thread-2
deposit(int val){	deposit(int val){
int tmp = bal;	int tmp = bal;
tmp = tmp + val;	tmp = tmp + val;
bal = tmp;	bal = tmp;
}	}

Case 2	Case 2
thread-1	thread-2
deposit(int val){	deposit(int val){
synchronized(o){	synchronized(o){
int tmp = bal;	int tmp = bal;
tmp = tmp + val;	tmp = tmp + val;
}	}
synchronized(o){	synchronized(o){
bal = tmp;	bal = tmp;
}	}
}	}

The programmer may be required to enforce sequential execution of operations as a whole to avoid atomicity violations.

Order Violation bug in Mozilla



Order violation bugs can be prevented using synchronisation variables and locks.

Course Review

Ajitha Rajan



Software Faults, Errors & Failures

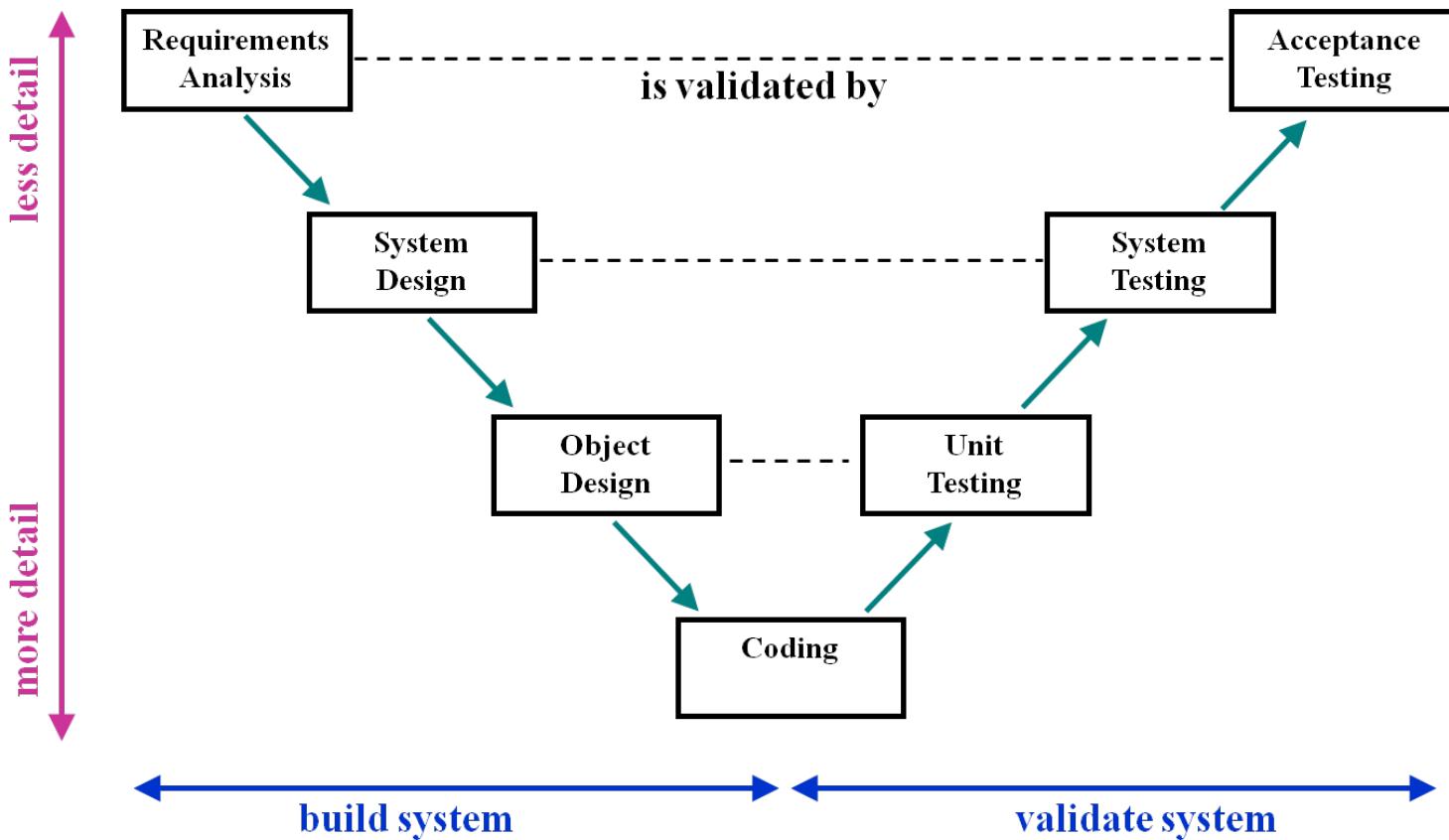
- **Software Fault** : A static defect in the software
- **Software Failure** : External, incorrect behavior with respect to the requirements or other description of the expected behavior
- **Software Error** : An incorrect internal state that is the manifestation of some fault

Summary: Why Do We Test Software ?

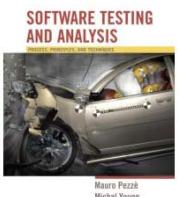
**A tester's goal is to eliminate faults
as early as possible**

- **Improve quality**
- **Reduce cost**
- **Preserve customer satisfaction**

V-model



Functional testing

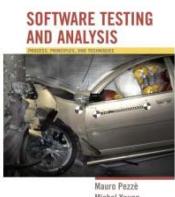


(c) 2007 Mauro Pezzè & Michal Young

Ch 10, slide 1

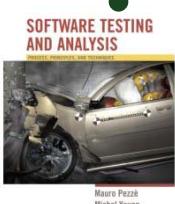
Functional testing

- 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



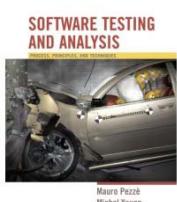
Systematic vs Random Testing

- 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

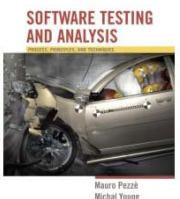


Functional testing: exploiting the specification

- 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

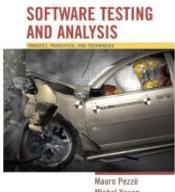


(c) 2007 Mauro Pezzè & Michal Young

Ch 11, slide 1

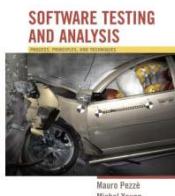
Combinatorial testing: Basic idea

- 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

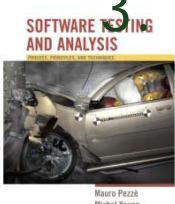
- Category-partition testing
 - separate (manual) identification of values that characterize the input space from (automatic) generation of combinations for test cases
- Pairwise testing
 - systematically test interactions among attributes of the program input space with a relatively small number of 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

Display Mode	Language	Fonts	Color	Screen size
full-graphics	English	Minimal	Monochrome	Hand-held
text-only	French	Standard	Color-map	Laptop
limited-bandwidth	Spanish	Document-loaded	16-bit	Full-size
	Portuguese		True-color	

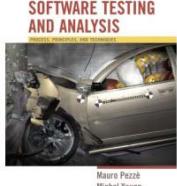
Pairwise combinations: 17 test cases

Language	Color	Display Mode	Fonts	Screen Size
English	Monochrome	Full-graphics	Minimal	Hand-held
English	Color-map	Text-only	Standard	Full-size
English	16-bit	Limited-bandwidth	-	Full-size
English	True-color	Text-only	Document-loaded	Laptop
French	Monochrome	Limited-bandwidth	Standard	Laptop
French	Color-map	Full-graphics	Document-loaded	Full-size
French	16-bit	Text-only	Minimal	-
French	True-color	-	-	Hand-held
Spanish	Monochrome	-	Document-loaded	Full-size
Spanish	Color-map	Limited-bandwidth	Minimal	Hand-held
Spanish	16-bit	Full-graphics	Standard	Laptop
Spanish	True-color	Text-only	-	Hand-held
Portuguese	-	-	Monochrome	Text-only
Portuguese	Color-map	-	Minimal	Laptop
Portuguese	16-bit	Limited-bandwidth	Document-loaded	Hand-held
Portuguese	True-color	Full-graphics	Minimal	Full-size
Portuguese	True-color	Limited-bandwidth	Standard	Hand-held



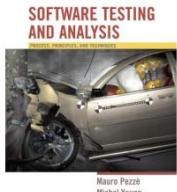
Next ...

