Introduction to Software Testing
Chapter 6
Input Space Partition Testing

Paul Ammann & Jeff Offutt

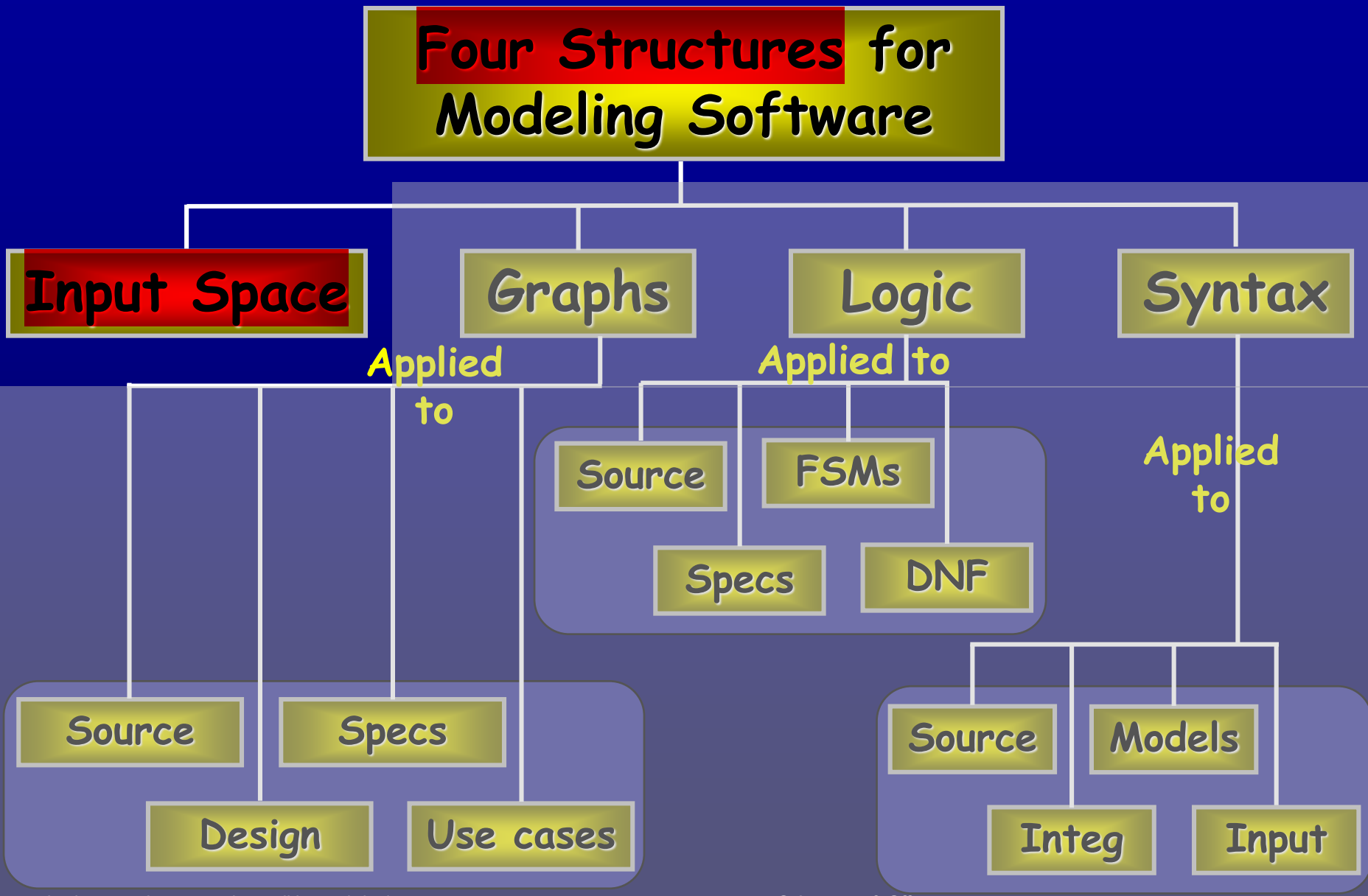
http://www.cs.gmu.edu/~offutt/softwaretest/

Engineers take ideas invented by quick thinkers

and

build products for slow thinkers.

Ch. 6 : Input Space Coverage



Input Space Partitioning

- Takes the view that we can directly divide the input space according to logical partitioning of the inputs.
- Is independent of the RIPR model–we only use the input space of the software under test.

Benefits of ISP

- Can be equally applied at several levels of testing
  - Unit
  - Integration
  - System

- Relatively easy to apply with no automation
- Easy to adjust the procedure to get more or fewer tests
- No implementation knowledge is needed
  - Just the input space

Input Domains

تمام مقادیر ممکن ورودی یک برنامه میشه اینپوت دامین

- The input domain for a program contains all the possible inputs to that program
- For even small programs, the input domain is so large that it might as well be infinite
- Testing is fundamentally about choosing finite sets of values from the input domain
- *Input parameters* define the scope of the input domain
  - Parameters to a method
  - Data read from a file
  - Global variables
  - User level inputs
- Input domains are partitioned into regions (blocks)
- At least one value is chosen from each block

Partitioning Domains

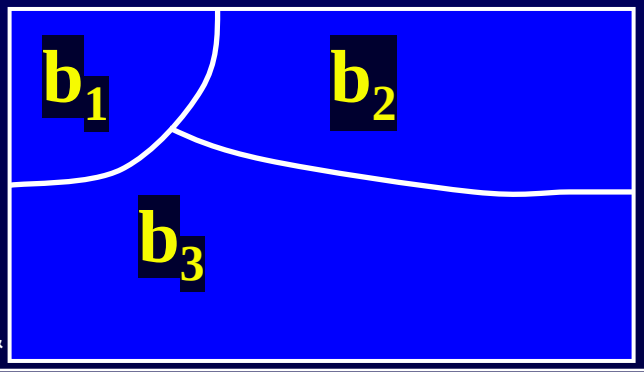
- *Domain D*
- *Partition scheme q of D*
- The partition *q* defines a *set of blocks*, $Bq = b_1, b_2, \ldots, b_Q$
- The partition must satisfy two properties :
  1. Blocks must be *pairwise disjoint* (no overlap)

$$b_i \cap b_j = \Phi, \forall\, i \neq j, b_i, b_j \in B_q$$

  2. [illegible] main *D* (complete)

2.Together the blocks cover the domain D (complete)

$$\bigcup_{b \in Bq} b = D$$



Using Partitions – Assumptions

- Choose a value from each block
- Each value is assumed to be equally useful for testing
- Application to testing
  - Find characteristics in the inputs : parameters, semantic descriptions, …
  - Partition each characteristic
  - Choose tests by combining values from characteristics
- Example Characteristics
  - Input X is null
  - Order of the input list F (sorted, inverse sorted, arbitrary, …)
  - Input device (DVD, CD, VCR, computer, …)

نبودن اشتراک دو به دو بین هیچ کدوم از بلاک ها باید بررسی شه

Choosing Partitions

- Choosing (or defining) partitions seems easy, but is easy to get wrong

لیست هایی که همه ی اعضاشون یکسان و مثل هم هستند

- Consider the characteristic "*order of elements in list F*"

b_1 = sorted in ascending order
b_2 = sorted in descending order
b_3 = arbitrary order

Design blocks for that characteristic

but … something's fishy …

What if the list is of length 1?

Can you find the problem?

