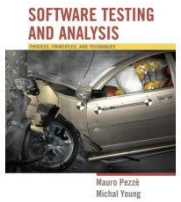
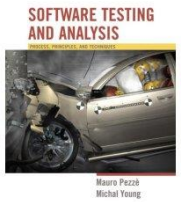
Data Flow Analysis

Computing data flow information



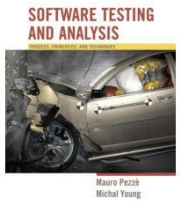
Remember from the last lecture

- Control flow models
- Def-use pairs
- Data dependence



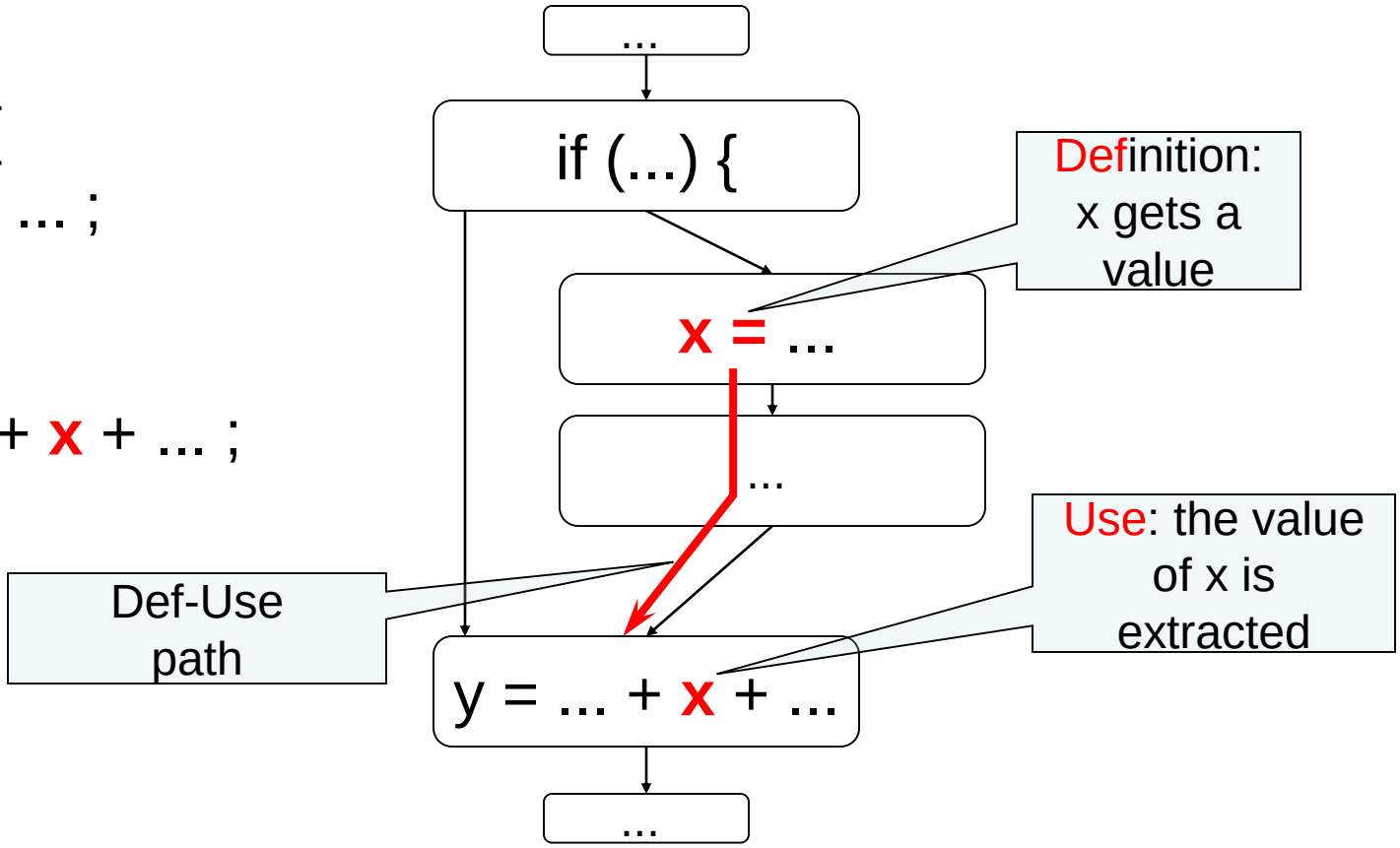
Def-Use Pairs (1)

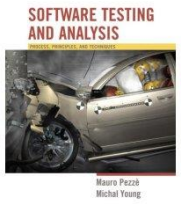
- A **def-use (du) pair** associates a point in a program where a value is produced with a point where it is used
- **Definition**: where a variable gets a value