- 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:

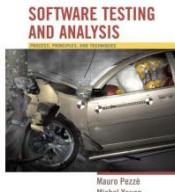
- Pre-conditions
- Post-conditions
- Definitions
- Variables
- Operations

Step 2:

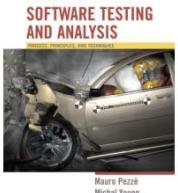
Derive a first set of test case specifications from pre-conditions, post-conditions and definitions

Step 3:

Complete the set of test case specifications using test catalogs



Structural Testing

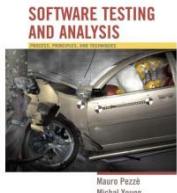


(c) 2007 Mauro Pezzè & Michal Young

Ch 12, slide 1

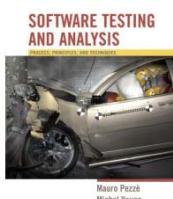
“Structural” testing

- 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

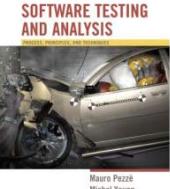
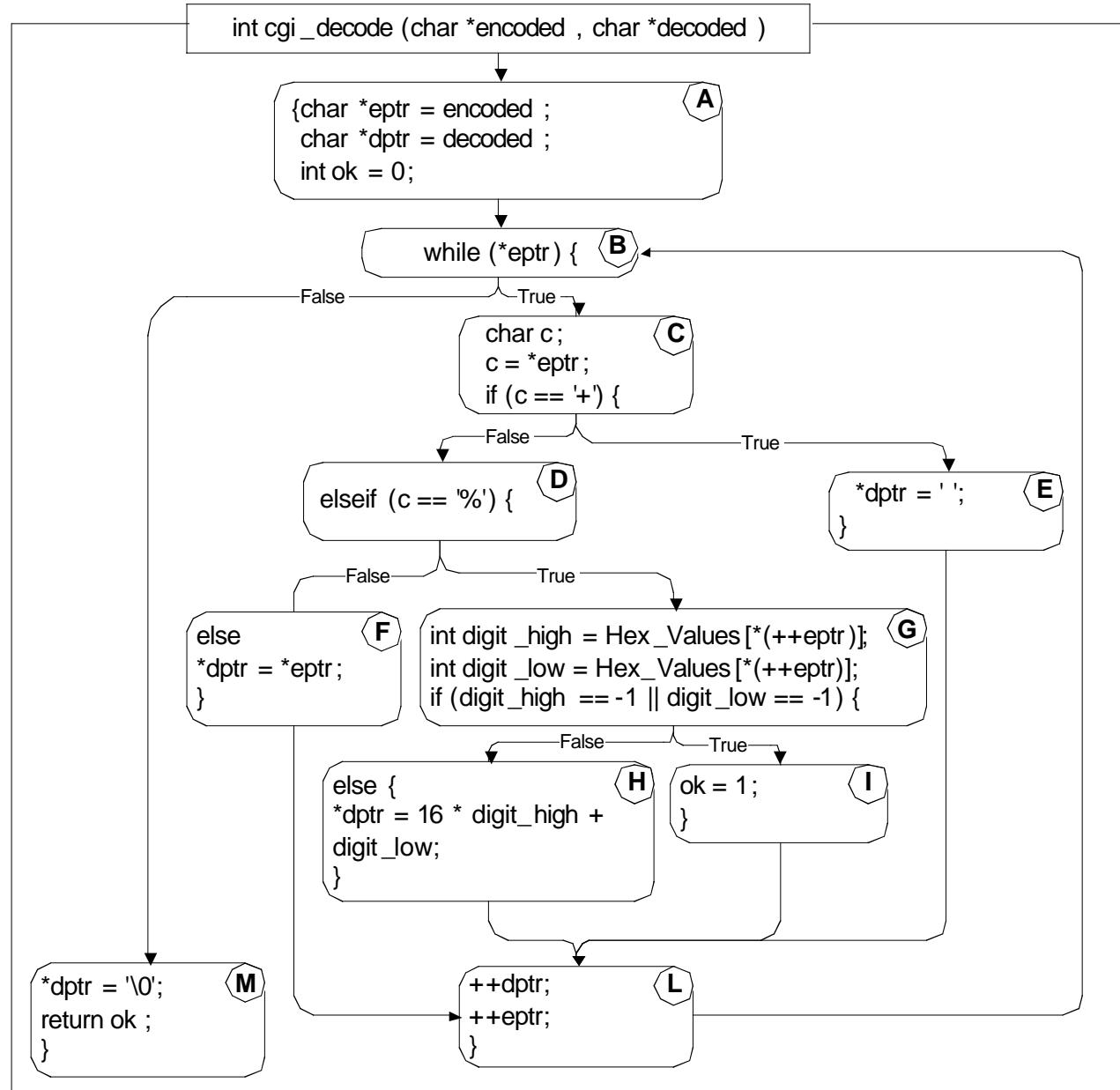


Example

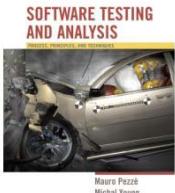
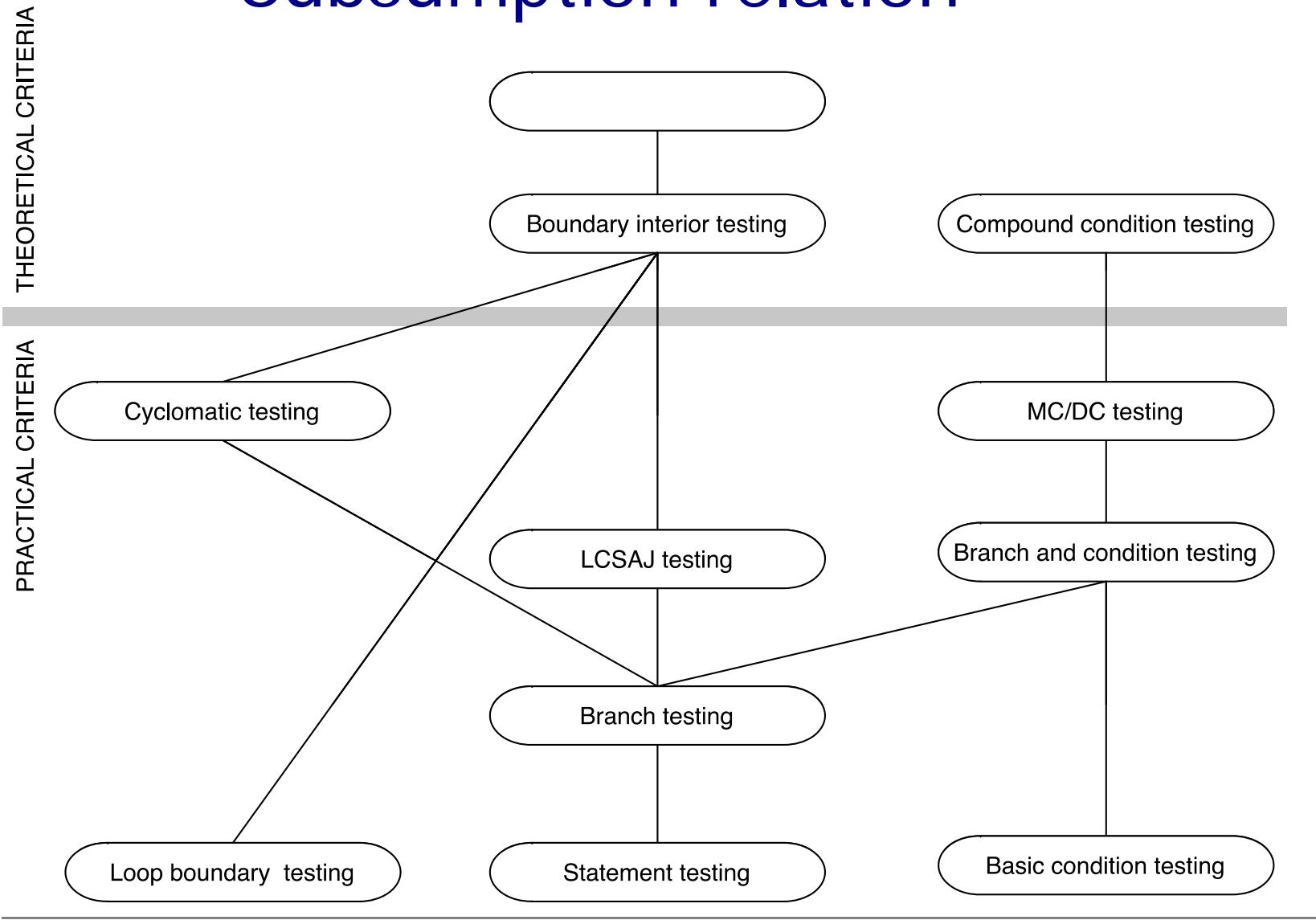
$T_0 =$
 $\{ "", "test",$
 $"test+case%1Dadequacy"\}$
 $17/18 = 94\% \text{ Stmt Cov.}$