The list will be in all three blocks
That is, disjointness is not satisfied

Solution:

Each characteristic should address just one property

Can you think of a solution?

C1:
- c1.b1 = true
- c1.b2 = false

C2: List F sorted descending
- c2.b1 = true
- c2.b2 = false

C1: List F sorted ascending

b1 = sorted in ascending order
b2 = sorted in descending order
b3 = arbitrary order

Properties of Partitions

- If the partitions are not complete or disjoint, that means the partitions have not been considered carefully enough
- They should be reviewed carefully, like any design
- Different alternatives should be considered
- We model the input domain in five steps …
  - Steps 1 and 2 move us from the implementation abstraction level to the design abstraction level (from chapter 2)
  - Steps 3 & 4 are entirely at the design abstraction level
  - Step 5 brings us back down to the implementation abstraction level

Modeling the Input Domain

اول باید مشخص کنیم چه فانکشن هایی رو میخاهیم تست کنیم؟

- Step 1 : Identify testable functions
  - Individual methods have one testable function
  - Methods in a class often have the same characteristics
  - Programs have more complicated characteristics—modeling documents such as UML can be used to design characteristics
  - Systems of integrated hardware and software components can use devices, operating systems, hardware platforms, browsers, etc.

باید همه ی پارامترهایی که رفتار یک فانکشن رو تحت تاثیر قرار میدن پیدا بشن

- S
a

Each usecase is associated with a specific intended functionality of the system, so it is very likely that the usecase designers have useful characteristics in mind that are relevant to developing test cases.

Step 2 : Find all the parameters that can affect the behavior of a given testable function.
Often fairly straightforward, even mechanical

  - Important to be complete
  - Methods : Parameters and state (non-local) variables used
  - Components : Parameters to methods and state variables
  - System : All inputs, including files and databases

مثلا ما میخاهیم یه نود رو به یک درخت اضافه کنیم
لی وضعیت ابتدایی درخت هم مهم است که در ابتدا نال
هست یانه؟ اگه نود داشته باشه از قبل باید پیمایش کنیم
مثلا تا برسیم به اخر درخت و نود جدید اضافه شه

Modeling the Input Domain (*cont*)

اینجا کاملا دیگه توی فاز abstractionاومدیم و یگه به پیاده سازی کاری نداریم

- Step 3 : Model the input domain
  - The domain is scoped by the parameters
  - The structure is defined in terms of characteristics
  - Each characteristic is partitioned into sets of blocks
  - Each block represents a set of values
  - This is the most creative design step in using ISP
- Step 4 : Apply a test criterion to choose combinations of values
  - A test input has a value for each parameter
  - One block for each characteristic
  - Choosing all combinations is usually infeasible
  - Coverage criteria allow subsets to be chosen
- Step 5 : Refine combinations of blocks into test inputs
  - Choose appropriate values from each block

Two Approaches to Input Domain Modeling

1. Interface-based approach
   - Develops characteristics directly from individual input parameters
   - Simplest application
   - Can be partially automated in some situations

2. Functionality-based approach
   - Develops characteristics from a behavioral view of the program under test
   - Harder to develop—requires more design effort
   - May result in better tests, or fewer tests that are as effective

***Input Domain Model* (IDM)**

1. Interface-Based Approach

- Mechanically consider each parameter in isolation
- This is an easy modeling technique and relies mostly on syntax

هرکدوم از پارامترها به صورت مستقل و مجزا بررسی میشه

- Easy to identify characteristics.
- Some domain and semantic information won't be used
  - Could lead to an incomplete IDM
  - Not all the information available to the test engineer will be reflected in the interface domain model.

- Ignores relationships among parameters
  - Important sub-combinations may be missed.

1. Interface-Based Example

- Consider method *triang()* from class *TriangleType* on the book website :
  - http://www.cs.gmu.edu/~offutt/softwaretest/java/Triangle.java
  - http://www.cs.gmu.edu/~offutt/softwaretest/java/TriangleType.java

```
public enum Triangle { Scalene, Isosceles, Equilateral, Invalid }
public static Triangle triang (int Side, int Side2, int Side3)
// Side1, Side2, and Side3 represent the lengths of the sides of a triangle
// Returns the appropriate enum value
```

در این روش هر پارامتری میشه یک characteristic
هر characteristicای هم براش پارتیشنینگ انجام میدیم:
بزرگتر از صفر، مساوی صفر، کمتر از صفر یعنی ارتباط هر پارامتر
ورودی رو با صفر بررسی میکنیم

The IDM for each parameter is identical

Reasonable characteristic : *Relation of side with zero*

2. Functionality-Based Approach

- Identify characteristics that correspond to the intended functionality
- Requires more design effort from tester
- Can incorporate domain and semantic knowledge
- Can use relationships among parameters
- Modeling can be based on requirements, not implementation
  - Can start early in development.

دانش زمینه و حوزه ی نرم افزار رو درنظر میگیره

بر مبنای رفتار قابل قبول برنامه میتونیم مدل کنیم

2. Functionality-Based Approach(Cnt'd)

- May be yields better test cases than the interface-based approach because the input domain models include more semantic information.
- Transferring more semantic information from the specification to the IDM makes it more likely to generate expected results for the test cases.
- The same parameter may appear in multiple characteristics, or characteristics do not map to single parameters of the software interface
  - so it's harder to translate values to test cases

رای طراحی تست کیس ها داریم از دانش بیشتری استفاده میکنیم برای همین کیفیت بهتری دارند نسبت به روش قبلی

مثل روش interface نیست که characteristicها به پارامترها مپ بشن بلکه به راحتی و به طور مستقیم مپ نمیشن
باید مشخص کنیم رفتار های برنامه از نظر تست کیس ها چه معنی ای دارند؟

2. Functionality-Based Example

- Again, consider method *triang()* from class *TriangleType* :

The three parameters represent a *triangle*

The IDM can combine all parameters

Reasonable characteristic : *Type of triangle*

نوع مثلث: مثلث معمولی، متساوی الساقین، متساوی الاضلاع،
نامعتبر بودن مثلث وارد شده

Steps 1 & 2—Identifying Functionalities, Parameters and Characteristics

- A creative engineering step
- More characteristics means more tests
- Interface-based : Translate parameters to characteristics
- Candidates for characteristics :
  - Preconditions and postconditions
  - Relationships among variables
  - Relationship of variables with special values (zero, null, blank, …)
- Should not use program source—characteristics should be based on the input domain
  - Program source should be used with graph or logic criteria
- Better to have more characteristics with few blocks
  - Fewer mistakes

هنگام خارج شدن از تابع باید یه شرطی برقرار باشه مثلامطمین هستیم که ارایه مون مرتب شده است وقتی از تابع میایم بیرون

precondtiion:
شرط هایی که باید درست باشند تا یه تابع مثلا کال شه مثلا x حتما مثبت باشه تا یه تابع کال شه

characteristicها باید برمبنای کل فضای ورودی باشند نه سورس برنامه

هرچی تعداد کاراکتریستیک هامون بیشتر باشه بهتره چون داریم اون تابع رو از ابعاد بیشتری بررسی میکنیم
چون میخایم تعداد تست هامون خیلی نشه پارتیشنینگ رو طوری انجام میدیم که تعداد بلاک هامون کم باشه

بهتره پارتیشنینگ
برمبنای دانش برنامه و
ون نیازمندی های برنامه
باشه و نه براساس
سورس برنامه

The tester should apply input space partitioning by using *domain knowledge* about the problem, not the implementation.

