Course Review

Ajitha Rajan

School of informatics

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**
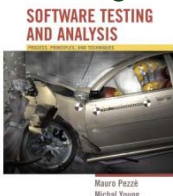
Functional testing
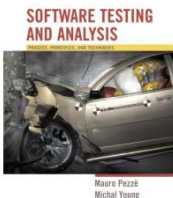
Functional testing

- Functional testing: Deriving test cases from program specifications
    - *Functional* refers to the source of information used in test case design, not to what is tested

- *Also known as*:

    - specification-based testing (from specifications)

    - black-box testing (no view of the code)

- Functional specification = description of intended program behavior
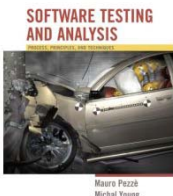
    - either formal or informal
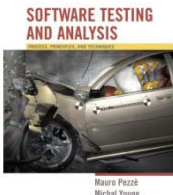
Systematic vs Random Testing

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

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

- ## Functional testing is systematic testing

Functional testing: exploiting the specification

- Functional testing uses the specification (formal or informal) to partition the input space

    - E.g., specification of "roots" program suggests division between cases with zero, one, and two real roots

- Test each category, and boundaries between categories

    - No guarantees, but experience suggests failures often lie at the boundaries (as in the "roots" program)
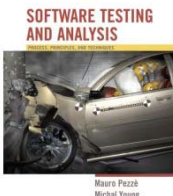
Combinatorial testing
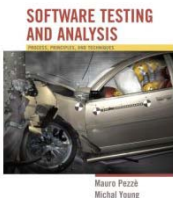
Combinatorial testing: Basic idea

- Identify distinct attributes that can be varied
  - In the data, environment, or configuration
  - Example: browser could be "IE" or "Firefox", operating system could be "Vista", "XP", or "OSX"
- Systematically generate combinations to be tested
  - Example: IE on Vista, IE on XP, Firefox on Vista, Firefox on OSX, …
- Rationale: Test cases should be varied and include possible "corner cases"

Key ideas in combinatorial approaches

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

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

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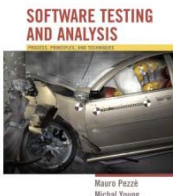
Category partition (manual steps)

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

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

3. Introduce constraints
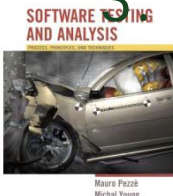
Example: Display Control

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

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

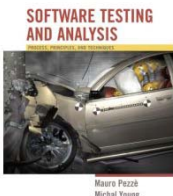| Language	Color	Display Mode	Fonts	Screen Size
English	Monochrome	Full-graphics	Minimal	Hand-held
English	Color-map	Text-only	Standard	Full-size
English	16-bit	Limited-bandwidth	-	Full-size
English	True-color	Text-only	Document-loaded	Laptop
French	Monochrome	Limited-bandwidth	Standard	Laptop
French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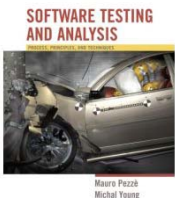
Catalog based testing

- 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

Step1:

Analyze the initial specification to identify simple elements:
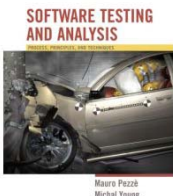
- Pre-conditions

- Post-conditions
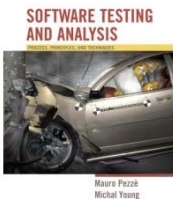
- Definitions

- Variables

- Operations

Step 2:

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

Step 3:

Complete the set of test case specifications using test catalogs

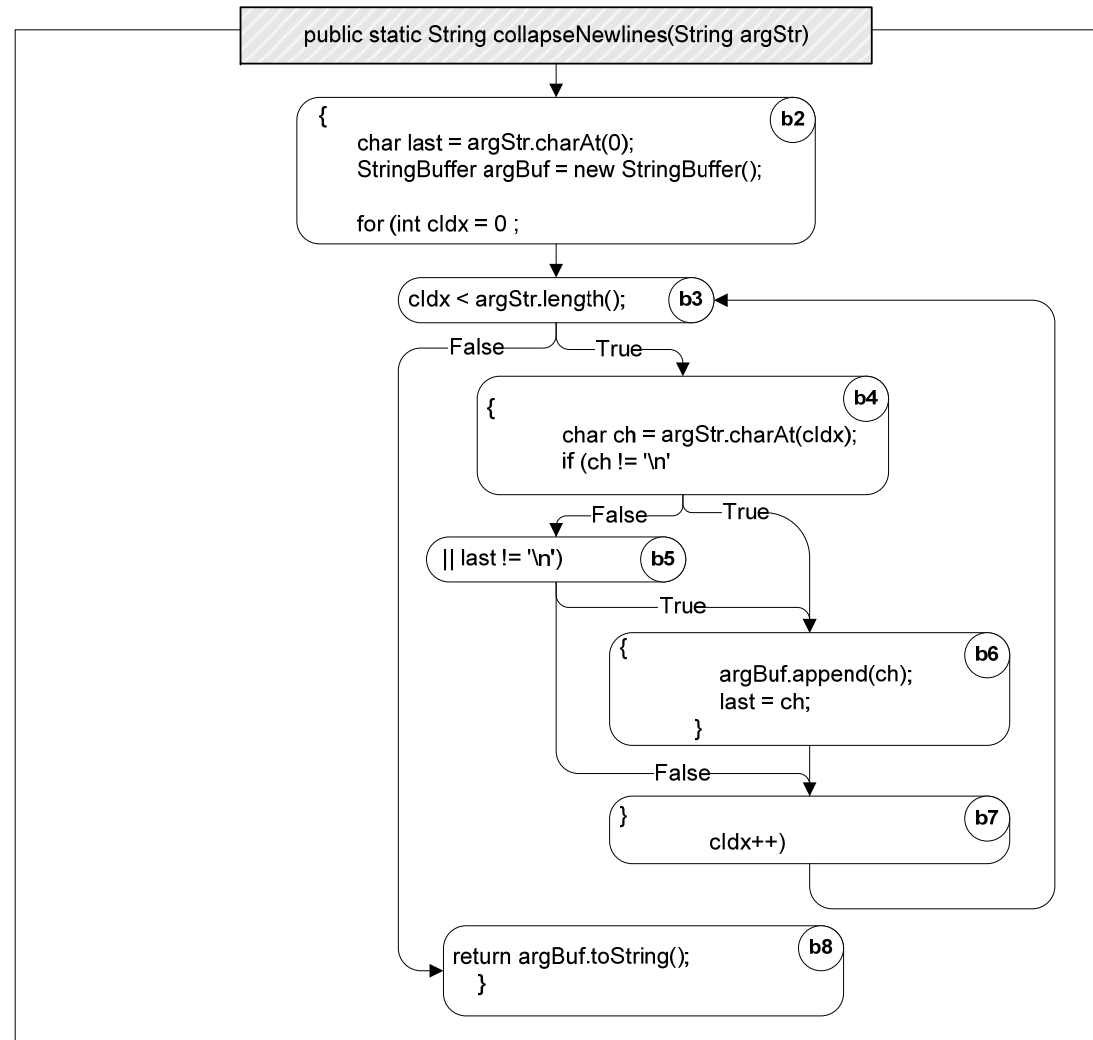# Finite Models

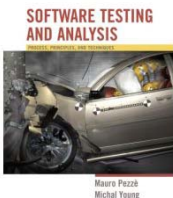# Example of Control Flow Graph

```
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();
    }
```
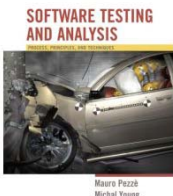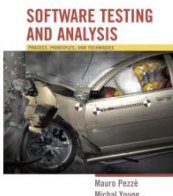
Structural Testing

"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.