$T_1 =$
 $\{"adequate+test%0Dexecuti$
 $\text{on}%7U"\}$
 $18/18 = 100\% \text{ Stmt Cov.}$

$T_2 =$
 $\{"%3D", "%A", "a+b",$
 $"test"\}$
 $18/18 = 100\% \text{ Stmt Cov.}$

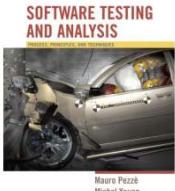


Subsumption relation



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



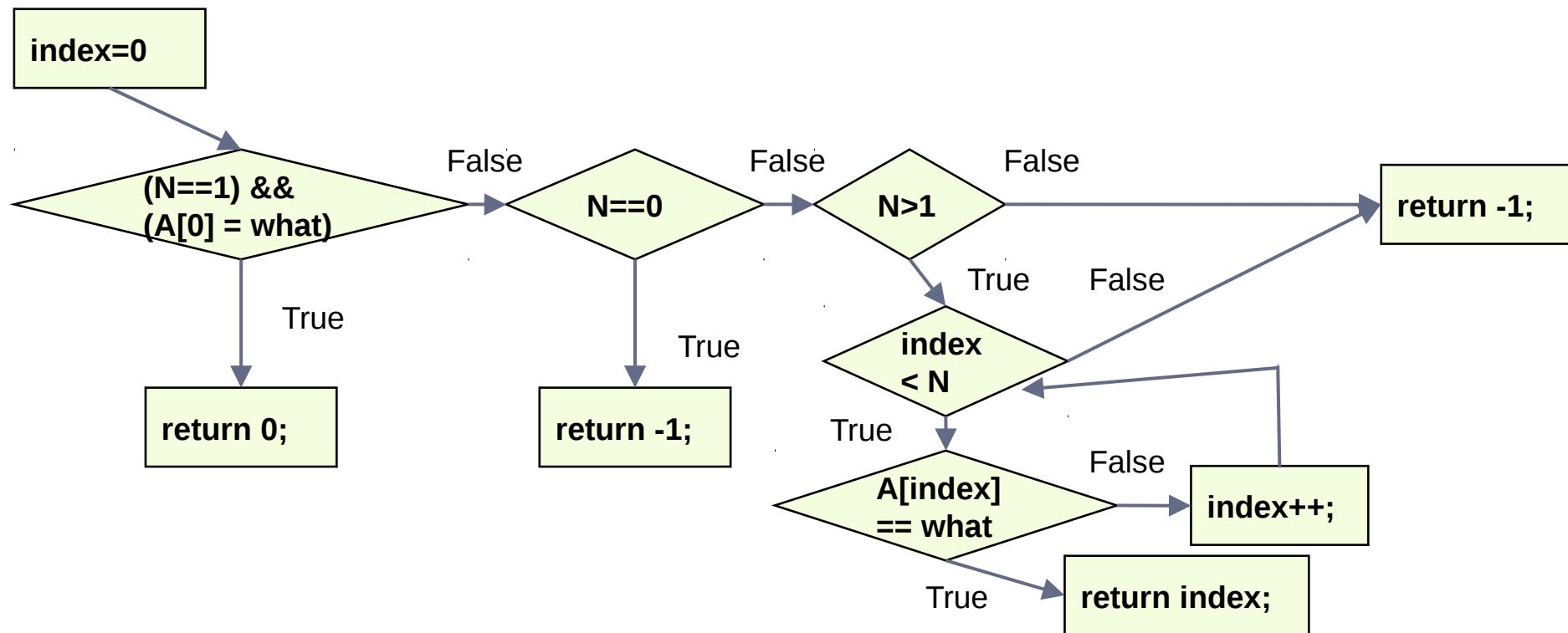
Activity

- Write tests that provide statement, branch, and basic condition coverage over the following code:

```
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;  
}
```

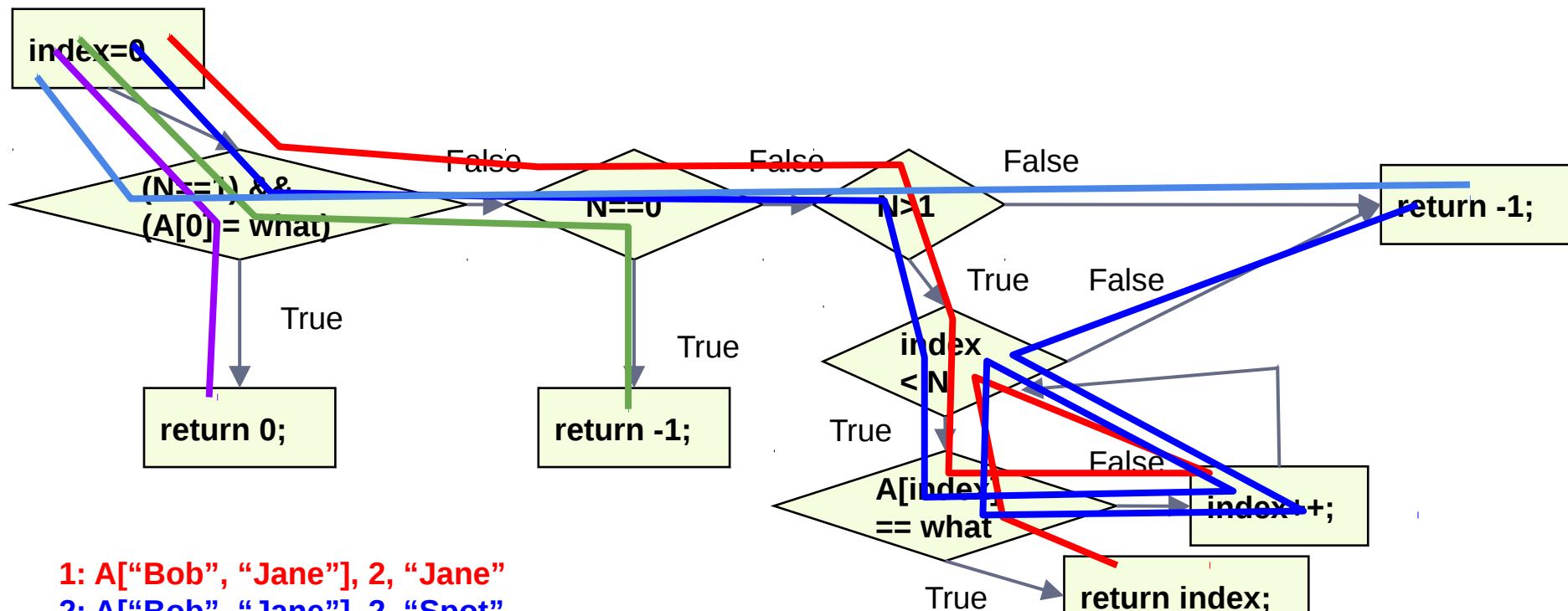
Activity - Possible Solution

- Write tests that provide statement, branch, and basic condition coverage over the following code:



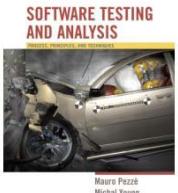
Activity - Possible Solution

- Write tests that provide statement, branch, and basic condition coverage over the following code:



- 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

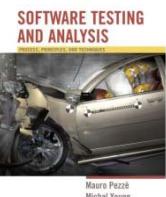
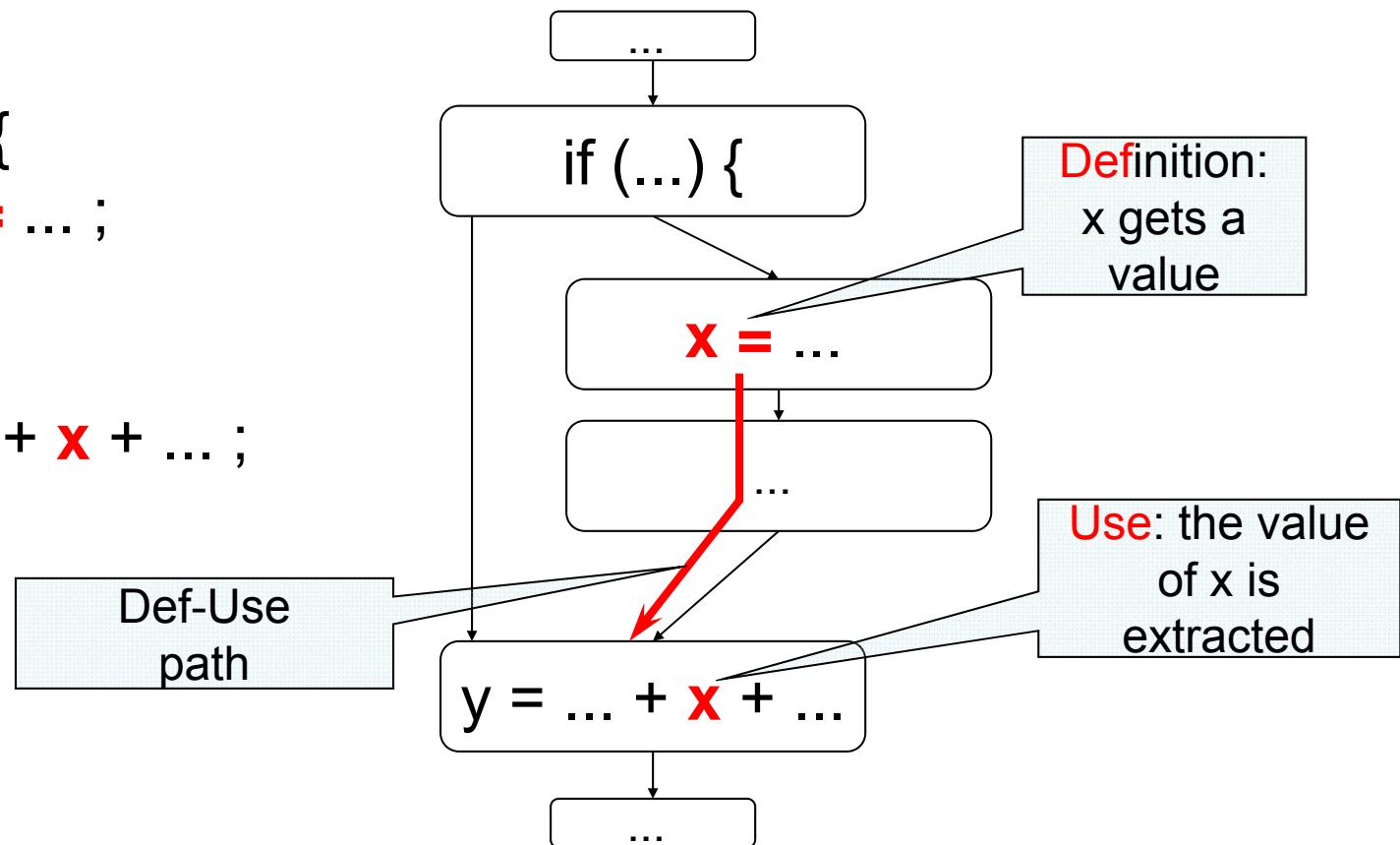


(c) 2007 Mauro Pezzè & Michal Young

Ch 6, slide 1

Def-Use Pairs

```
...
if (...) {
    X = ... ;
}
y = ... + X + ... ;
```

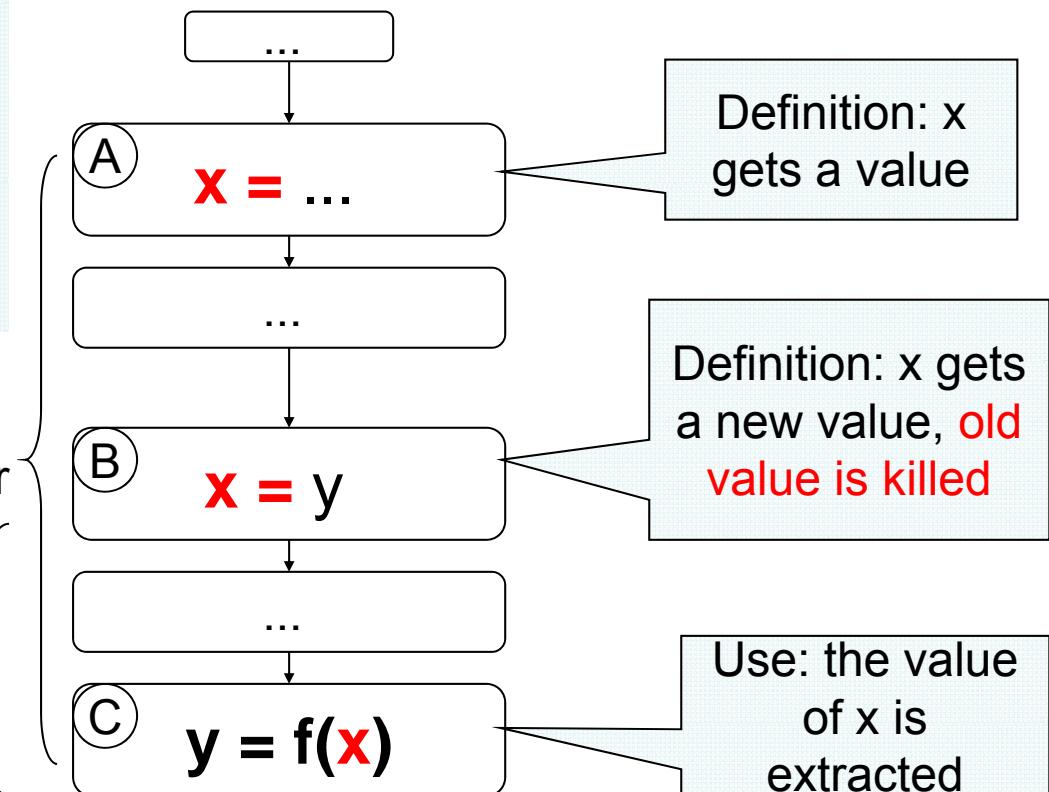


Definition-Clear or Killing

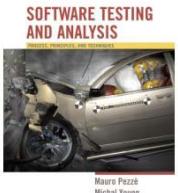
```
x = ... // A: def x  
q = ...  
x = y; // B: kill x, def x  
z = ...  
y = f(x); // C: use x
```

Path A..C is not definition-clear

Path B..C is definition-clear



Data flow testing

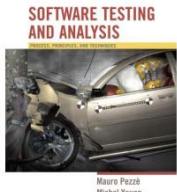


(c) 2007 Mauro Pezzè & Michal Young

Ch 13, slide 1

Terms

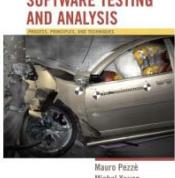
- DU pair: a pair of *definition* and *use* for some variable, such that at least one DU path exists from the definition to the use
 - x = ... is a *definition* of x
 - = ... x ... is a *use* of x
- DU path: a definition-clear path on the CFG starting from a definition to a use of a same variable
 - Definition clear: Value is not replaced on path
 - Note - loops could create infinite DU paths between a def and a use



Adequacy criteria

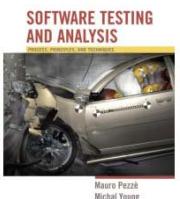
- All DU pairs: Each DU pair is exercised by at least one test case
- All DU paths: Each *simple* (non looping) DU path is exercised by at least one test case
- All definitions: For each definition, there is at least one test case which exercises a DU pair containing it
 - (Every computed value is used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