However, in practice, the code may be all that is available.

Overall, the more semantic information the test engineer can incorporate into characteristics, the better the resulting test set is likely to be.

Steps 1 & 2—Interface & Functionality-Based

```
public boolean findElement (List list, Object element)
// Effects: if list or element is null throw NullPointerException
//          else return true if element is in the list, false otherwise
```

Interface-Based Approach

Two parameters : list, element

Characteristics :

- list is null (block1 = true, block2 = false)
- list is empty (block1 = true, block2 = false)

مجموعه ی ورودی هایی که لیستشون نال است میشه بلاک ۱

ول لیست پارامترهامون رو مشخص میکنیم
توی این روش هر پارامتر میشه یه کرکتریستیک

Functionality-Based Approach

Two parameters : list, element

Characteristics :

- number of occurrences of element in list
  - (0, 1, >1)
- element occurs first in list
  - (true, false)
- element occurs last in list
  - (true, false)

اینجا ۳ تا characteristic داریم و برای هر کدوم یک پارتیشنینگ داریم که نتیجه ی این پارتیشن کردنه میشه k تا بلاک:
در روش functionality based از معنای برنامه استفاده میکنیم به کمک پارامترهای ورودی برنامه پس دانش رفتاری برنامه رو برای مدل سازی درنظر میگیریم

Step 3 : Modeling the Input Domain

- Partitioning characteristics into blocks and values is a very creative engineering step
- More blocks means more tests
- Partitioning often flows directly from the definition of characteristics and both steps are done together
  - Should evaluate them separately – sometimes fewer characteristics can be used with more blocks and vice versa
- Strategies for identifying values :
  - Include valid, invalid and special values
  - Sub-partition some blocks
  - Explore boundaries of domains
  - Include values that represent "normal use"
  - Try to balance the number of blocks in each characteristic
  - Check for completeness and disjointness

پارتیشنینگ و characteristic از هم جدای از هم باید بررسی و ارزیابی بشن

بیانگر رفتار عادی برنامه هستن

این دوتا رو توی پارتیشنینگ حتما باید چک کنیم تا مطمئن بشیم عملیات افرازمون درست بوده

Interface-Based –*triang()*

- *triang*() has one testable function and three integer inputs

First Characterization of TriTyp's Inputs

Characteristic	b_1	b_2	b_3
q_1 = "Relation of Side 1 to 0"	greater than 0	equal to 0	less than 0
q_2 = "Relation of Side 2 to 0"	greater than 0	equal to 0	less than 0
q_3 = "Relation of Side 3 to 0"	greater than 0	equal to 0	less than 0

اگه همه ی ترکیبات ممکن رو درنظر بگیریم ۲۷تا تست خاهیم داشت

این پیداکردن b1,b2,b3 میشه پارتیشنینگ که فضای ورودی ما رو به ۳تا بلاک میاد افراز میکنه

- A maximum of 3*3*3 = 27 tests
- Some triangles are valid, some are invalid
- Refining the characterization can lead to more tests …

میتونیم پارتیشنینگمون رو دقیق تر انجام بدیم و refineش کنیم ینی افرازمون رو دقیق تر و ریزبینانه تر انجام بدیم

Interface-Based IDM—*triang()*

Second Characterization of triang()'s Inputs

Characteristic	b_1	b_2	b_3	b_4
q_1 = "Refinement of q_1"	greater than 1	equal to 1	equal to 0	less than 0
q_2 = "Refinement of q_2"	greater than 1	equal to 1	equal to 0	less than 0
q_3 = "Refinement of q_3"	greater than 1	equal to 1	equal to 0	less than 0

اینجا تعداد تست هامون بیشتر میشه و باید هزینه ی بیشتری برای تست بپردازیم

اگه نوع اعداد اعشاری بود دیگه پارتیشنینگ ما کامل نبود چون اعداد بین صفر و یک رو درنظر نگرفتیم توی بلاک های بالا

- A maximum of 4*4*4 = 64 tests
- Complete because the inputs are integers (0 . . 1)

Possible values for partition q_1

Characteristic	b_1	b_2	b_3	b_4
Side 1	2	1	0	-1

Test boundary conditions

اگه اعداد ورودی مون صحیح باشن یعنی complete هه افرازمون چون همه رو بررسی کردیم

Functionality-Based IDM—*triang*()

- First two characterizations are based on syntax–parameters and their type
- A semantic level characterization could use the fact that the three integers represent a triangle

Equilateral is also isosceles !
We need to refine the example to make characteristics valid

Geometric Characterization of *triang*()'s Inputs

Characteristic	b_1	b_2	b_3	b_4
q_1 = "Geometric Classification"	scalene	isosceles	equilateral	invalid

دارای اضلاع نامساوی

متساوی الساقین

متساوی الاضلاع

- Equilateral is also is
- We need to refine t cteristics valid

What's wrong with this partitioning?

Correct Geometric Characterization of *triang*()'s Inputs

Characteristic	b_1	b_2	b_3	b_4
q_1 = "Geometric Classification"	scalene	isosceles, not equilateral	equilateral	invalid

مثلث متساوی الساقینی که متساوی الاضلاع نباشه

Functionality-Based IDM—*triang*()

- Values for this partitioning can be chosen as

Possible values for geometric partition q_1

Characteristic	b_1	b_2	b_3	b_4
Triangle	(4, 5, 6)	(3, 3, 4)	(3, 3, 3)	(3, 4, 8)

Functionality-Based IDM—*triang*()

- A different approach would be to break the geometric characterization into four separate characteristics

Four Characteristics for *triang*()

Characteristic	b_1	b_2
q_1 = "Scalene"	True	False
q_2 = "Isosceles"	True	False
q_3 = "Equilateral"	True	False
q_4 = "Valid"	True	False

چون خود کرکتریستیک ها باهم رابطه دارن میتونیم بینشون قید بذاریم مثلا بگیم: اگه مثلثمون متساوی الاضلاع بود پس متساوی الساقین هم هست

- Use constraints to ensure that
  - Equilateral = True implies Isosceles = True
  - Valid = False implies Scalene = Isosceles = Equilateral = False