SOFTWARE TESTING
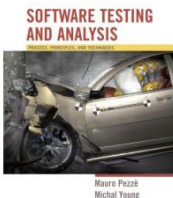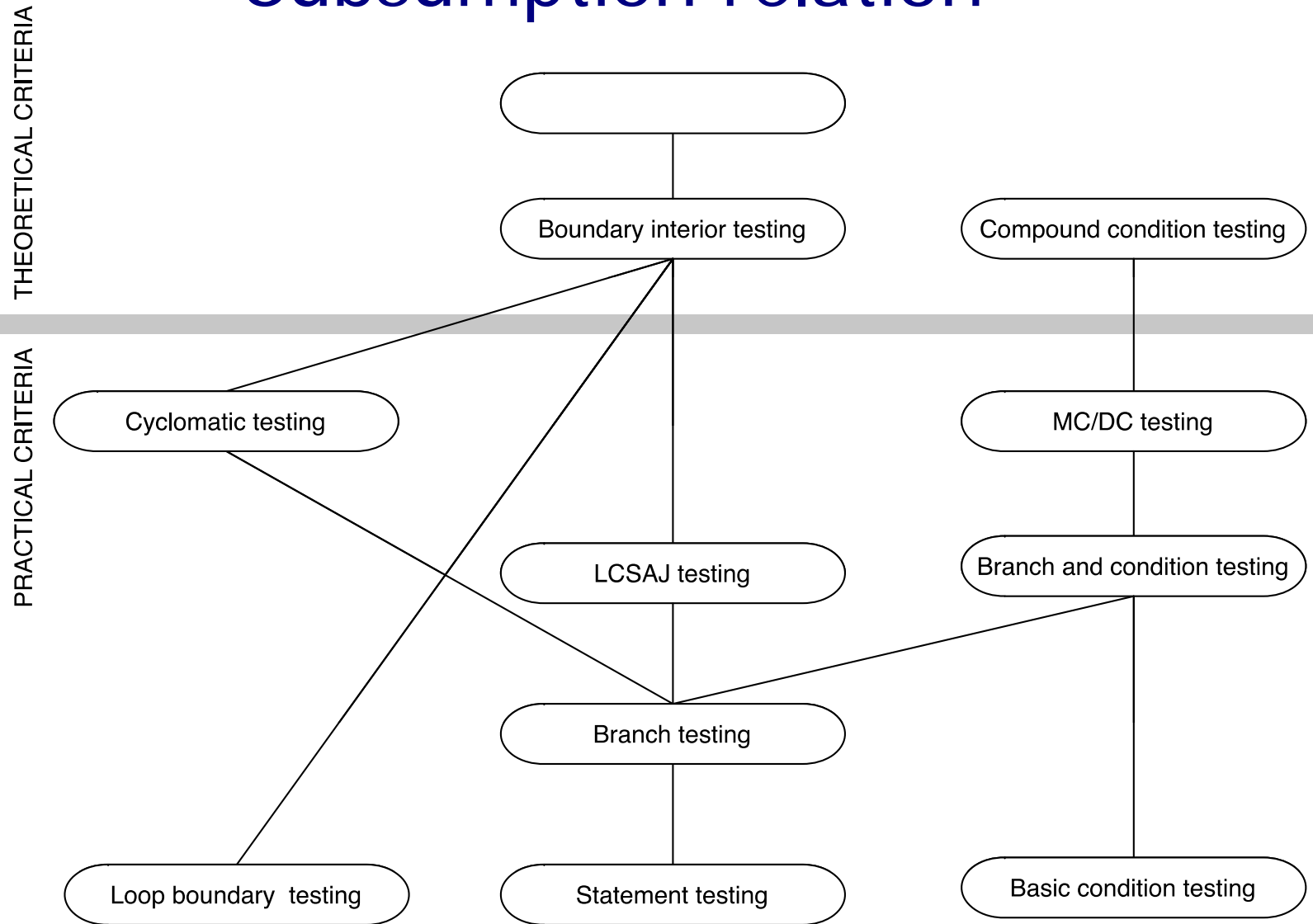AND ANALYSIS

Mauro Pezzè
Michal Young

Structural testing *complements* functional testing

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

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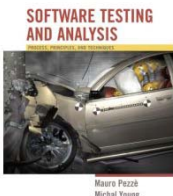 - Typical case: missing path faults

Subsumption relation

THEORETICAL CRITERIA

PRACTICAL CRITERIA

Boundary interior testing

Compound condition testing

Cyclomatic testing

MC/DC testing

LCSAJ testing

Branch and condition testing

Branch testing

Loop boundary testing

Statement testing

Basic condition testing

SOFTWARE TESTING
AND ANALYSIS

Mauro Pezzè
Michal Young

Summary

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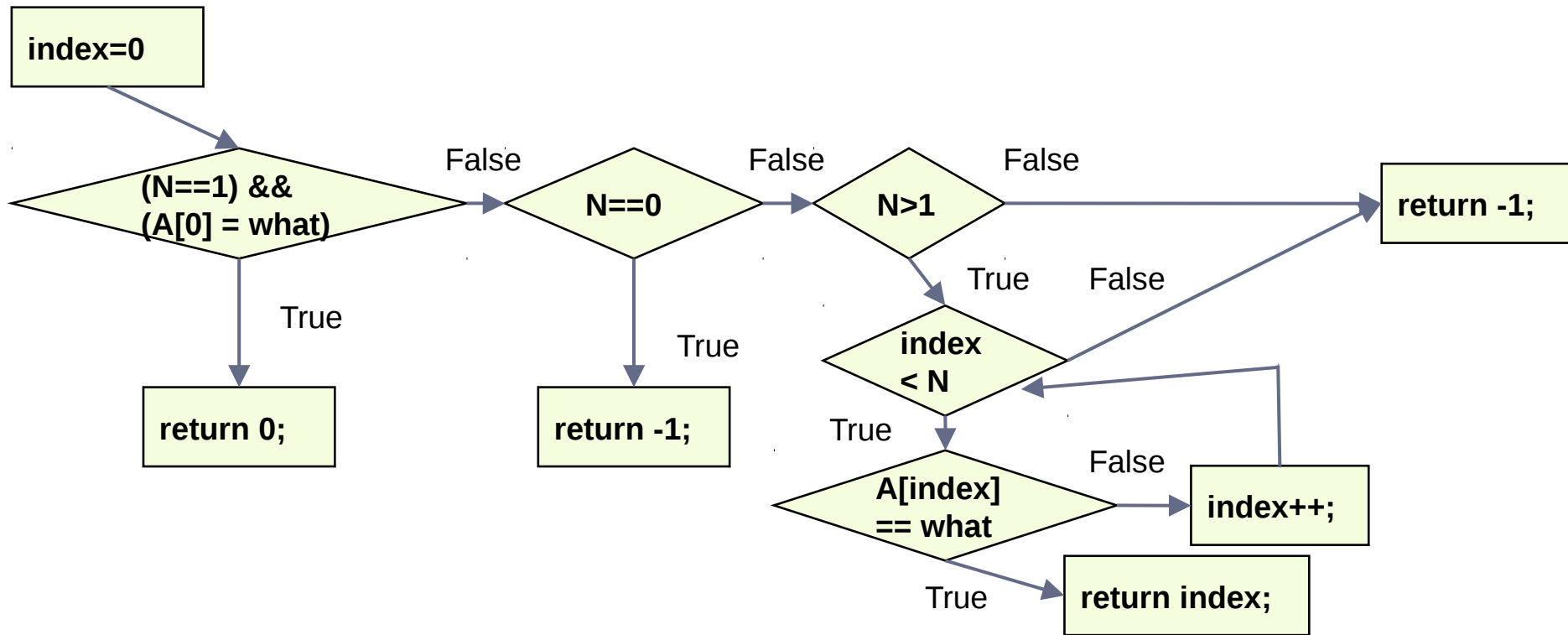 - May drive test improvement