  - Variable declaration (often the special value "uninitialized")
  - Variable initialization
  - Assignment
  - Values received by a parameter
- **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
   }
```

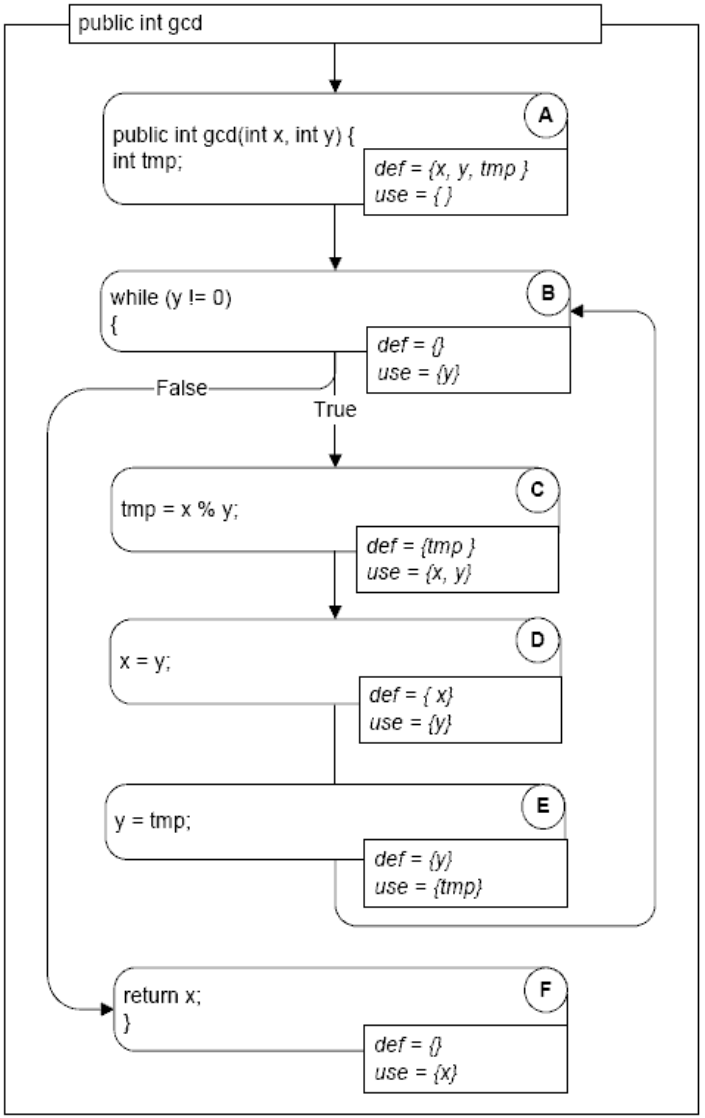
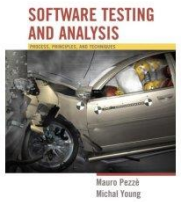
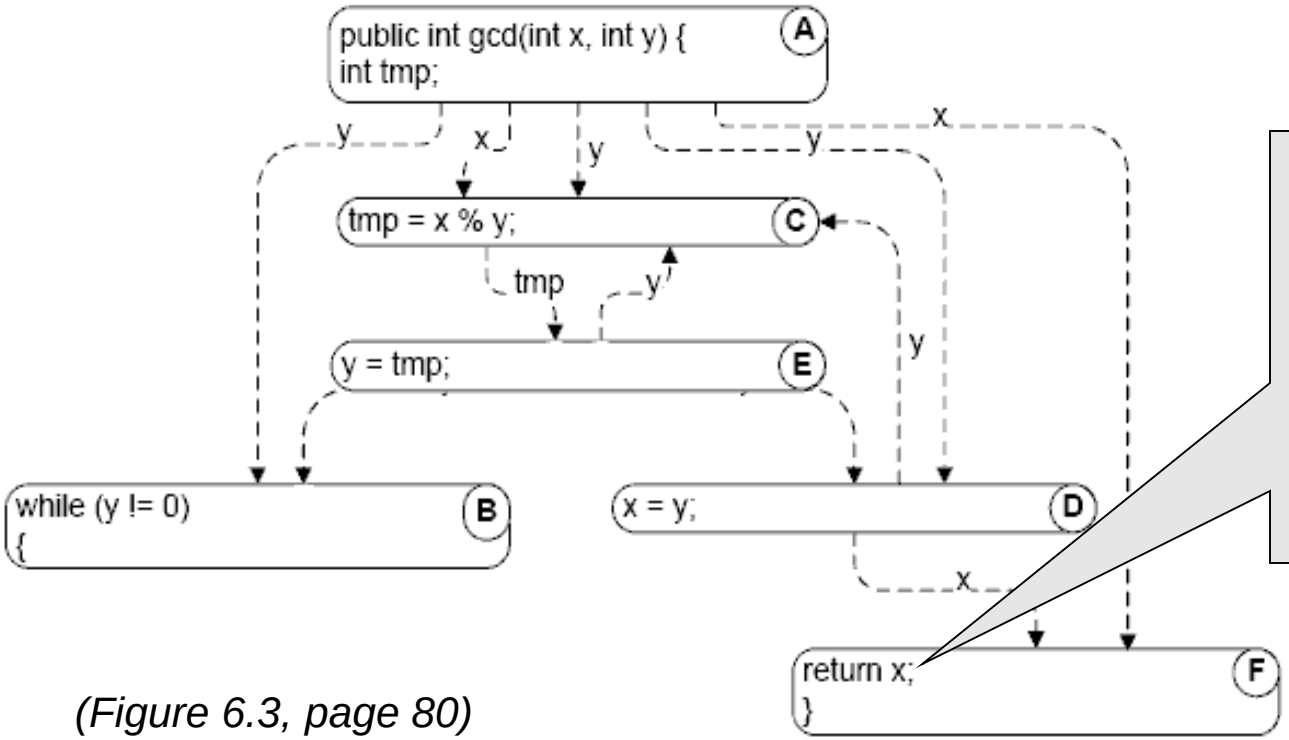


Figure 6.2, page 79

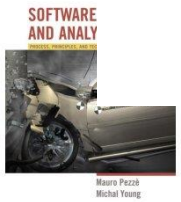


(Direct) Data Dependence Graph

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

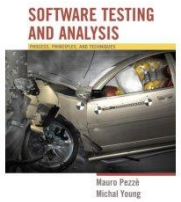


(Figure 6.3, page 80)



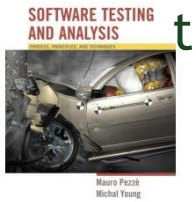