Corresponding coverage fractions can also be defined



Testing Object Oriented Software

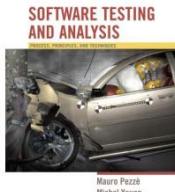
Chapter 15



Characteristics of OO Software

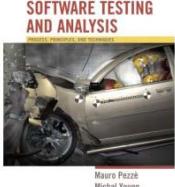
Typical OO software characteristics that impact testing

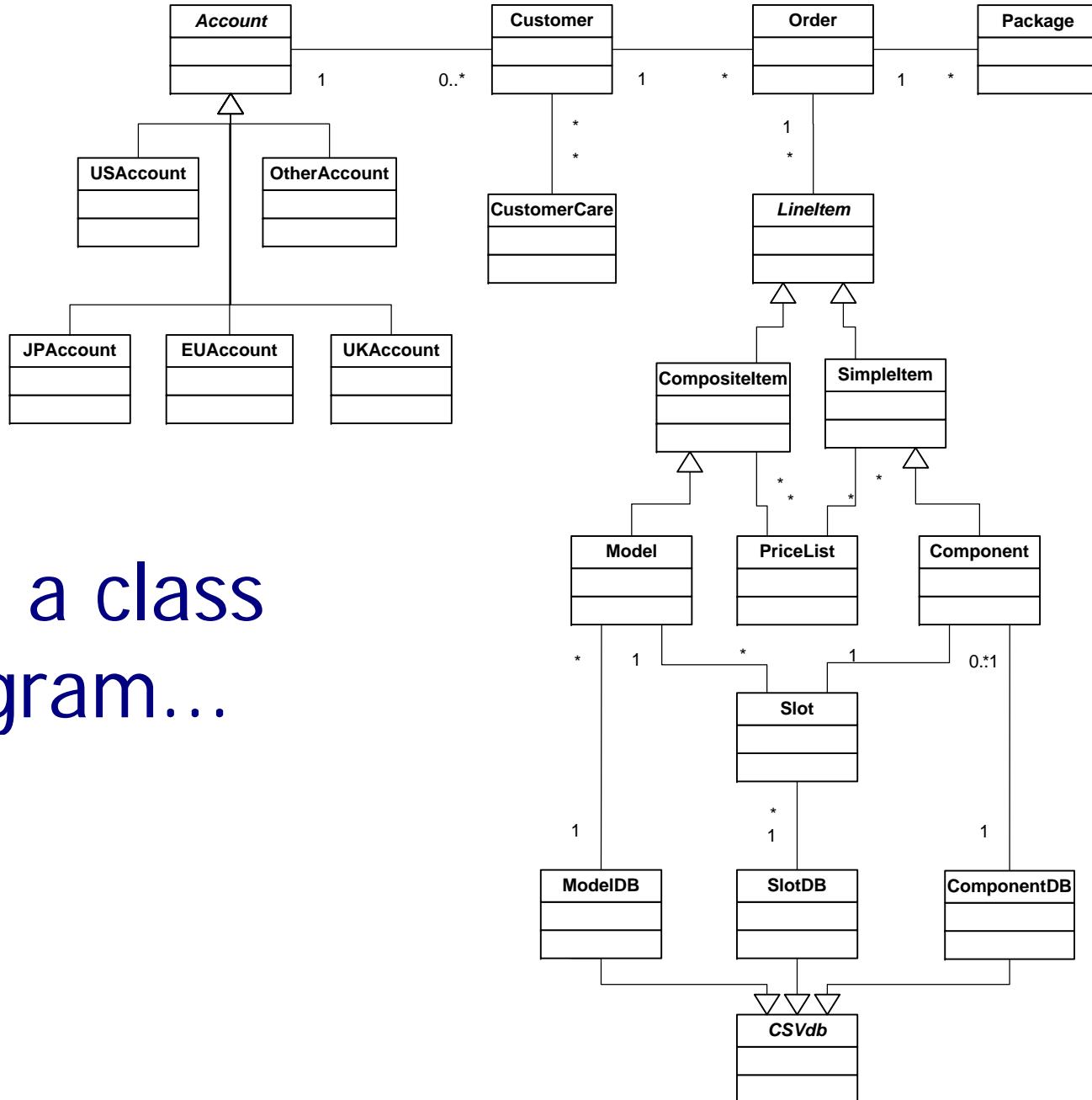
- State dependent behavior
- Encapsulation
- Inheritance
- Polymorphism and dynamic binding
- Abstract and generic classes
- Exception handling



Interclass Testing

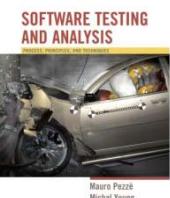
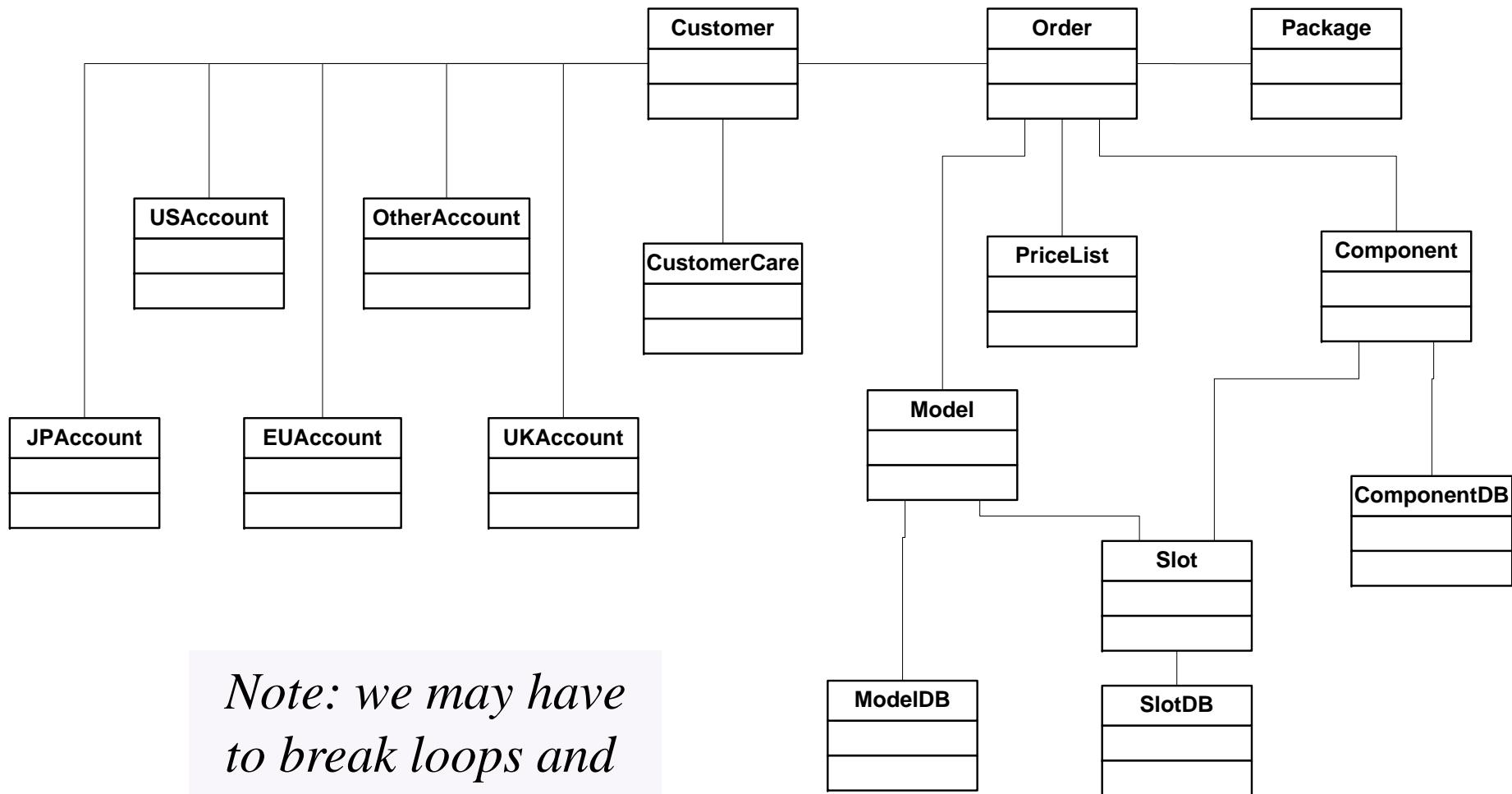
- The first level of *integration testing* for object-oriented software
 - Focus on interactions between classes
- Bottom-up integration according to “depends” relation
 - A depends on B: Build and test B, then A
- Start from use/include hierarchy
 - Implementation-level parallel to logical “depends” 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”





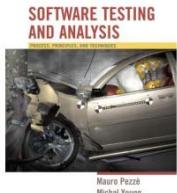
from a class
diagram...