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;
}
```
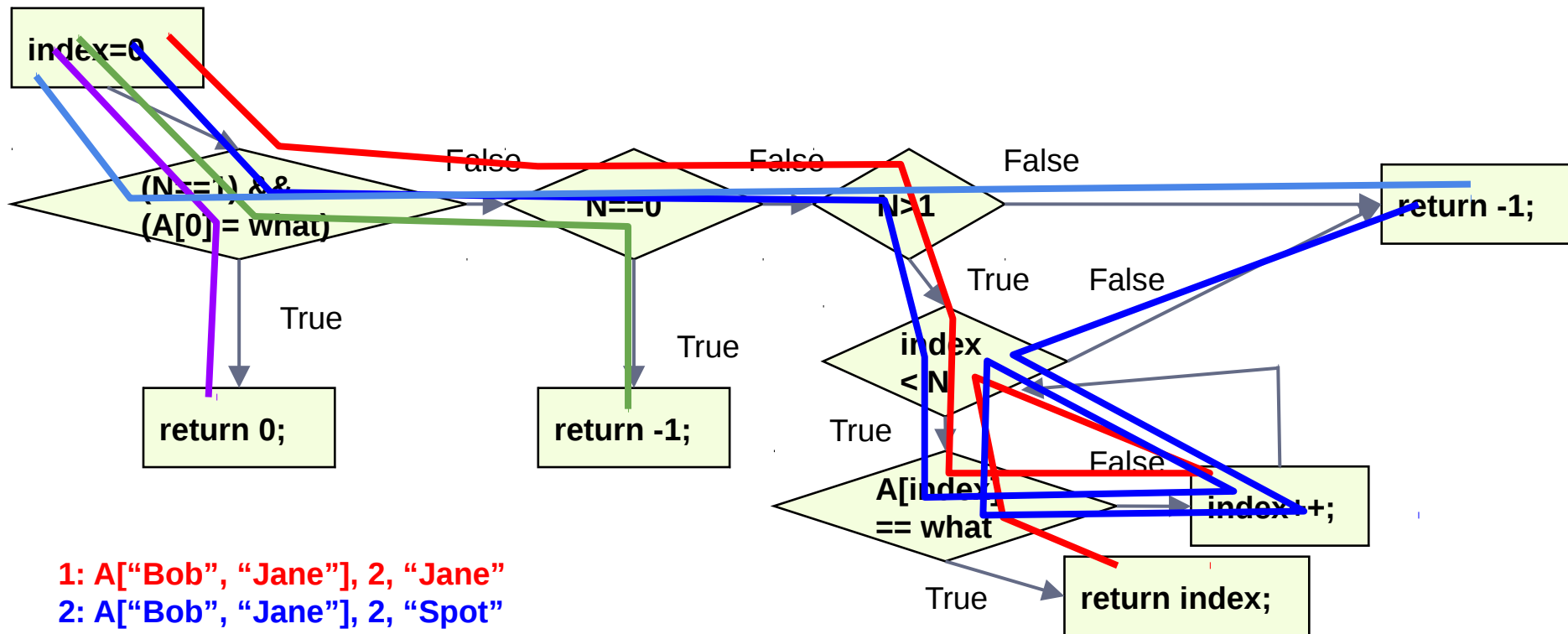
Activity - Possible Solution

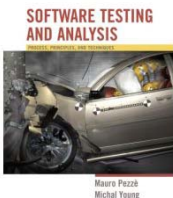■ Write tests that provide statement, branch, and basic condition coverage over the following code:

Activity - Possible Solution

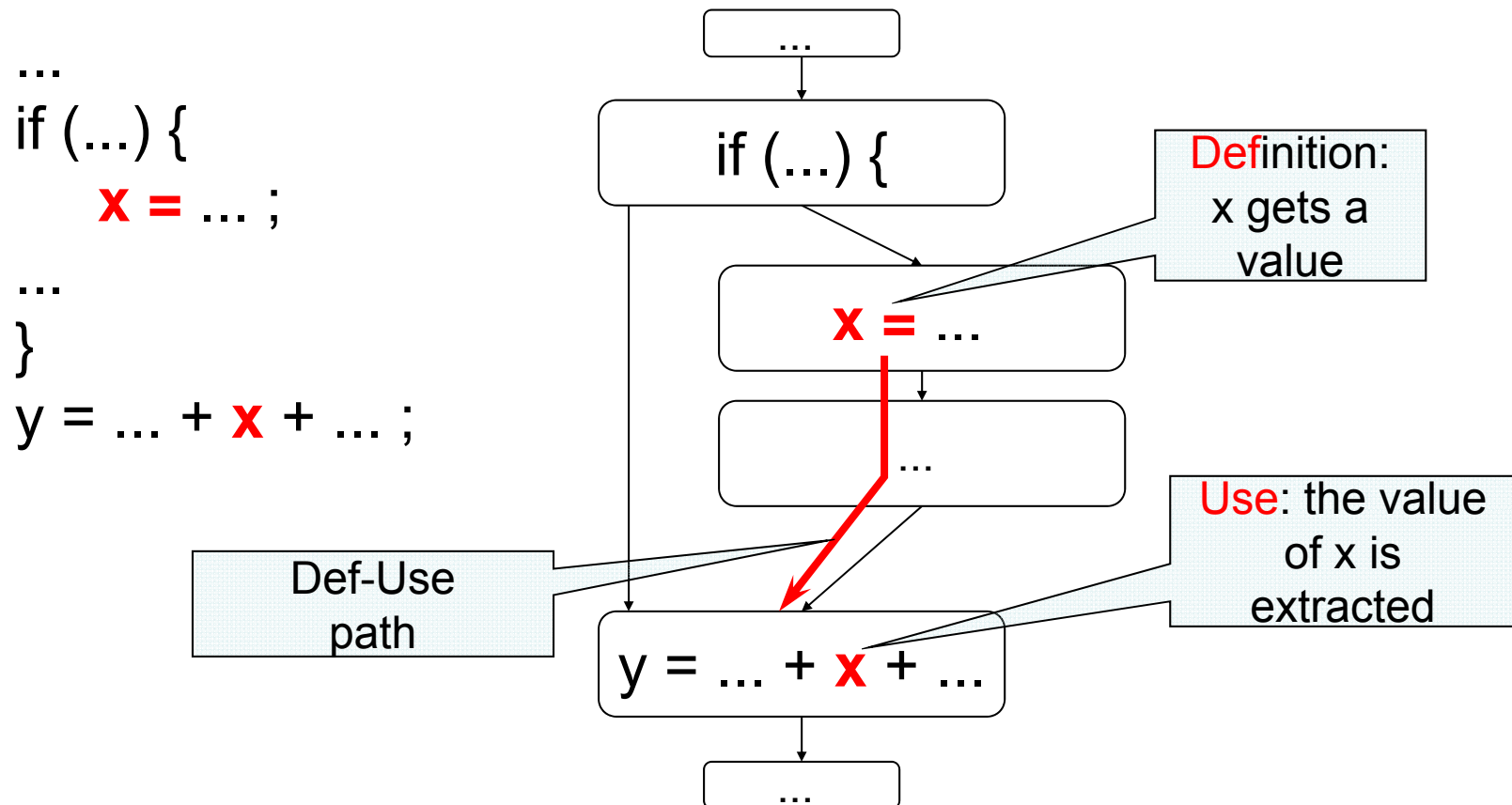■ Write tests that provide statement, branch, and basic condition coverage over the following code:

index=0

(N==1) &&
(A[0] = what)

N==0

N>1

False   False   False

return -1;

True   True

return 0;   return -1;   index
                               < N

True

A[index]
== what

index++;

True

return index;

False

False

1: A["Bob", "Jane"], 2, "Jane"
2: A["Bob", "Jane"], 2, "Spot"
3: A[], 0, "Bob"
4. A["Bob"], 1, "Bob"
5. A["Bob"], 1, "Spot"

Dependence and Data Flow Models

Def-Use Pairs


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

...

if (...) {

Definition:
x gets a
value

x = ...

...

Def-Use
path

y = ... + x + ...

Use: the value
of x is
extracted

...

SOFTWARE TESTING
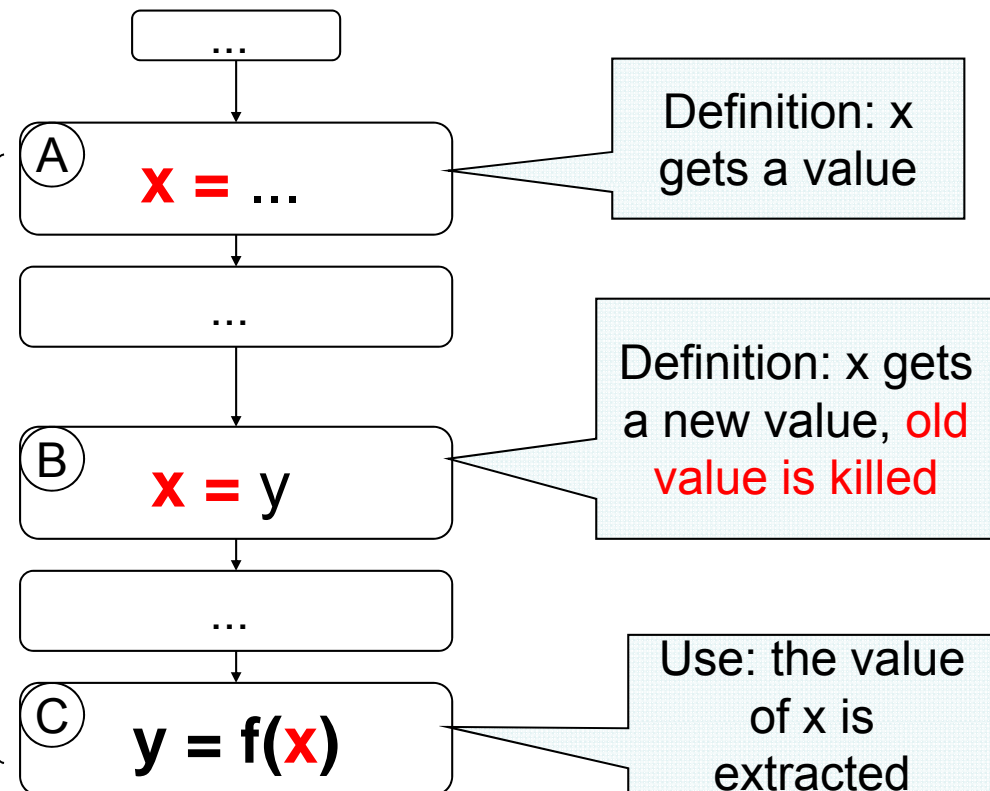AND ANALYSIS

Mauro Pezzè
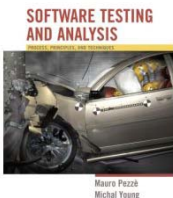Michal Young

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

...

(A) **X = ...** → Definition: x gets a value

...

(B) **X = y** → Definition: x gets a new value, old value is killed

...