Data Flow Analysis

Computing data flow information

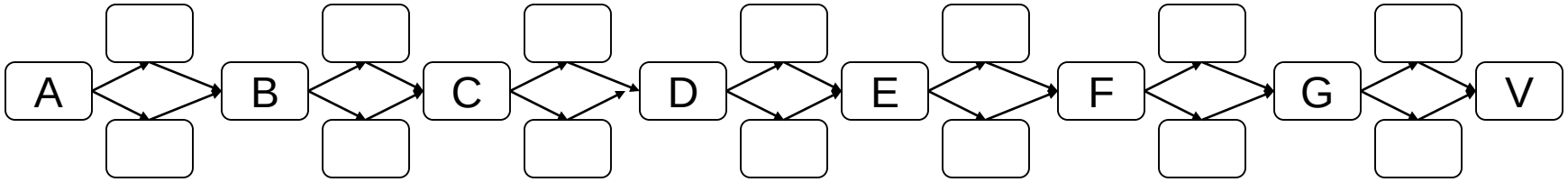


Calculating def-use pairs

- 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

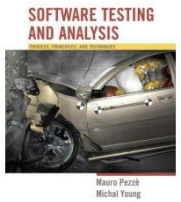
8 from A to D

16 from A to E

...

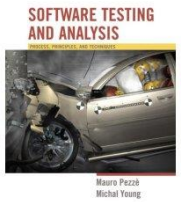
128 paths from A to V

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_d of variable v reaches a predecessor node p, and if v is not redefined at that node (in which case we say the v_d is killed at that point), then the definition is propagated on from p to n.