Using More than One IDM

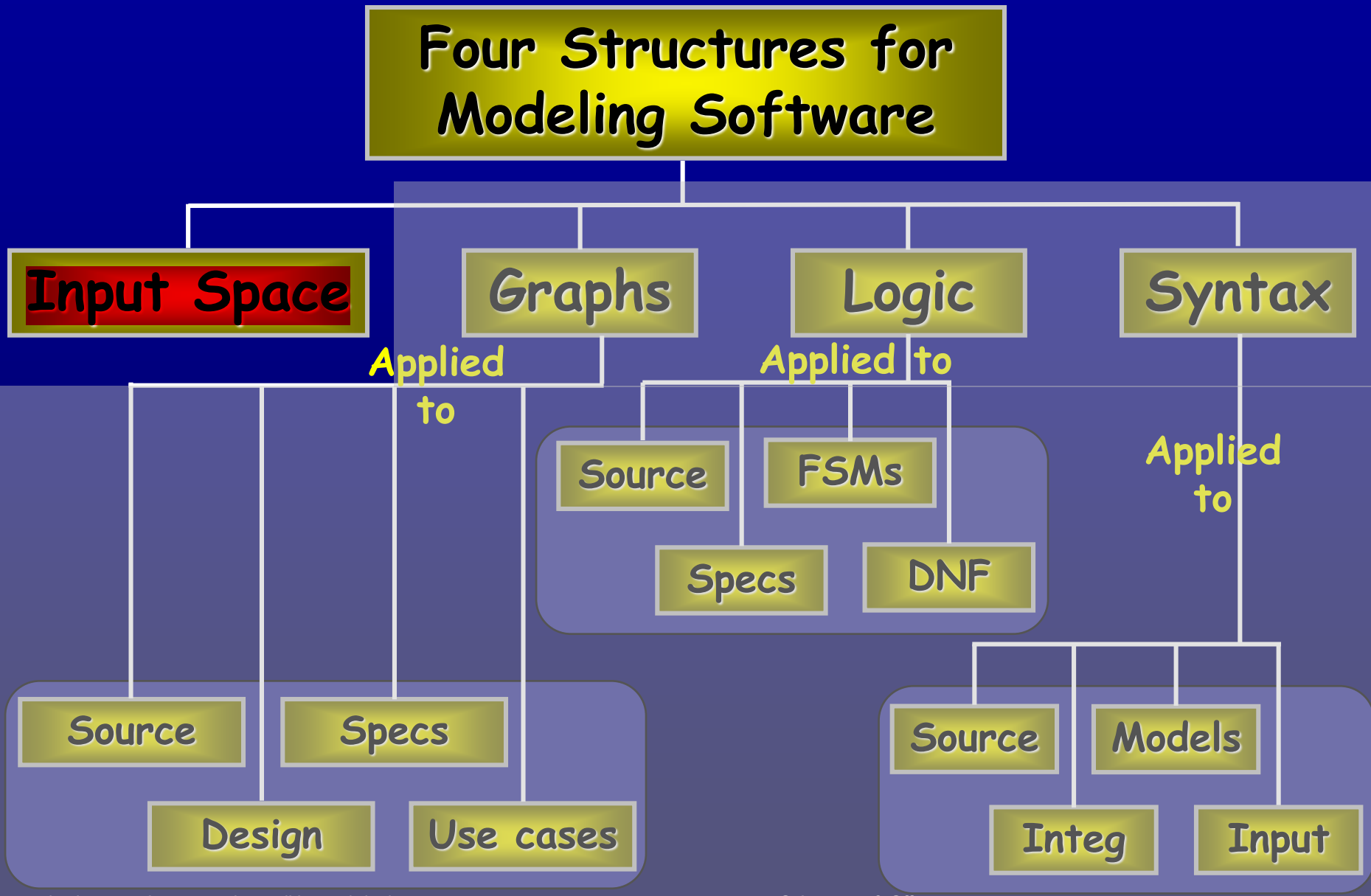
- Some programs may have dozens or even hundreds of parameters
- Create several small IDMs
  - A divide-and-conquer approach
- Different parts of the software can be tested with different amounts of rigor
  - For example, some IDMs may include a lot of invalid values
- It is okay if the different IDMs overlap
  - The same variable may appear in more than one IDM

Introduction to Software Testing
Chapter 6
Input Space Partition Testing

Paul Ammann & Jeff Offutt

http://www.cs.gmu.edu/~offutt/softwaretest/

Ch. 6 : Input Space Coverage



Modeling the Input Domain

- Step 1 : Identify testable functions
- Step 2 : Find all the parameters that can affect the behavior of a given testable function.
- Step 3 : Model the input domain
  - The domain is scoped by the parameters
  - The structure is defined in terms of characteristics
  - Each characteristic is partitioned into sets of blocks
  - Each block represents a set of values
  - This is the most creative design step in using ISP

ارزش این ولیوها یکسان است

Modeling the Input Domain (*cont*)

- Step 4 : Apply a test criterion to choose combinations of values
  - A test input has a value for each parameter
  - One block for each characteristic
  - Choosing all combinations is usually infeasible
  - Coverage criteria allow subsets to be chosen
- Step 5 : Refine combinations of blocks into test inputs
  - Choose appropriate values from each block

چه ترکیب هایی از ولیوهای بلاک های کرکتریستیک ها رو باید انتخاب کنیم برای ست کردن؟ نمیشه که همشون رو تست کنیم!!

اینکه چطوری بفهمیم یک کدوم زیرمجموعه از ترکیب ها روباید انتخاب کنیم براساس criteriaیی که داریم

Step 4 – Choosing Combinations of Values (6.2)

- Once characteristics and partitions are defined, the next step is to choose test values
- We use criteria – to choose effective subsets
- The most obvious criterion is to choose all combinations

All Combinations (ACoC) : All combinations of blocks from all characteristics must be used.

- Number of tests is the product of the number of blocks in each characteristic : $\prod_{i=1}^{Q}(B_i)$

حاصل ضرب تعداد بلاکهای هر کرکتریستیک میشه تعداد کل تست هامون

- The second characterization of triang() results in 4*4*4 = 64 tests
  - Too many ?

۳تا پارامتر داشتیم، هرپارامتر معادل یک کرکتریستیک بود و هرکرکتریستیک میشد ۴تا بلاک
پس ۶۴تا تست در کل داشتیم

ISP Criteria – All Combinations

- Consider the "second characterization" of Triang as given before:

Characteristic	b_1	b_2	b_3	b_4
q_1 = "Refinement of q_1"	greater than 1	equal to 1	equal to 0	less than 0
q_2 = "Refinement of q_2"	greater than 1	equal to 1	equal to 0	less than 0
q_3 = "Refinement of q_3"	greater than 1	equal to 1	equal to 0	less than 0

- For convenience, we relabel the blocks using abstractions:

Characteristic	b_1	b_2	b_3	b_4
A	A1	A2	A3	A4
B	B1	B2	B3	B4
C	C1	C2	C3	C4

ISP Criteria – ACoC Tests

A1 B1 C1	A2 B1 C1	A3 B1 C1	A4 B1 C1
A1 B1 C2	A2 B1 C2	A3 B1 C2	A4 B1 C2
A1 B1 C3	A2 B1 C3	A3 B1 C3	A4 B1 C3
A1 B1 C4	A2 B1 C4	A3 B1 C4	A4 B1 C4
A1 B2 C1	A2 B2 C1	A3 B2 C1	A4 B2 C1
A1 B2 C2	A2 B2 C2	A3 B2 C2	A4 B2 C2
A1 B2 C3	A2 B2 C3	A3 B2 C3	A4 B2 C3
A1 B2 C4	A2 B2 C4	A3 B2 C4	A4 B2 C4
A1 B3 C1	A2 B3 C1	A3 B3 C1	A4 B3 C1
A1 B3 C2	A2 B3 C2	A3 B3 C2	A4 B3 C2
A1 B3 C3	A2 B3 C3	A3 B3 C3	A4 B3 C3
A1 B3 C4	A2 B3 C4	A3 B3 C4	A4 B3 C4
A1 B4 C1	A2 B4 C1	A3 B4 C1	A4 B4 C1
A1 B4 C2	A2 B4 C2	A3 B4 C2	A4 B4 C2
A1 B4 C3	A2 B4 C3	A3 B4 C3	A4 B4 C3
A1 B4 C4	A2 B4 C4	A3 B4 C4	A4 B4 C4

از هر کرکتریستیک فقط دوتا بلاک ها معتبر هستن:
A1,A2
B1,B2
C1,C2
که میشه :
8 = 2*2*2

ACoC yields
4*4*4 = 64 tests
for Triang!