....to a hierarchy



Intraclass 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

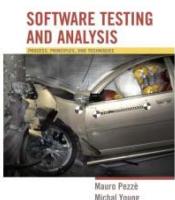
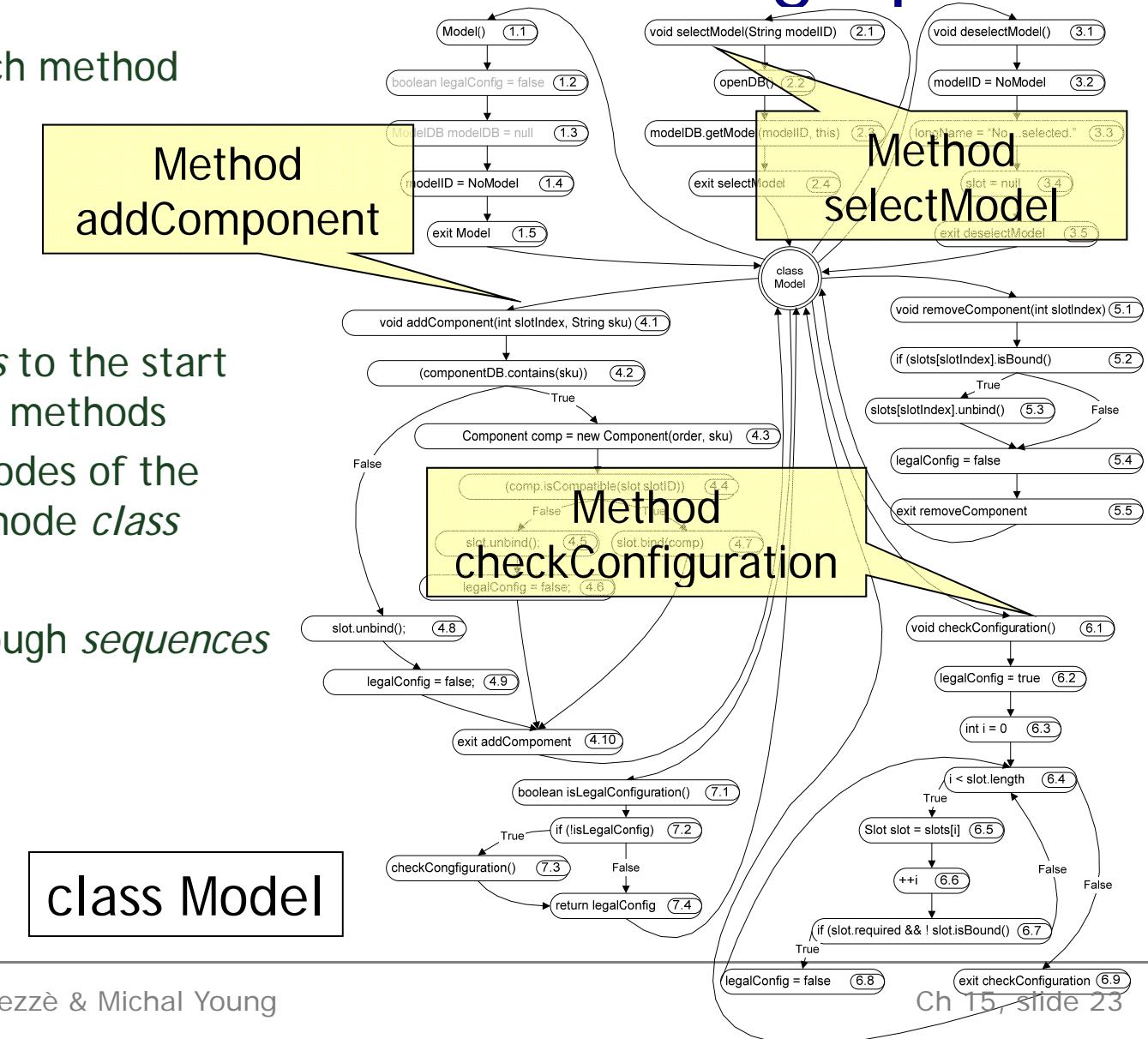
Control flow for each method

+
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

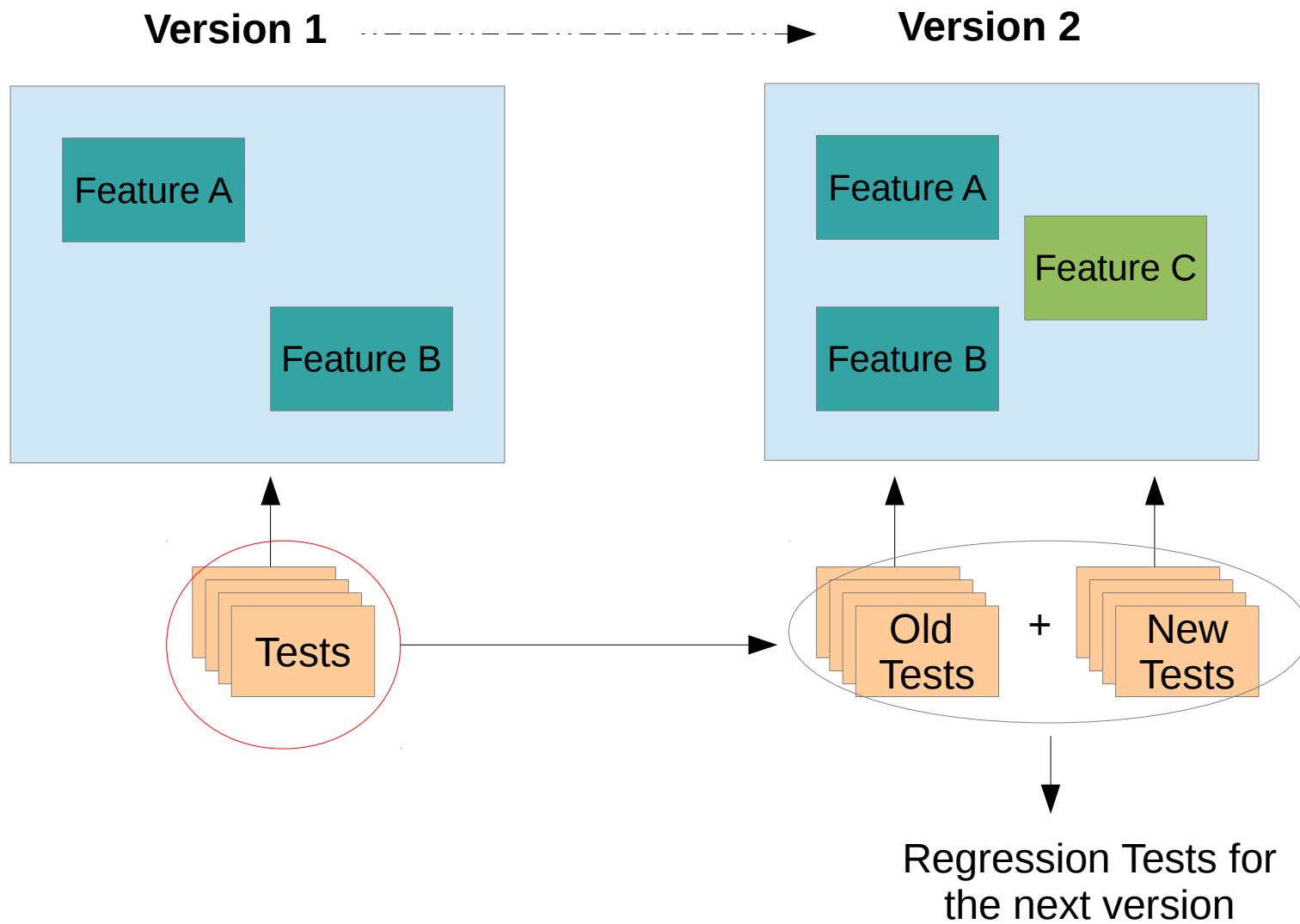
Example of Mutation Operators I

- 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

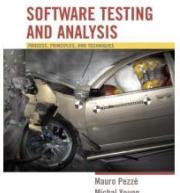
Example



Regression Test Optimization

- Re-test All
- Regression Test Selection
- Regression Test Set Minimisation
- Regression Test Set Prioritisation