(C) **y = f(x)** → Use: the value of x is extracted

SOFTWARE TESTING
AND ANALYSIS

Mauro Pezzè
Michal Young

Data flow testing

Terms

- DU pair: a pair of *definition* and *use* for some variable, such that at least one DU path exists from the definition to the use

  x = … 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
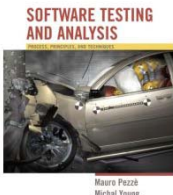
Adequacy criteria

- 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
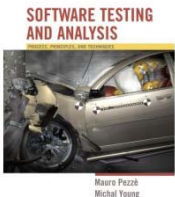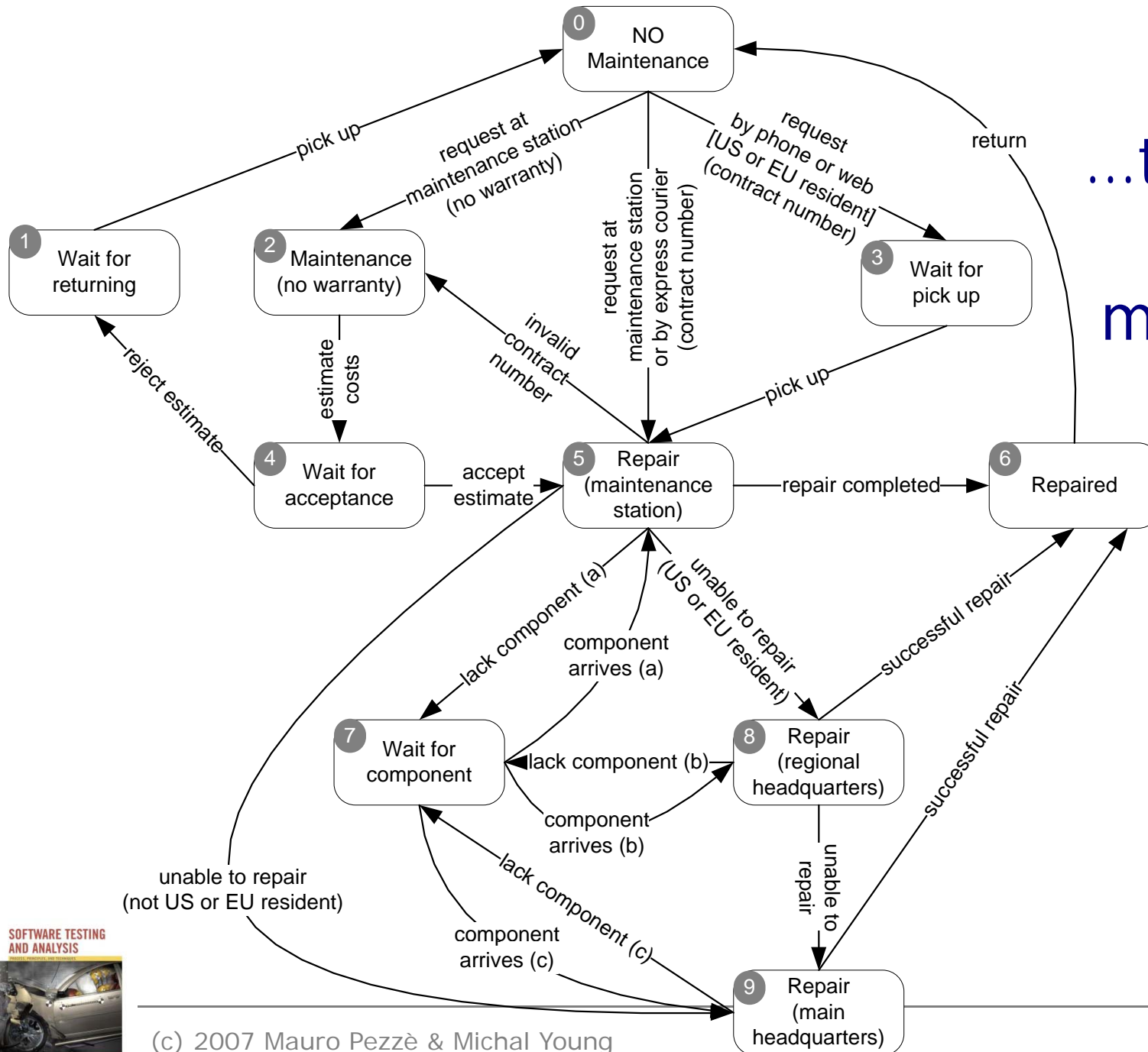
  - (Every computed value is used somewhere)

Corresponding coverage fractions can also be defined
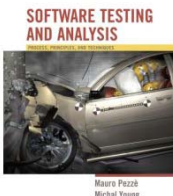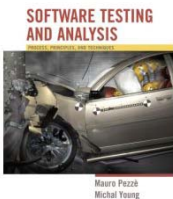
Model based testing
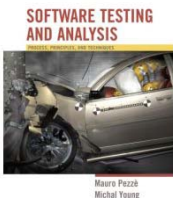
...to a finite state machine...

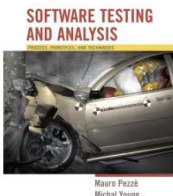# Testing Object Oriented Software

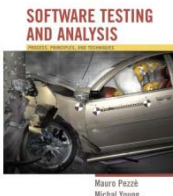## Chapter 15

Characteristics of OO Software

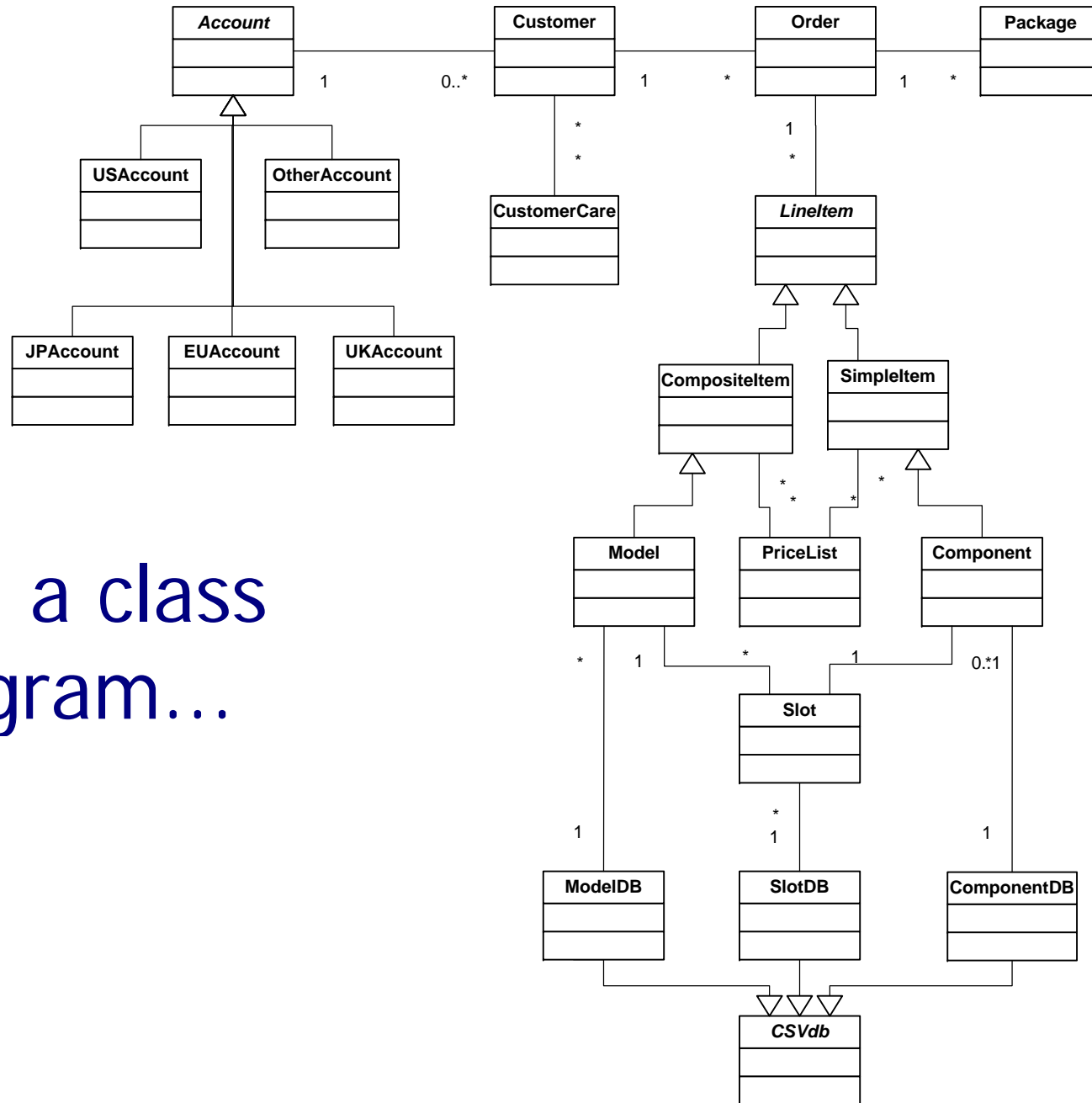Typical OO software characteristics that impact testing

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

Interclass Testing

- The first level of *integration testing* for object-oriented software

  - Focus on interactions between classes

- Bottom-up integration according to "depends" relation

  - A depends on B: Build and test B, then A