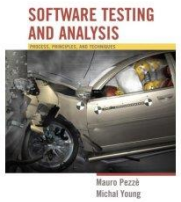
Equations of node E (y = tmp)

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
   }
```

$$Reach(E) = ReachOut(D)$$

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

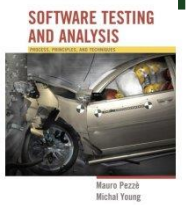


Equations of node B (while (y != 0))

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
   }
```

- Reach(B) = ReachOut(A) $\cup$ ReachOut(E)
- ReachOut(A) = gen(A) = $\{x_A, y_A, tmp_A\}$
- ReachOut(E) = (Reach(E) \ $\{y_A\}$) $\cup$ $\{y_E\}$



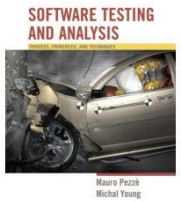
General equations for Reach analysis

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

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

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

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



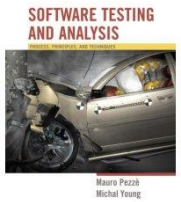
Avail equations

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

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

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

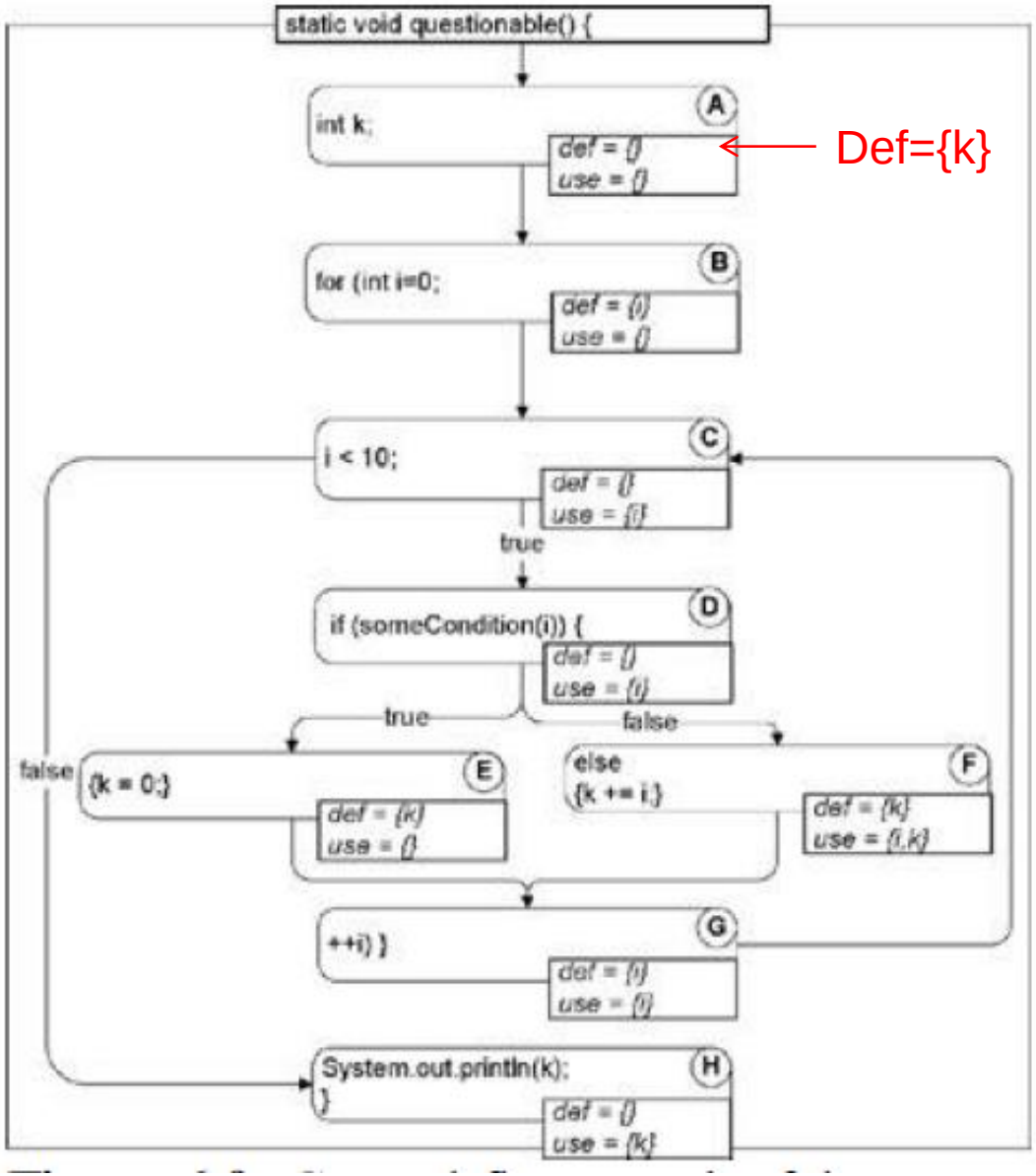
$$\text{kill}(n) = \{ \text{exp} \mid \text{exp has variables assigned at n} \}$$

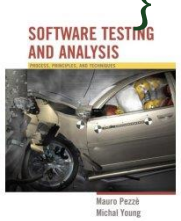


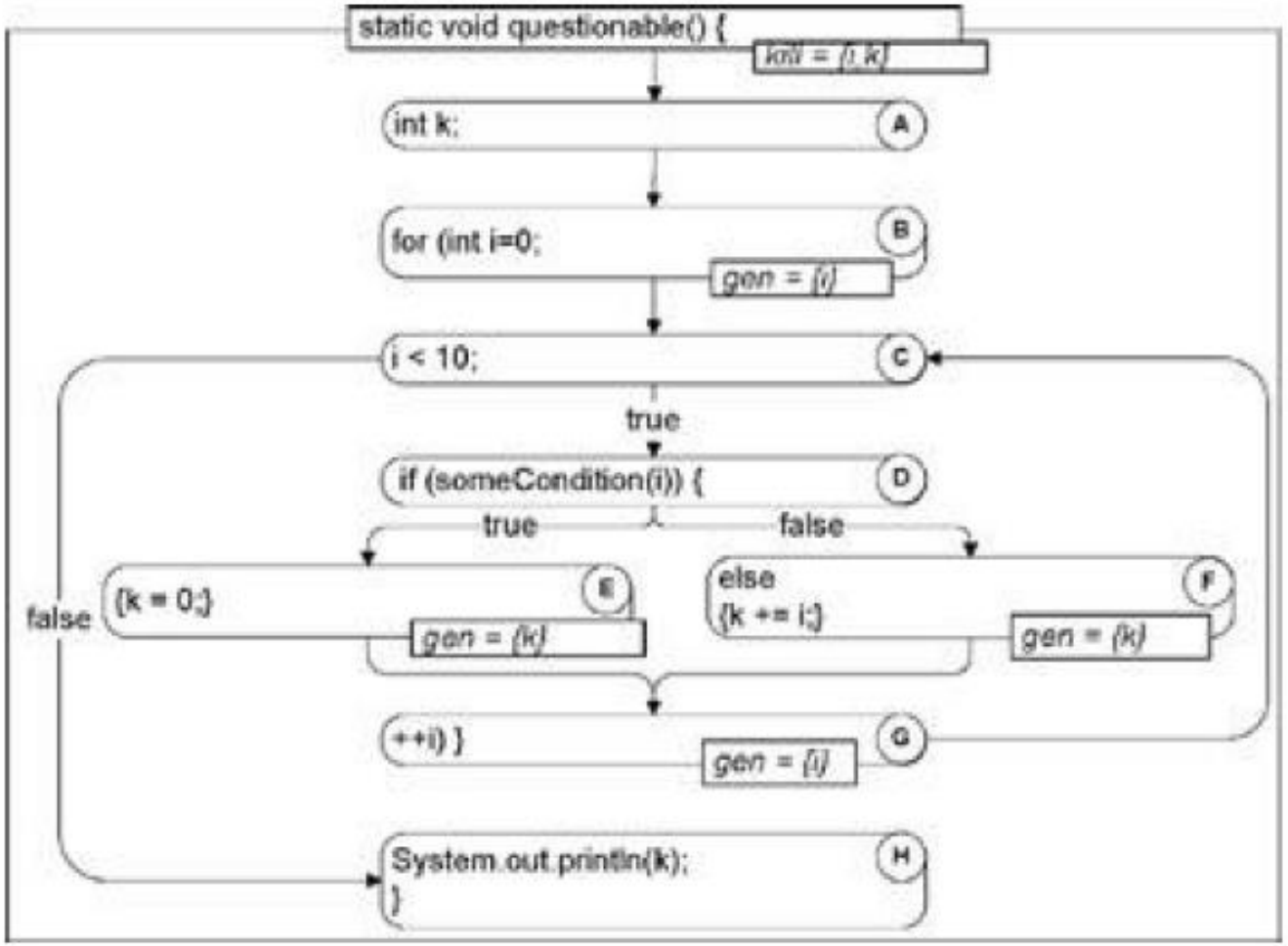
Example

```
static void questionable(){
    int k;
    for(int i=0; i < 10; ++i){
        if(someCondition(i)){
            k=0;
        }
        else{
            k+=i;
        }
    }
    System.out.println(k);
}
```