Integration and Component-based Software Testing

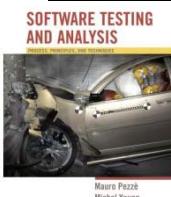


(c) 2007 Mauro Pezzè & Michal Young

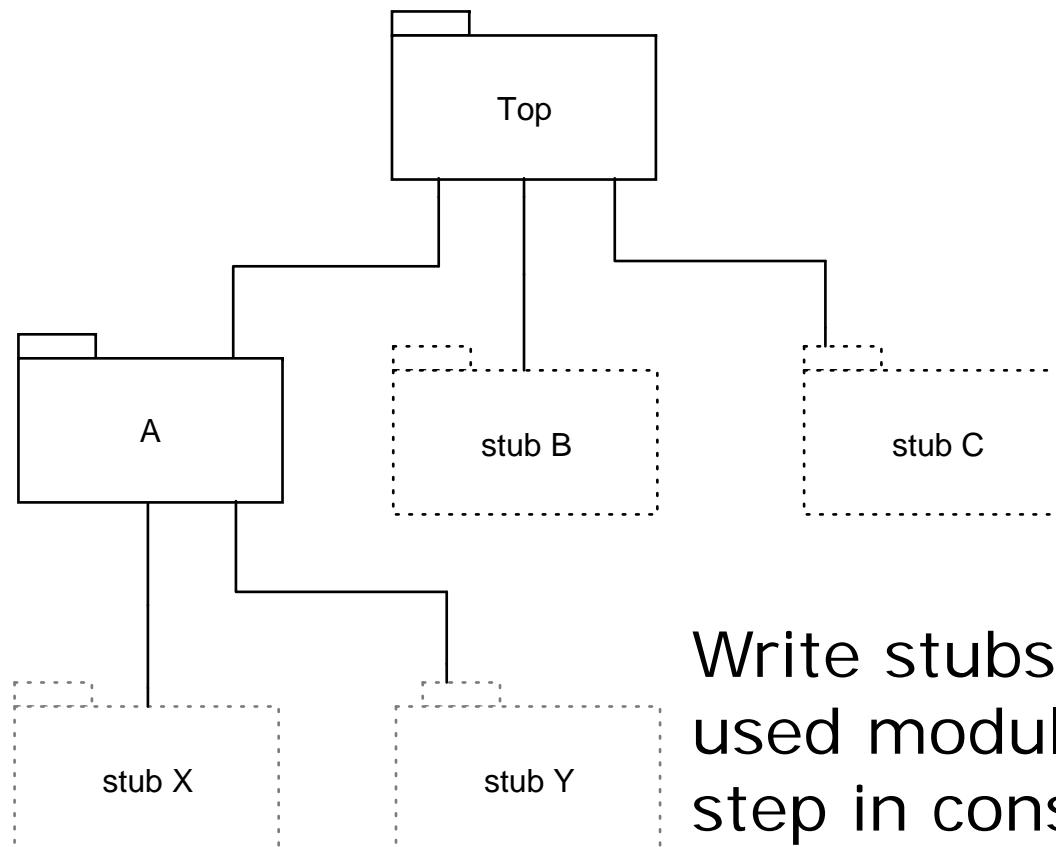
Ch 21, slide 1

What is integration testing?

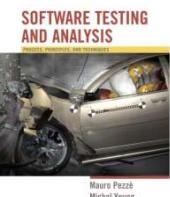
	Module test	Integration test	System test
Specification:	Module interface	Interface specs, module breakdown	Requirements specification
Visible structure:	Coding details	Modular structure (software architecture)	— none —
Scaffolding required:	Some	Often extensive	Some
Looking for faults in:	Modules	Interactions, compatibility	System functionality



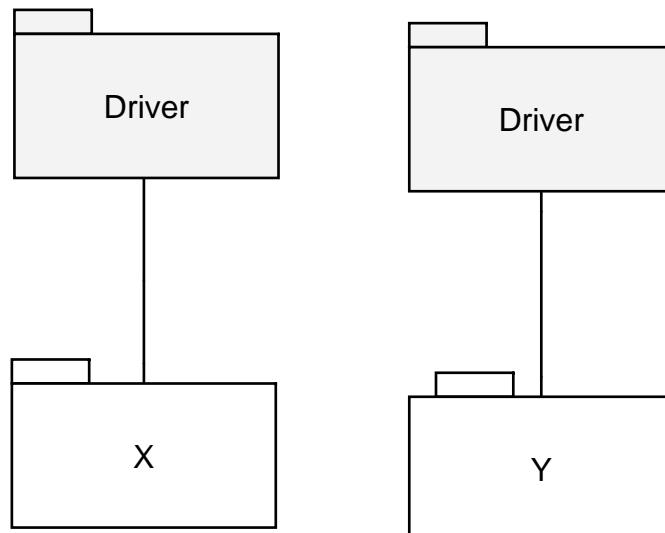
Top down ..



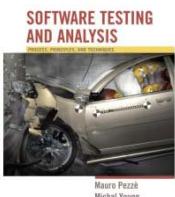
Write stubs of called or used modules at each step in construction



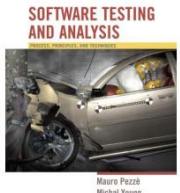
Bottom Up ..



... but we must
construct drivers for
each module (as in
unit testing) ...



System, Acceptance, and Regression Testing



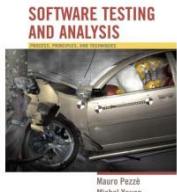
(c) 2007 Mauro Pezzè & Michal Young

Ch 22, slide 1

System Testing

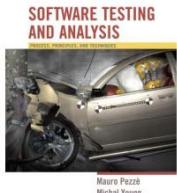
- Key characteristics:
 - Comprehensive (the whole system, the whole spec)
 - Based on specification of observable behavior
 - Verification against a requirements specification, not validation, and not opinions
 - Independent of design and implementation