- Start from use/include hierarchy

    - Implementation-level parallel to logical "depends" relation
  - Class A makes method calls on class B
  - Class A objects include references to class B methods
    - but only if reference means "is part of"

from a class diagram…

....to a hierarchy

Customer

Order

Package

USAccount

OtherAccount

CustomerCare

PriceList

Component

JPAccount
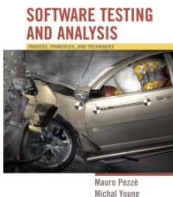
EUAccount

UKAccount

Model

ComponentDB

Slot
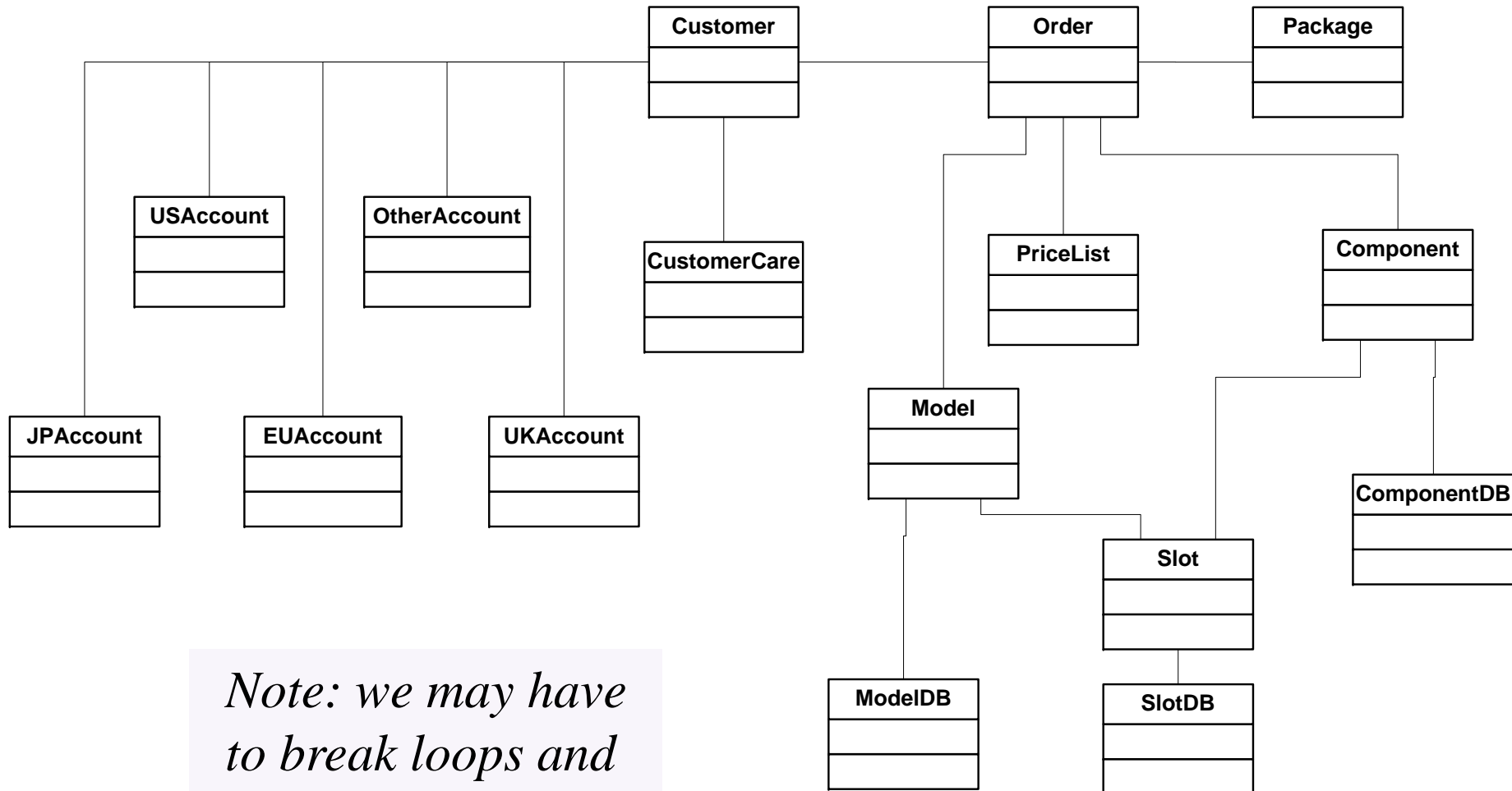
Note: we may have to break loops and generate stubs

ModelDB

SlotDB
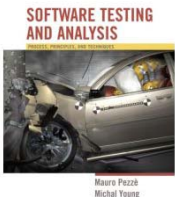
Intraclass data flow testing

- 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

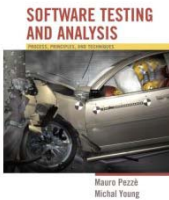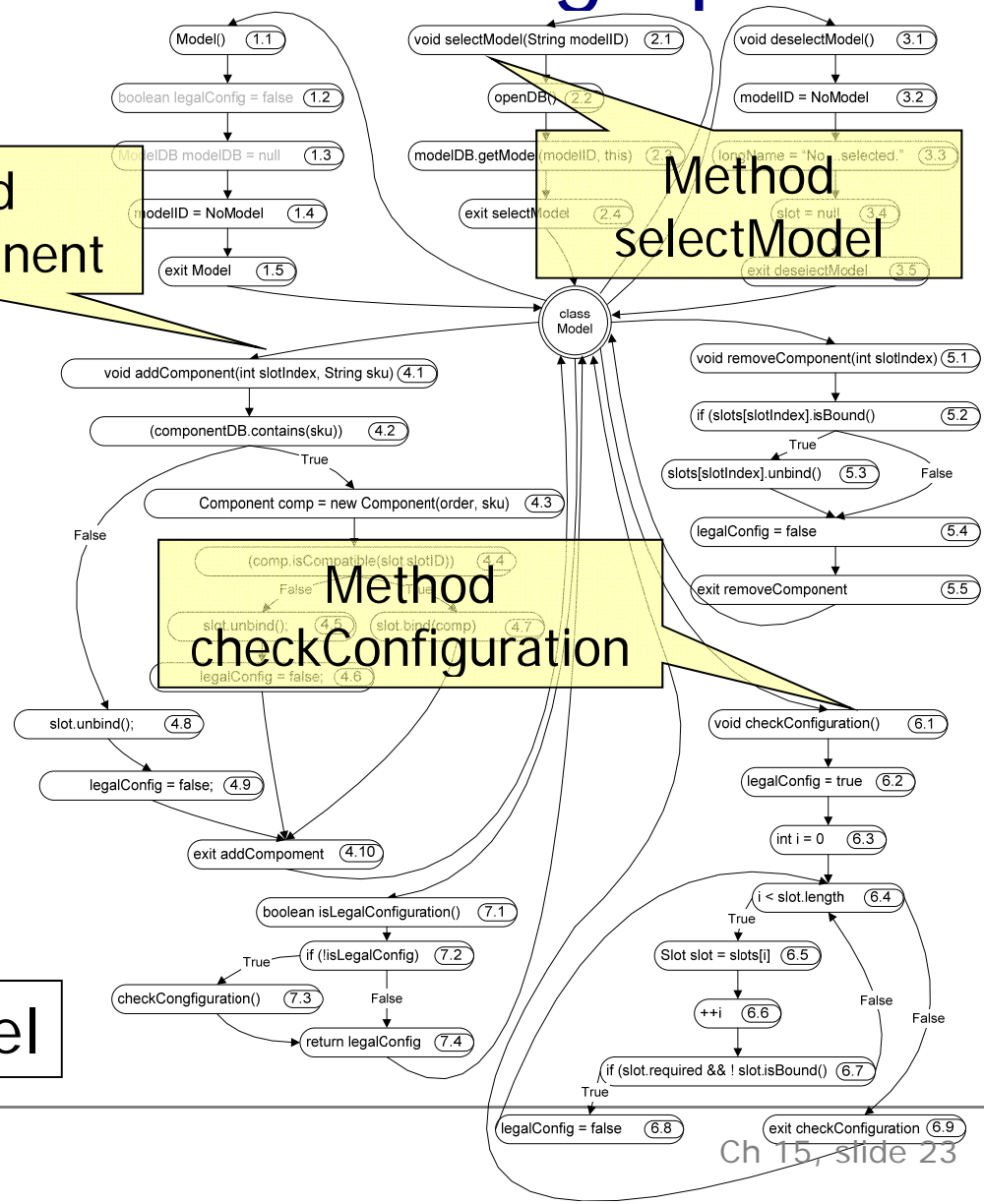   from the end nodes of the methods to node *class*

=> control flow through *sequences* of method calls

class Model

Model() 1.1
boolean legalConfig = false 1.2
ModelDB modelDB = null 1.3
modelID = NoModel 1.4
exit Model 1.5

void selectModel(String modelID) 2.1
openDB() 2.2
modelDB.getMode (modelID, this) 2.3
exit selectModel 2.4

void deselectModel() 3.1
modelID = NoModel 3.2
longName = "None selected." 3.3
slot = null 3.4
exit deselectModel 3.5

Method selectModel

Method addComponent

void addComponent(int slotIndex, String sku) 4.1
(componentDB.contains(sku)) 4.2