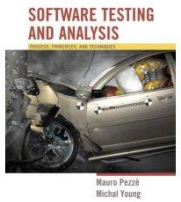
Avail set from G to C?
Avail set from B to C?
Avail(F)?







- Avail set from G to C: {i,k}
- Avail set from B to C: {i}
- All-paths analysis → Avail(C):{i}
- This value propagates through nodes C and D to node F.
- Since k is not a member of Avail(F) → uninitialized variable problem



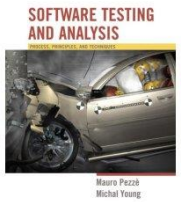
Live variable equations

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

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

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

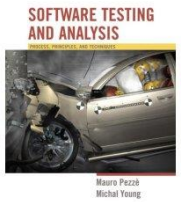
$$kill(n) = \{\ v \mid v \text{ is modified at } n\ \}$$



Classification of analyses

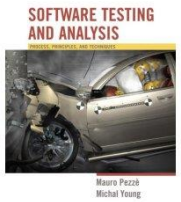
- Forward/backward: a node's set depends on that of its predecessors/successors
- Any-path/all-path: a node's set contains a value iff it is coming from any/all of its inputs

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



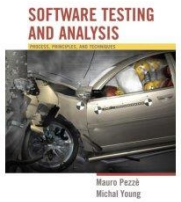
Iterative Solution of Dataflow Equations

- 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)



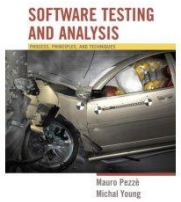
Cooking your own: From Execution to Conservative Flow Analysis

- 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



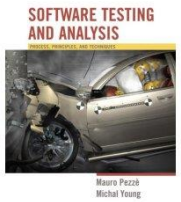
Data flow analysis with arrays and pointers

- Arrays and pointers introduce uncertainty
- In the example: a[i]=13; k=a[j];
  - Are two lines a def-use pair?