This is almost certainly more than we need

Only 8 are valid
(all sides greater than zero)

ISP Criteria – Each Choice

- 64 tests for triang() is almost certainly way too many
- One criterion comes from the idea that we should try at least one value from each block

به ازای هربلاک اگه یک ولیو فقط داشته باشیم کافیه یعنی مثلا اگه گفتیم به ازای کرکتریستیک اولم میام A1 رو میذارم کافیه و کافیه یکی از بلاک های کرکتریستیک اول مثلا توی اون تستمون وجود داشته باشه

Each Choice Coverage (ECC) : One value from each block for each characteristic must be used in at least one test case.

- Number of tests is the number of blocks in the largest characteristic : $\mathbf{Max}_{i=1}^{Q}(B_i)$

کرکتریستیکی که بیشترین تعداد بلاک رو تولید میکنه

For *triang*() : A1, B1, C1
A2, B2, C2
A3, B3, C3
A4, B4, C4

اگه ما A1 رو توی یکی از تست هامون داشته باشیم کافیه

Write down EC tests
Use the abstract labels (A1,A2, …)

Substituting values: 2, 2, 2
1, 1, 1
0, 0, 0
-1, -1, -1

Suggest values …

ISP Criteria – Pair-Wise

- Each choice yields few tests—cheap but maybe ineffective
- Another approach combines values with other values

ارتباطات کرکتریستیک
های مختلف رو باتوجه به
بلاک هاش درنظر بگیره

Pair-Wise Coverage (PWC) : A value from each block for each characteristic must be combined with a value from every block for each other characteristic.

- Number of tests is at least the product of two largest characteristics $(\text{Max}_{i=1}^{Q}(B_i)) * (\text{Max}_{j=1,\, j!=i}^{Q}(B_j))$

For *triang()* :

A1, B1, C1	A1, B2, C2	A1, B3, C3	A1, B4, C4
A2, B1, C2	A2, B2, C3	A2, B3, C4	A2, B4, C1
A3, B1, C3	A3, B2, C4	A3, B3, C1	A3, B4, C2
A4, B1, C4	A4, B2, C1	A4, B3, C2	A4, B4, C3

Write down PWC tests
Use the abstract labels
(Hint: Should be 16 tests)

Given the above example of three partitions with blocks [A, B], [1, 2, 3], and [x, y], then PWC will need tests to cover the following 16 combinations:

(A, 1)	(B, 1)	(1, x)
(A, 2)	(B, 2)	(1, y)
(A, 3)	(B, 3)	(2, x)
(A, x)	(B, x)	(2, y)
(A, y)	(B, y)	(3, x)
		(3, y)

> CRITERION **6.3 Pair-Wise Coverage (PWC):** *A value from each block for each characteristic must be combined with a value from every block for each other characteristic.*

Given the above example of three partitions with blocks [A, B], [1, 2, 3], and [x, y], then PWC will need tests to cover the following 16 combinations:

(A, 1)	(B, 1)	(1, x)
(A, 2)	(B, 2)	(1, y)
(A, 3)	(B, 3)	(2, x)
(A, x)	(B, x)	(2, y)
(A, y)	(B, y)	(3, x)
		(3, y)

PWC allows the same test case to cover more than one unique pair of values. So the above combinations can be combined in several ways, including:

(A, 1, x)	(B, 1, y)
(A, 2, x)	(B, 2, y)
(A, 3, x)	(B, 3, y)
(A, -, y)	(B, -, x)

The tests with '-' mean that any block can be used.

A test set that satisfies PWC will pair each value with each other value, or have at least($Max_{i=1}{}^{Q}B_i$) * ($Max_{i=1,\,j=1}{}^{Q}B_j$) values. Each characteristic in `triang()` (Table 6.3) has four blocks; so at least 16 tests are required.

Several algorithms to satisfy PWC have been published and appropriate references are provided in the bibliography section of the chapter.

A natural extension to Pair-Wise Coverage is to require groups of *t* values instead of pairs.

Table 6.2. Second partitioning of `triang()`'s inputs (interface-based).

Partition	b_1	b_2	b_3	b_4
q_1 = "Length of Side 1"	*greater than 1*	*equal to 1*	*equal to 0*	*less than 0*
q_2 = "Length of Side 2"	*greater than 1*	*equal to 1*	*equal to 0*	*less than 0*
q_3 = "Length of Side 3"	*greater than 1*	*equal to 1*	*equal to 0*	*less than 0*

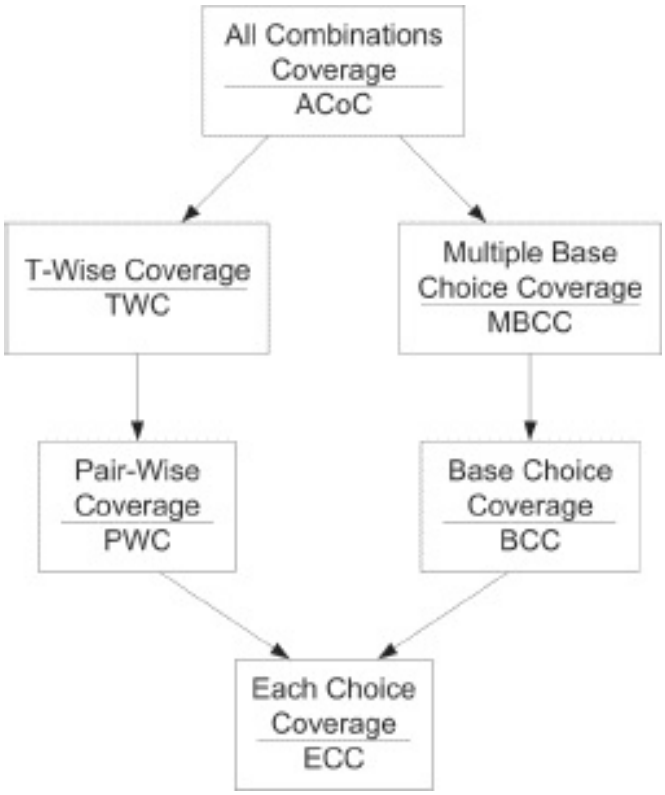
2. Enumerate all 16 tests to satisfy the Pair-Wise (PWC) criterion for the second categorization of TriTyp's inputs in Table 4.2. Use the values in Table 4.3.

Solution:

Note: Lots of possibilities here, as suggested by the ways in which pairs are chosen below.

$\{(2,2,2),$
$(2,1,1),$
$(2,0,0),$
$(2,-1,-1),$
$(1,2,1),$
$(1,1,2),$
$(1,0,-1),$
$(1,-1,0),$
$(0,2,0),$
$(0,1,-1),$
$(0,0,2),$
$(0,-1,1),$
$(-1,2,-1),$
$(-1,1,0),$

$(-1,0,1),$
$(-1,-1,2)\}$



ISP Criteria –T-Wise

- A natural extension is to require combinations of *t* values instead of *2*

t-Wise Coverage (TWC) : A value from each block for each group of t characteristics must be combined.

- Number of tests is at least the product of *t* largest characteristics
- If all characteristics are the same size, the formula is

$$(\text{Max}_{i=1}^{Q}(B_i))^t$$

- If *t* is the number of characteristics *Q*, then all combinations
- That is … *Q-wise = AC*

all combination

- *t*-wise is expensive and benefits are not clear

ISP Criteria – Base Choice

- Testers sometimes recognize that certain values are important
- This uses domain knowledge of the program

ما میدونیم یه سری تست ها رو حتما باید داشته باشیم و برمبنای دانش زمینه ای برنامومون یه سری تست رو میگیم میخایم حتما

Base Choice Coverage (BCC) : A base choice block is chosen for each characteristic, and a base test is formed by using the base choice for each characteristic. Subsequent tests are chosen by holding all but one base choice constant and using each non-base choice in each other characteristic.