Component comp = new Component(order, sku) 4.3
(comp.isCompatible(slot slotID)) 4.4
slot.unbind(); 4.5
slot.bind(comp) 4.7
legalConfig = false; 4.6
slot.unbind(); 4.8
legalConfig = false; 4.9
exit addCompoment 4.10

void removeComponent(int slotIndex) 5.1
if (slots[slotIndex].isBound() 5.2
slots[slotIndex].unbind() 5.3
legalConfig = false 5.4
exit removeComponent 5.5

Method checkConfiguration

void checkConfiguration() 6.1
legalConfig = true 6.2
int i = 0 6.3
i < slot.length 6.4
Slot slot = slots[i] 6.5
++i 6.6
if (slot.required && ! slot.isBound() 6.7
legalConfig = false 6.8
exit checkConfiguration 6.9

boolean isLegalConfiguration() 7.1
if (!isLegalConfig) 7.2
checkCongfiguration() 7.3
return legalConfig 7.4

class Model

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

© Lionel Briand 2010

9

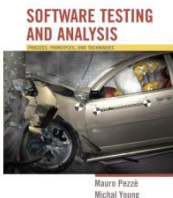# Regression Testing

Ajitha Rajan

Example
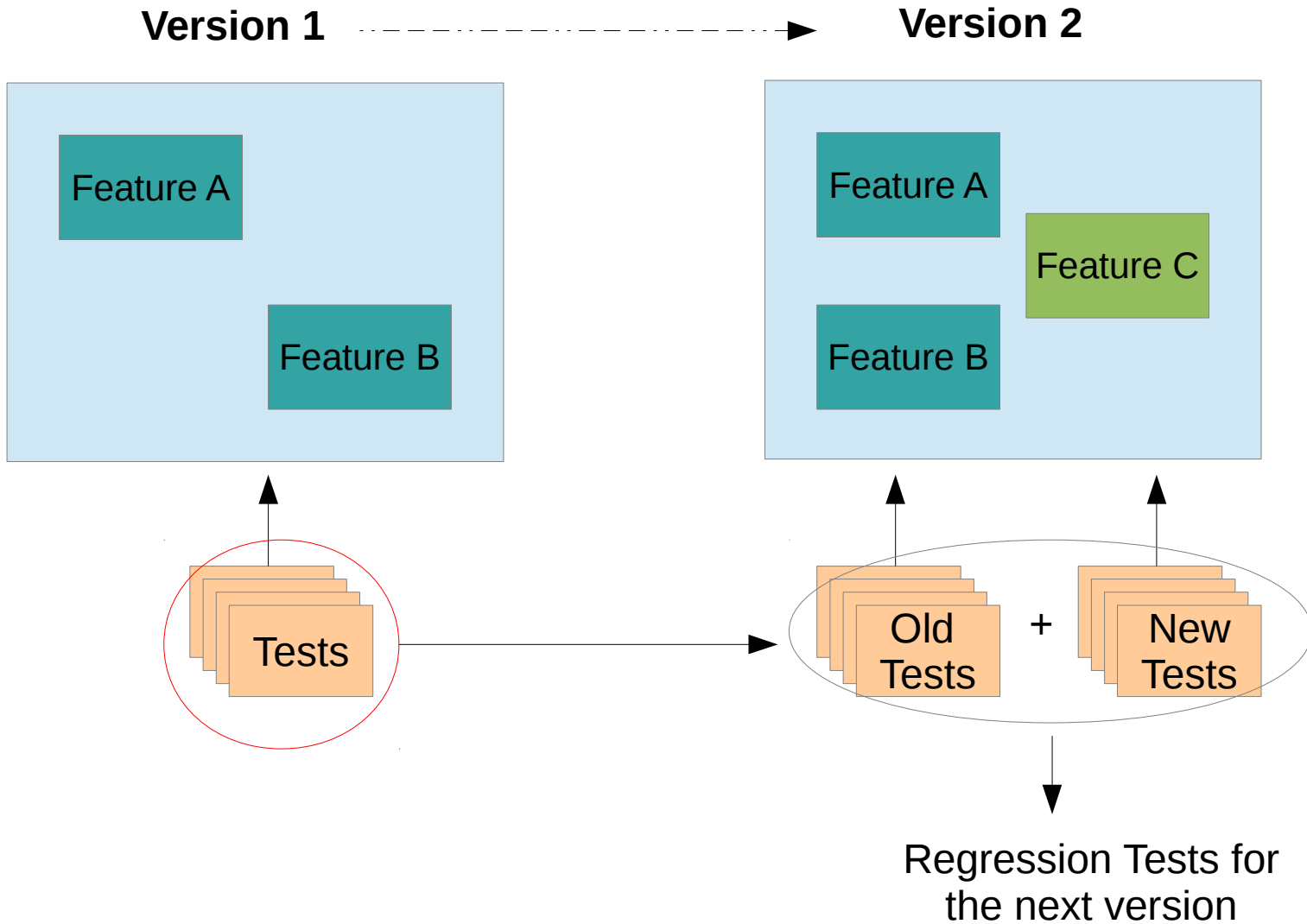
Regression Test Optimization

➜ Re-test All

➜ Regression Test Selection

➜ Regression Test Set Minimisation

➜ Regression Test Set Prioritisation

Integration and Component-based Software Testing

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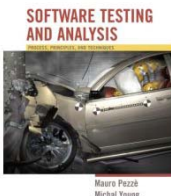
Top down ..
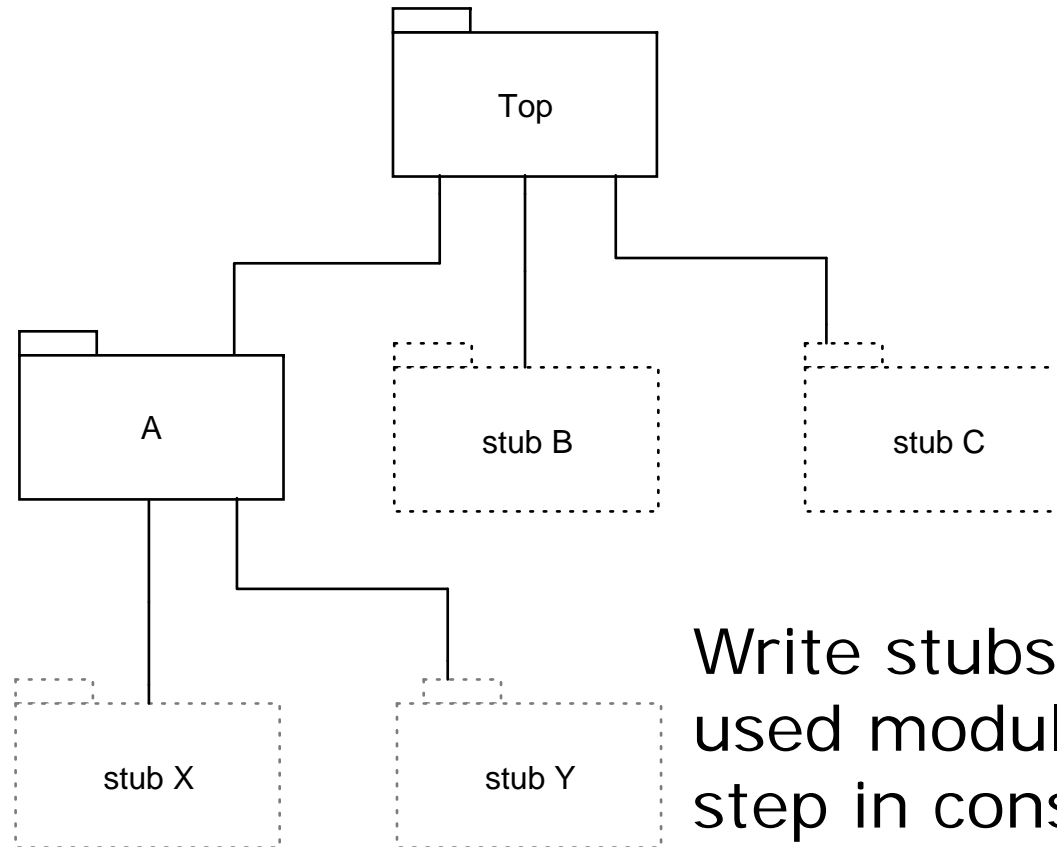
Write stubs of called or used modules at each step in construction