- In the example: a[2]=42; i=b[2];
  - Is there a def-use pair?
- The uncertainty is accomodated 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

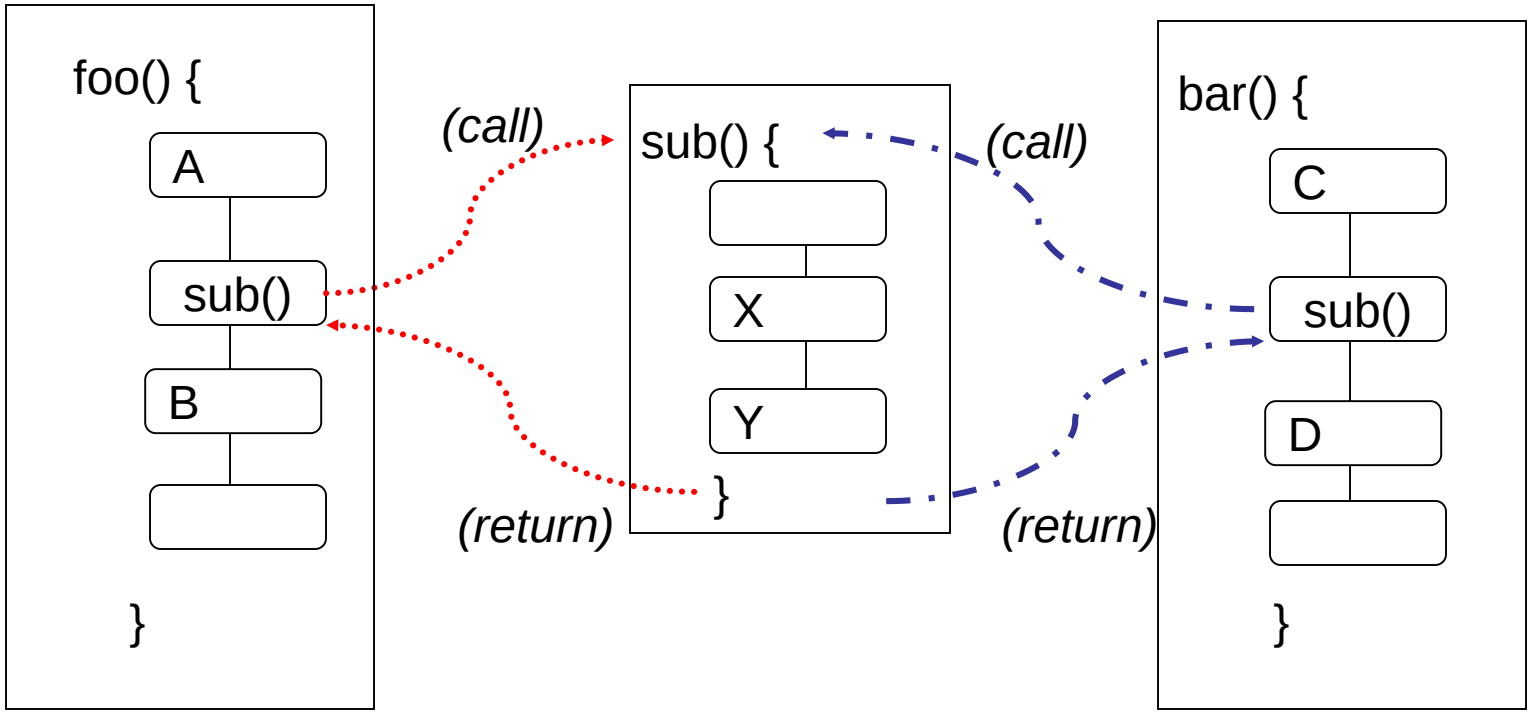


Example

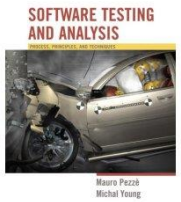
```
1 public void transfer (CustInfo fromCust, CustInfo
toCust) {
2
3 PhoneNum fromHome = fromCust.gethomePhone();
4 PhoneNum fromWork = fromCust.getworkPhone();
5
6 PhoneNum toHome = toCust.gethomePhone();
7 PhoneNum toWork = toCust.getworkPhone();
```



Context Sensitivity

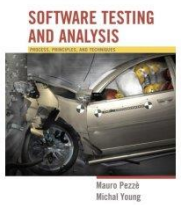


A **context-sensitive** (interprocedural) analysis distinguishes sub() called from foo() from sub() called from bar();
A **context-insensitive** (interprocedural) analysis does not separate them, as if foo() could call sub() and sub() could then return to bar()

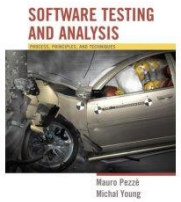


Summary

- 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

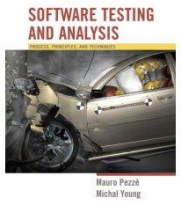


Functional testing



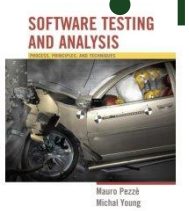
Functional testing

- 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

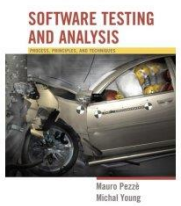


Why Not Random?