- Number of tests is one base test + one test for each other block $1 + \sum_{i=1}^{Q} (B_i - 1)$

Q=3,Bi=4

به ازای کرکتریستیک اول، دوتا رو ثابت میگیریم و بلاک های کرکتریستیک باقی مونده رو مینویسیم

For *triang*() : Base A1, B1, C1 A1, B1, C2 A1, B2, C1 A2, B1, C1
A1, B1, C3 A1, B3, C1 A3, B1, C1
A1, B1, C4 A1, B4, C1 A4, B1, C1

Write down BCC tests

اینکه هر سه تا ضلع مقدارشون بزرگتر از یک باشه برامون خیلی مهمه پس میذاریم بیس باشه

n-1 تا از کرکتریستیک ها رو ثابت میگیریم و فقط یکی رو هردفه تغییر میدیم

A1,C1 رو ثابت میگیریم و بلاک های مختلف کرکتریستیک دوم رو مینویسیم

Base Choice Notes

- The base test must be feasible
  - That is, all base choices must be compatible
- Base choices can be
  - Most likely from an end-use point of view
  - Simplest
  - Smallest
  - First in some ordering
- Happy path tests often make good base choices
- The base choice is a crucial design decision
  - Test designers should document why the choices were made

بیس چیس ها باید compatible باشن
ینی نباید کانفلیکت داشته باشن

base choice میتونه
از دیدگاه end user
بیاد و مشتری بگه چی
براش مهمه

ISP Criteria – Multiple Base Choice

- We sometimes have more than one logical base choice

Multiple Base Choice Coverage (MBCC) : At least one, and possibly more, base choice blocks are chosen for each characteristic, and base tests are formed by using each base choice for each characteristic at least once. Subsequent tests are chosen by holding all but one base choice constant for each base test and using each non-base choice in each other characteristic.

- **If *M* base tests and m_i base choices for each characteristic:**

M=2
Q=3

$$M + \sum_{i=1}^{Q} (M * (B_i - m_i))$$

Bi=4,mi=2

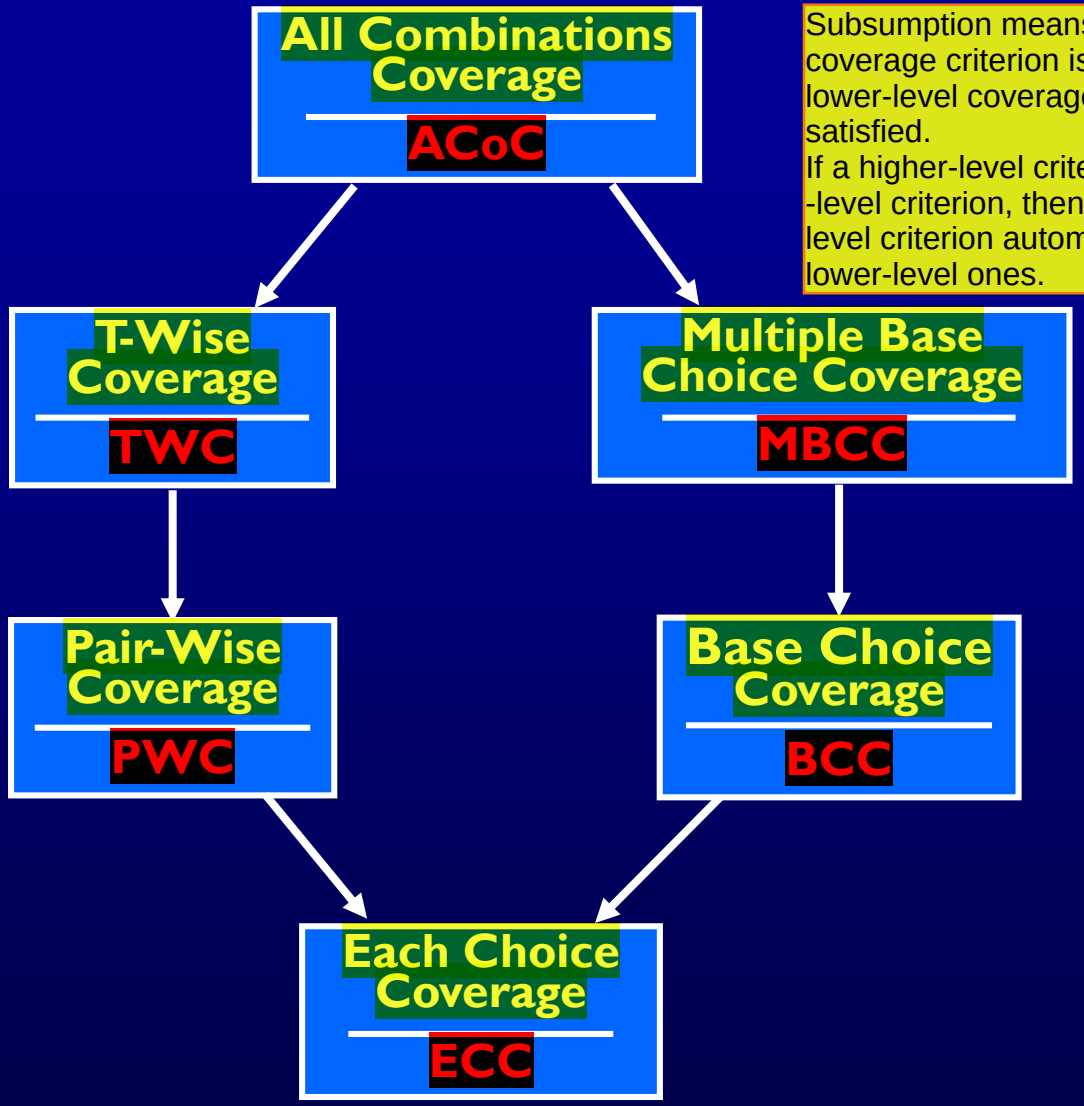
For *triang*() : Bases

به ازای هر characteristicای ما ۴تا بلاک داریم
ولی دوتا حالت یا دوتا بلاک از هر characteristicای برداشتیم

تعداد کل تست هامون میشه ۱۴ تا

A1, B1, C1	A1, B1, C3	A1, B3, C1	A3, B1, C1
	A1, B1, C4	A1, B4, C1	A4, B1, C1
A2, B2, C2	A2, B2, C3	A2, B3, C2	A3, B2, C2
	A2, B2, C4	A2, B4, C2	A4, B2, C2

ISP Coverage Criteria Subsumption



Subsumption means that if a higher-level coverage criterion is satisfied, then all lower-level coverage criteria are also satisfied.
If a higher-level criterion subsumes a lower -level criterion, then satisfying the higher-level criterion automatically satisfies the lower-level ones.