Bottom Up ..

Driver

Driver

X

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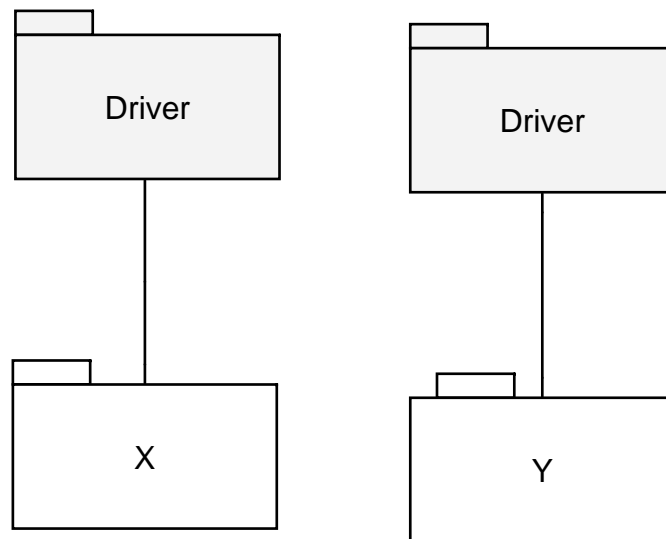
… but we must construct drivers for each module (as in unit testing) …

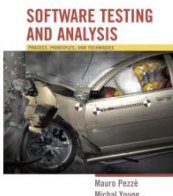# System, Acceptance, and Regression Testing
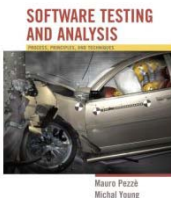
System Testing

- Key characteristics:
  - Comprehensive (the whole system, the whole spec)
  - Based on specification of observable behavior

    Verification against a requirements specification, not validation, and not opinions

  - Independent of design and implementation

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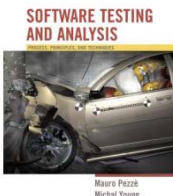
Global Properties

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

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

Context-Dependent Properties

- Beyond system-global: Some properties depend on the system context and use

  - Example: Performance properties depend on environment and configuration

  - Example: Privacy depends both on system and how it is used

    - Medical records system must protect against unauthorized use, and authorization must be provided only as needed

  - Example: Security depends on threat profiles

    - And threats change!

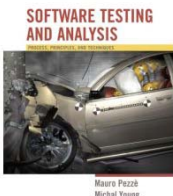- Testing is just one part of the approach