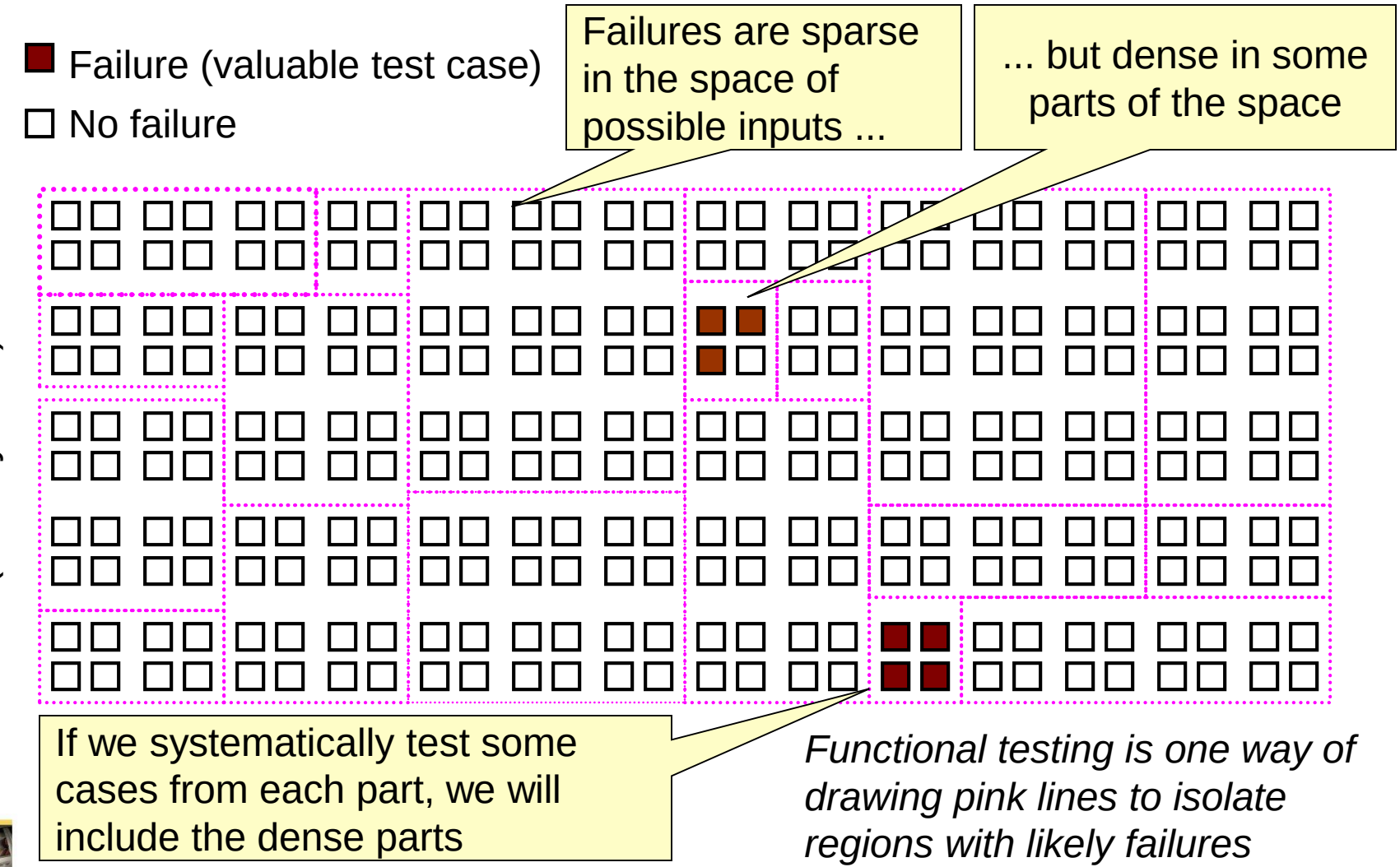
- Non-uniform distribution of faults
- *Example:* Java class "roots" applies quadratic equation $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

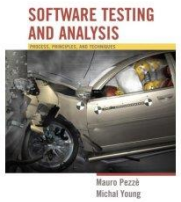
  Incomplete implementation logic: Program does not properly handle the case in which $b^2 - 4ac = 0$ and $a=0$

  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



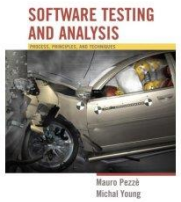
Systematic Partition Testing





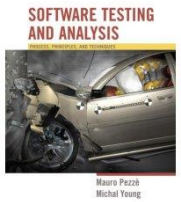
The partition principle

- 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



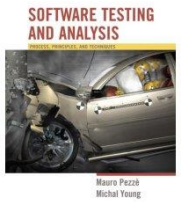
Why functional testing?