Constraints Among Characteristics (6.3)

- Some combinations of blocks are infeasible
  - "less than zero" and "scalene" … not possible at the same time
- These are represented as constraints among blocks
- Two general types of constraints
  - A block from one characteristic cannot be combined with a specific block from another
  - A block from one characteristic can ONLY BE combined with a specific block form another characteristic
- Handling constraints depends on the criterion used
  - ACC, PWC, TWC : Drop the infeasible pairs
  - BCC, MBCC : Change a value to another non-base choice to find a feasible combination

Example Handling Constraints

```
public boolean findElement (List list, Object element)
// Effects: if list or element is null throw NullPointerException
//          else return true if element is in the list, false otherwise
```

Characteristic	Block 1	Block 2	Block 3	Block 4
A : length and contents	One element	More than one, unsorted	More than one, sorted	More than one, all identical
B : match	element not found	element found once	element found more than once	
Invalid combinations : (A1, B3), (A4, B2)				

element cannot be in a one-element list more than once

If the list only has one element, but it appears multiple times, we cannot find it just once

Input Space Partitioning Summary

- Fairly easy to apply, even with no automation
- Convenient ways to add more or less testing
- Applicable to all levels of testing – unit, class, integration, system, etc.
- Based only on the input space of the program, not the implementation

Simple, straightforward, effective, and widely used

Introduction to Software Testing
(2nd edition)
Chapter 7.1, 7.2

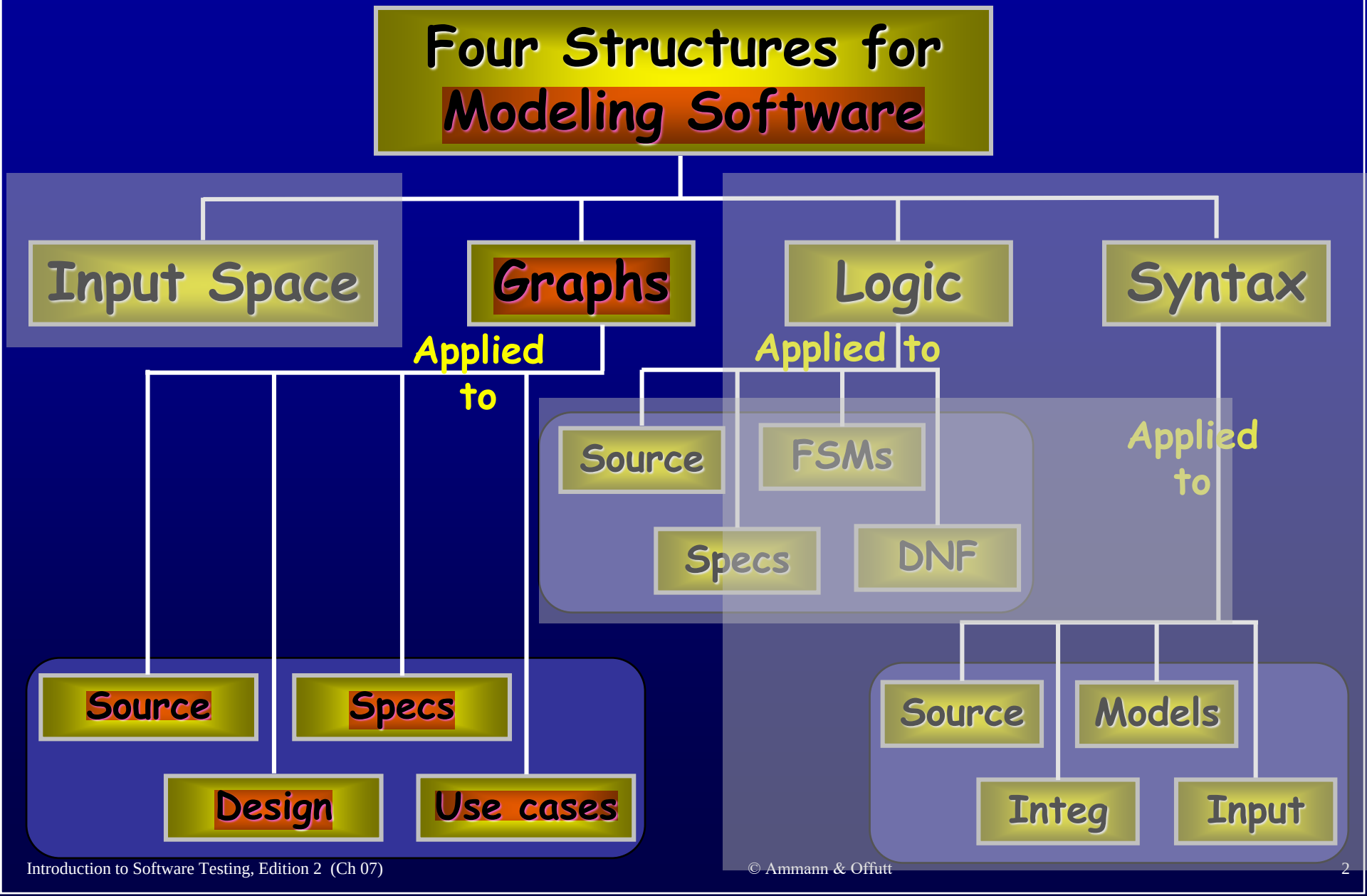
Overview Graph Coverage Criteria

Paul Ammann & Jeff Offutt

http://www.cs.gmu.edu/~offutt/softwaretest/

Update, January 2016

Ch. 7 : Graph Coverage



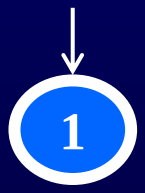
Covering Graphs (7.1)

- Graphs are the most commonly used structure for testing
- Graphs can come from many sources
  - Control flow graphs
  - Design structure
  - FSMs and statecharts
  - Use cases
- Tests usually are intended to "cover" the graph in some way

Definition of a Graph

- A set N of nodes, N is not empty
- A set N_0 of initial nodes, N_0 is not empty
- A set N_f of final nodes, N_f is not empty
- A set E of edges, each edge from one node to another
  - (n_i , n_j), i is predecessor, j is successor

Is this a graph?