Independence: Avoid repeating software design errors in system test design



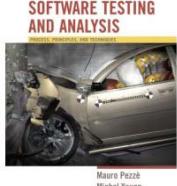
Global Properties

- 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



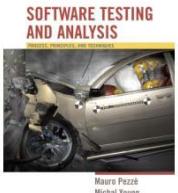
Context-Dependent Properties

- 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

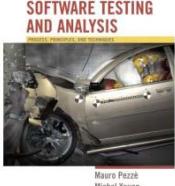


(c) 2007 Mauro Pezzè & Michal Young

Ch 22, slide 13

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



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

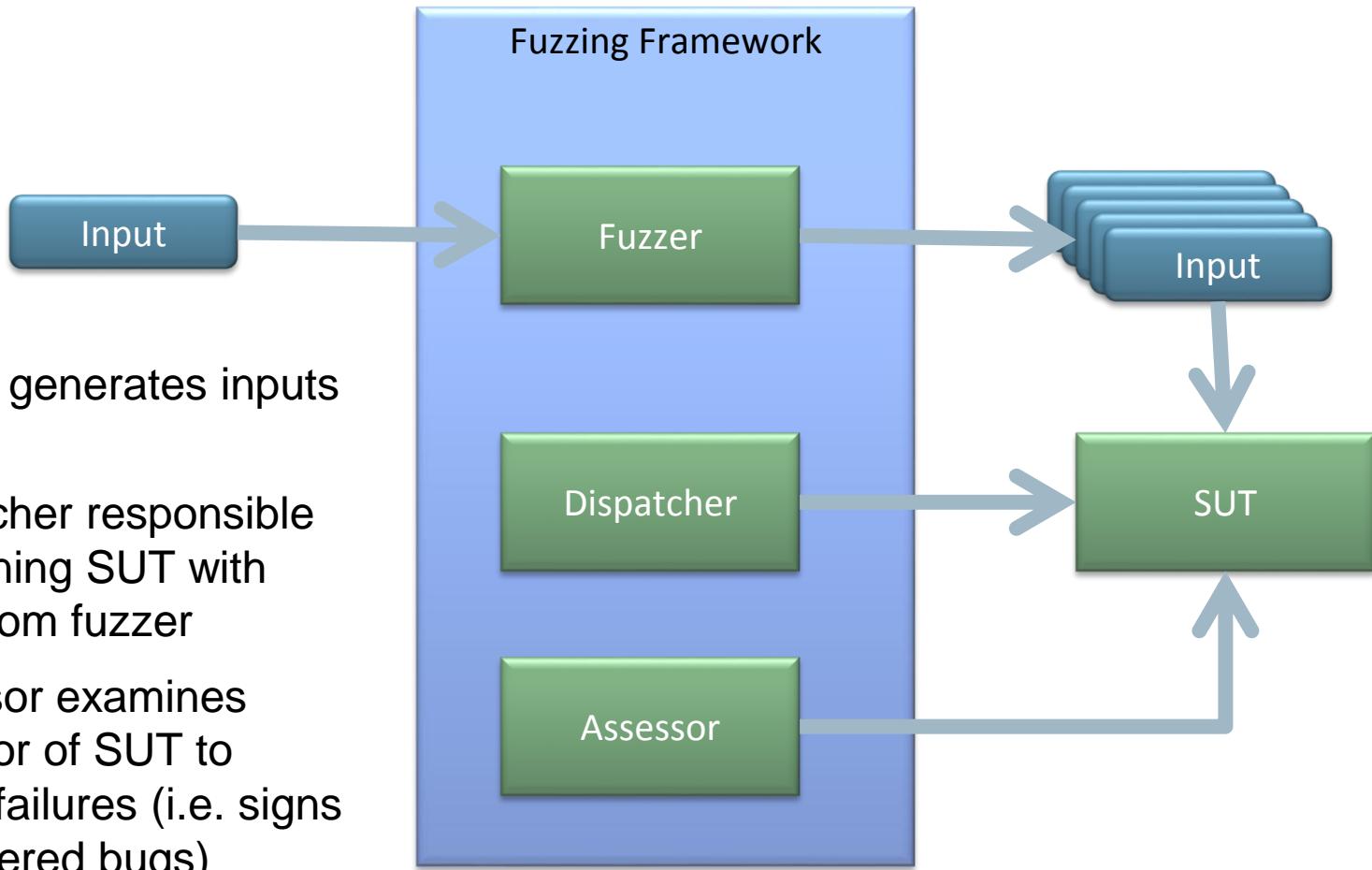
Two common approaches:

- 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



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.

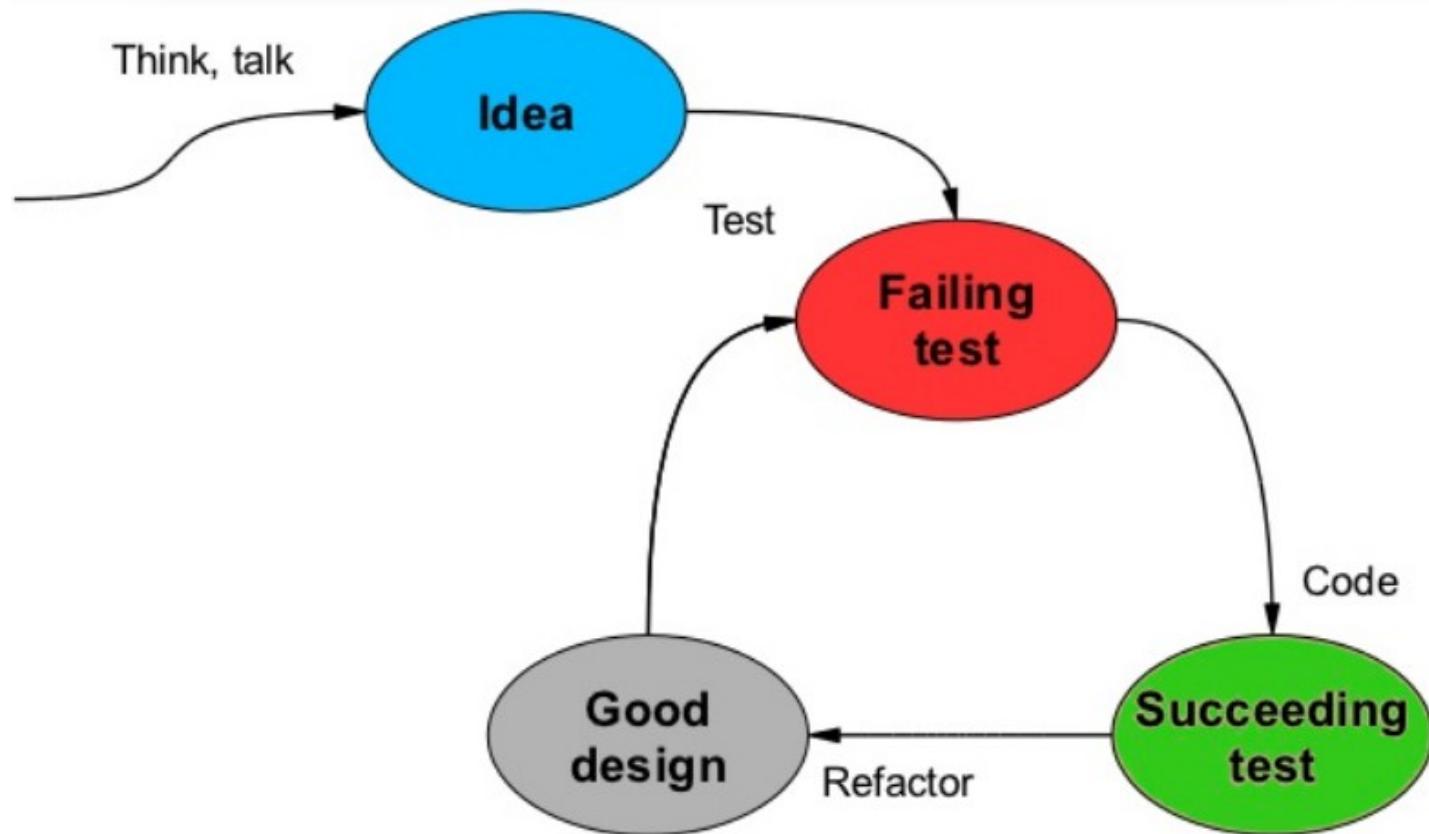
```
if(condition1)
    if(condition2)
        if(condition3)
            if(condition4)
                bug();
```

Mutational fuzzer would need to guess correct values of all four conditions in one go to reach bug!

Concurrency Bug Types

- **Data race** : Occurs when two conflicting accesses to one shared variable are executed without proper synchronization, e.g., not protected by a common lock.
- **Deadlock** : Occurs when two or more operations circularly wait for each other to release the acquired resource (e.g., locks).
- **Atomicity Violation bugs** : Bugs which are caused by concurrent execution unexpectedly violating the atomicity of a certain code region.
- **Order Violation bugs** : Bugs that don't follow the programmer's intended order.

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

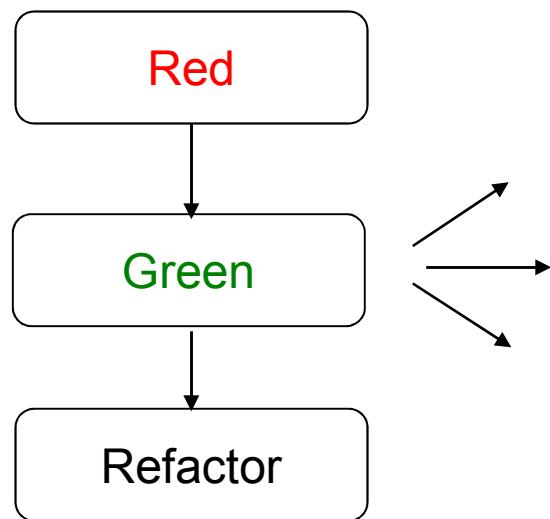
□ Write Functional Code

- 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)



See it fail
because there's
no dev code

See it fail
because no logic
is implemented

See the
test pass

TDD