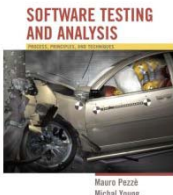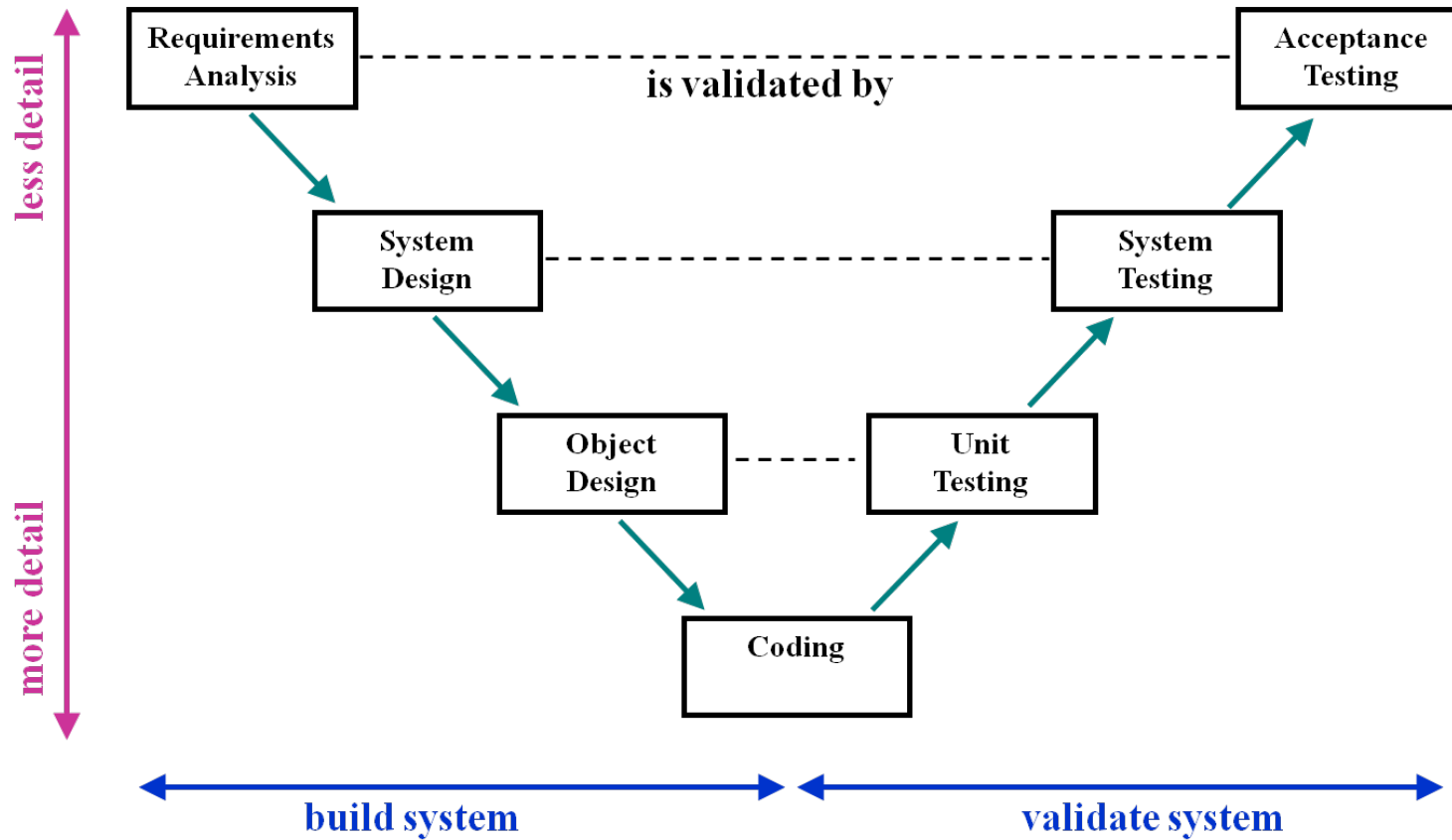
22.3

Acceptance testing
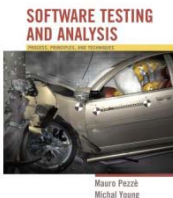
Estimating Dependability

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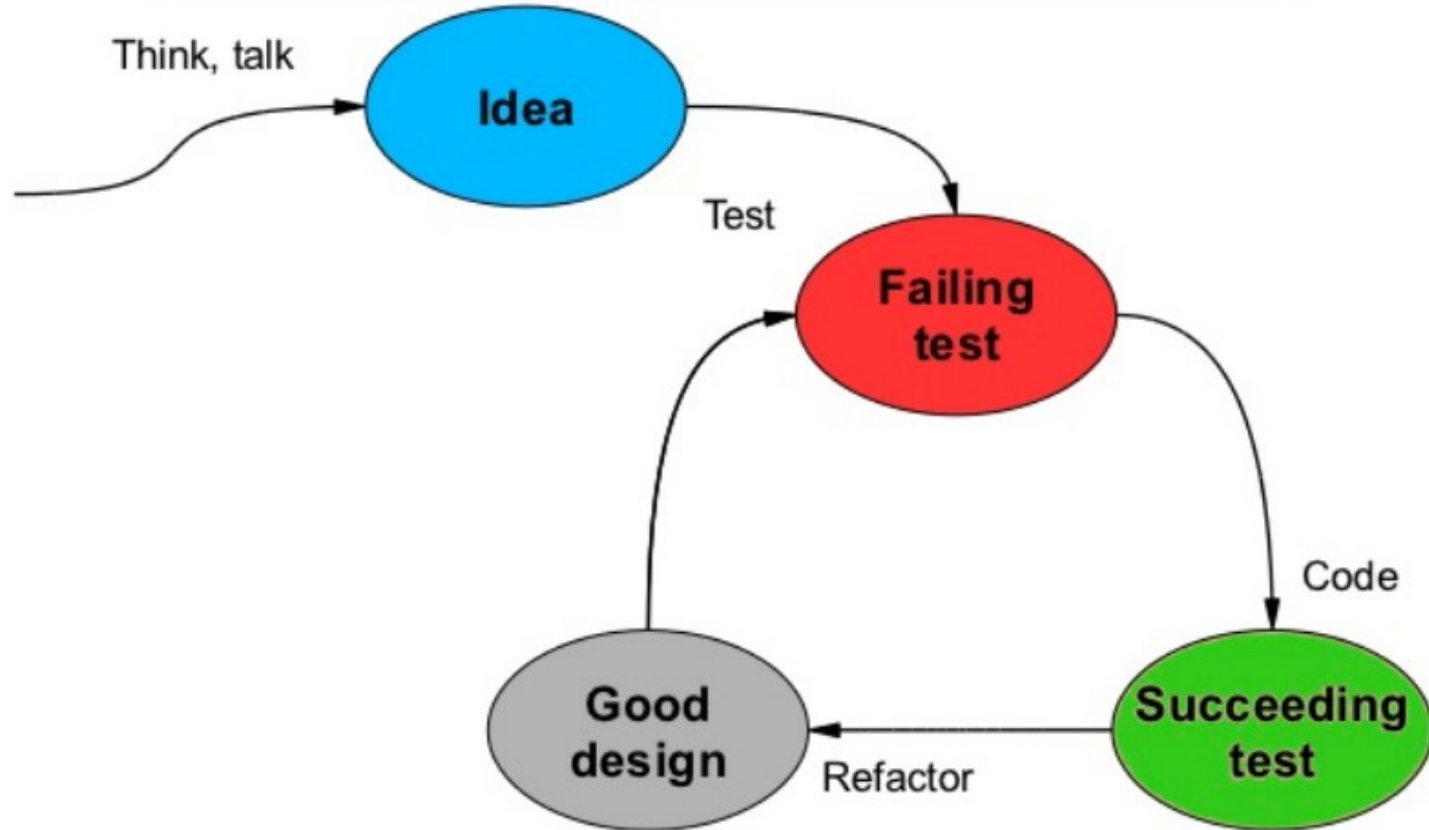
V-model

School of
informatics

eXtreme Programming (XP)

http://www.extremeprogramming.org/map/project.html

HOW DOES TDD HELP

TDD CYCLE

- ## Write Test Code
  - Guarantees that every functional code is testable
  - Provides a specification for the functional code
  - Helps to think about design
  - Ensure the functional code is tangible
- ## Write Functional Code
  - Fulfill the requirement (test code)
  - Write the simplest solution that works
  - Leave Improvements for a later step
  - The code written is only designed to pass the test
    - no further (and therefore untested code is not created).
- ## Refactor
  - Clean-up the code (test and functional)
  - Make sure the code expresses intent
  - Remove code smells
  - Re-think the design
  - Delete unnecessary code

Principle of TDD (In Practice)

Start

Write a Test

See it fail because there's no dev code → Run the Test

Write (just enough) Dev Code to compile

See it fail because no logic is implemented → Run the Test

Red

Green

Refactor

Write (just enough) Dev Code to pass

See the test pass → Run the Test

Refactoring

TDD