- 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



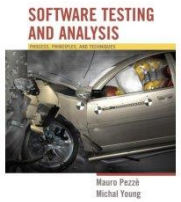
Early functional test design

- 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



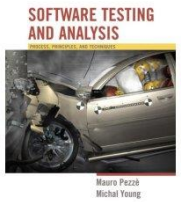
Functional versus Structural: Classes of faults

- 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!



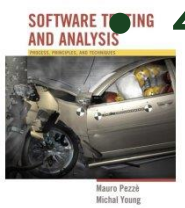
Functional vs structural test: granularity levels

- 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

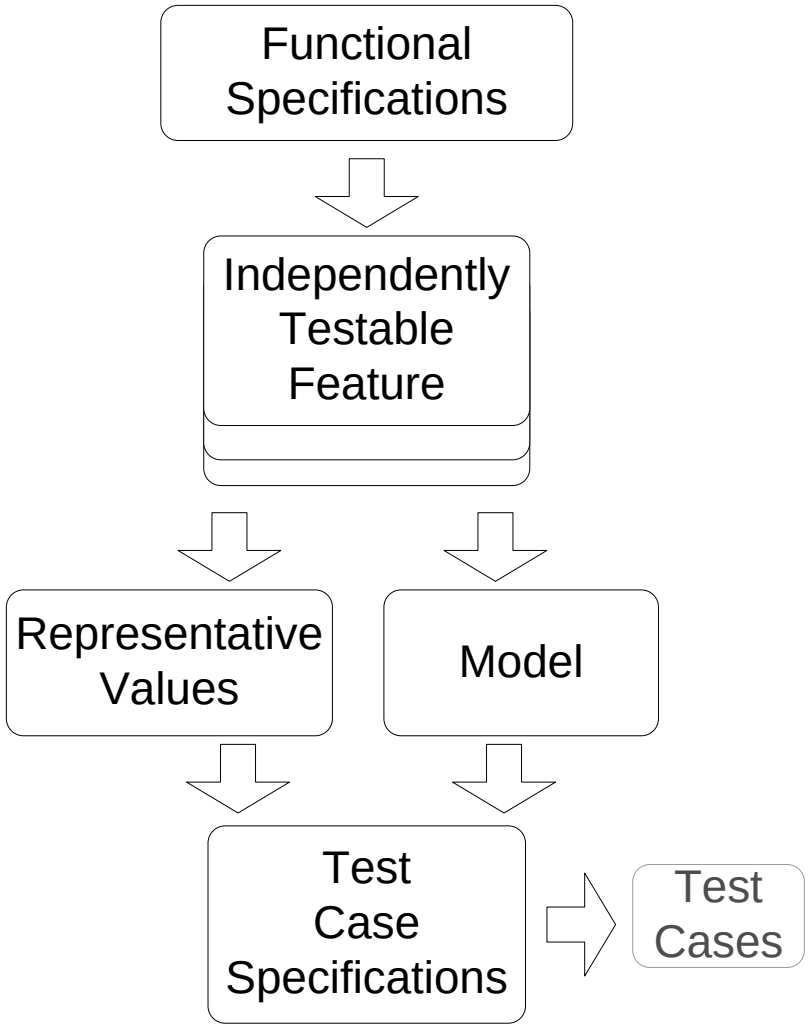


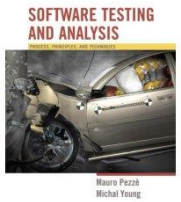
Steps: From specification to test cases

- 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



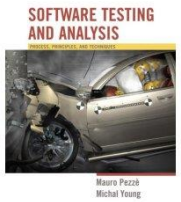
From specification to test cases





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?

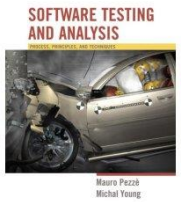


Example: Representative values

Simple example with one input, one output

- 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