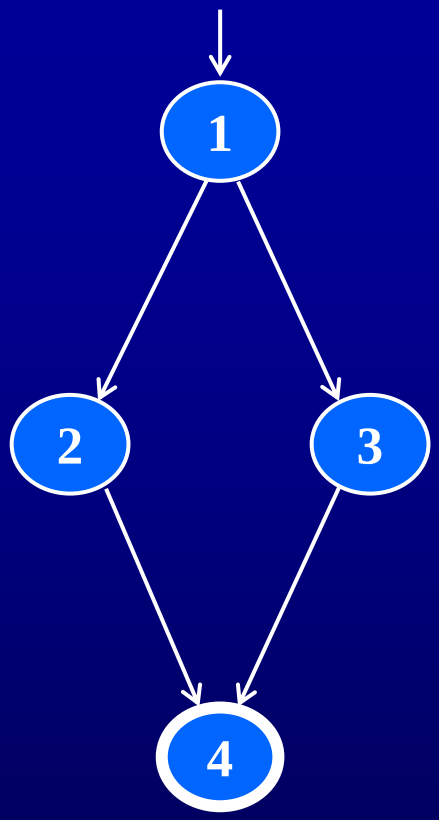


$N_0 = \{ 1 \}$

$N_f = \{ 1 \}$

$E = \{ \}$

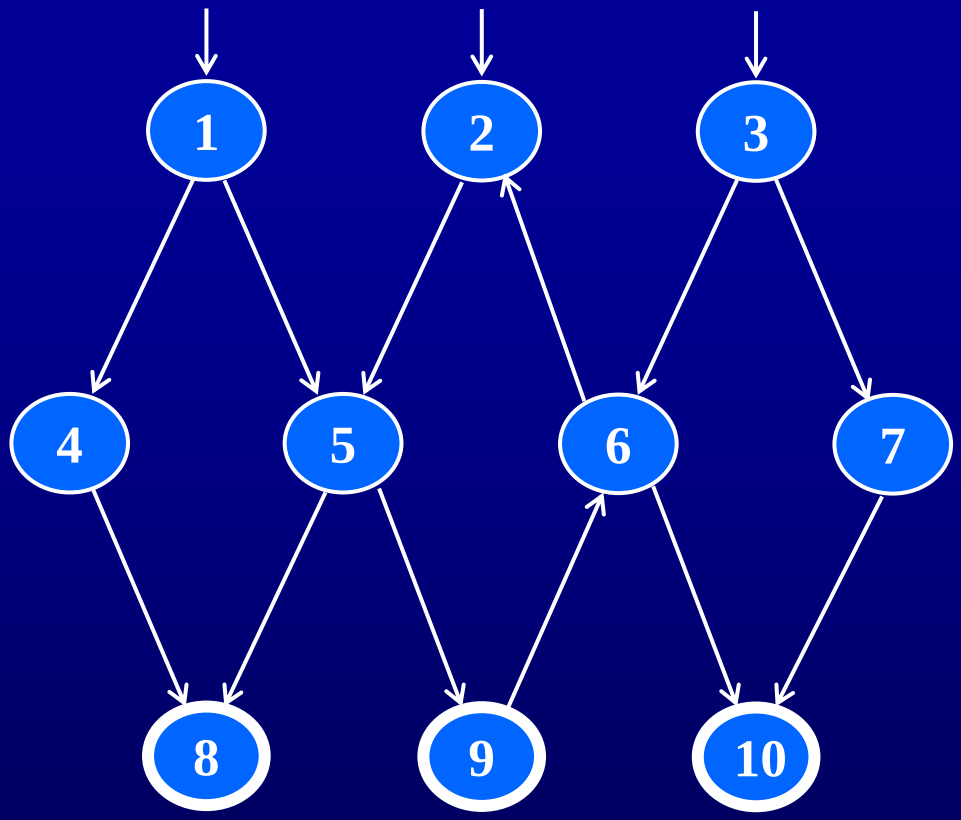
Example Graphs



N_0 = { 1 }

N_f = { 4 }

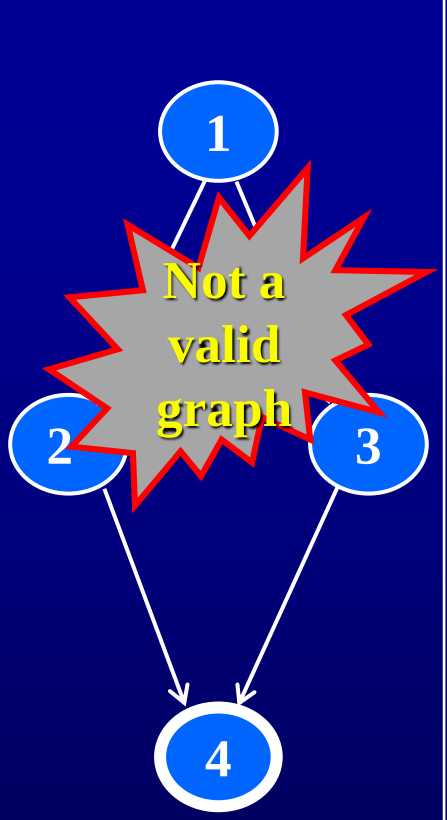
E = { (1,2), (1,3), (2,4), (3,4) }



N_0 = { 1, 2, 3 }

N_f = { 8, 9, 10 }

E = { (1,4), (1,5), (2,5), (3,6), (3, 7), (4, 8), (5,8), (5,9), (6,2), (6,10), (7,10) (9,6) }



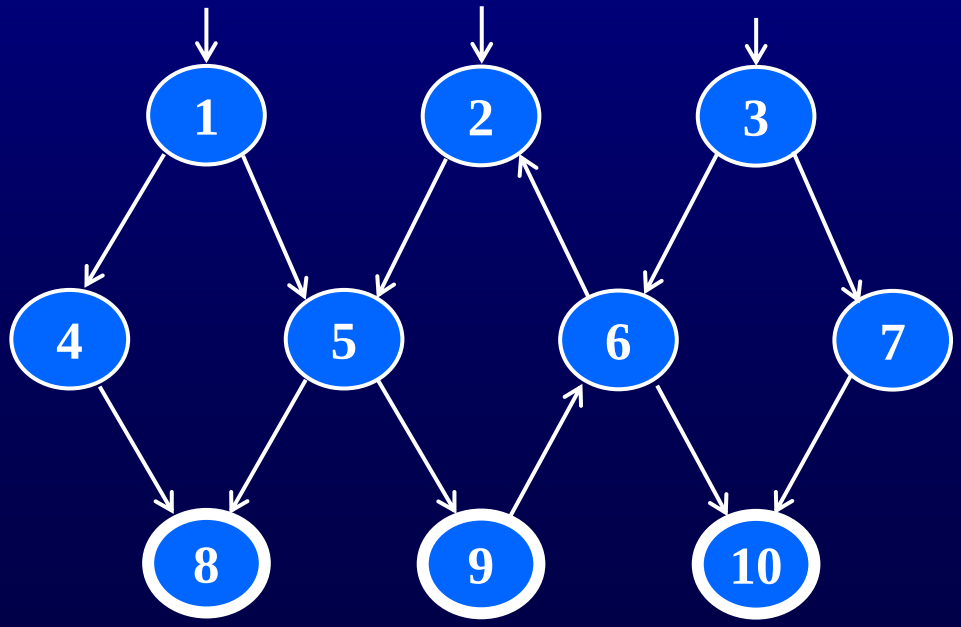
N_0 = { }

N_f = { 4 }

E = { (1,2), (1,3), (2,4), (3,4) }

Paths in Graphs

- **Path** : A sequence of nodes – $[n_1, n_2, \ldots, n_M]$
  - Each pair of nodes is an edge
- **Length** : The number of edges
  - A single node is a path of length 0
- **Subpath** : A subsequence of nodes in p is a subpath of p



A Few Paths

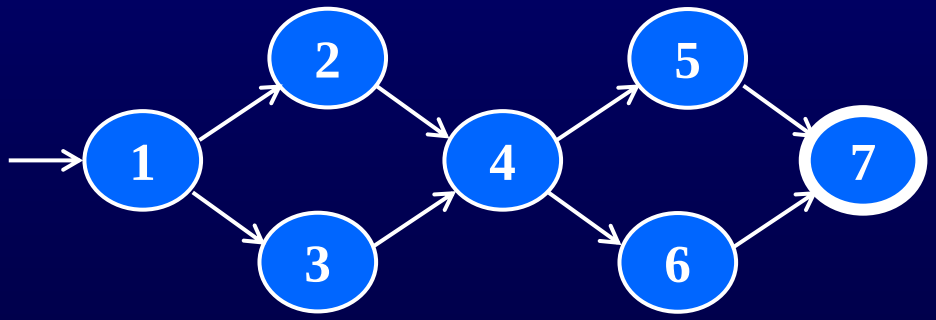
[1, 4, 8]

[2, 5, 9, 6, 2]

[3, 7, 10]

Test Paths and SESEs

- **Test Path** : A path that starts at an initial node and ends at a final node
- Test paths represent execution of test cases
  - Some test paths can be executed by many tests
  - Some test paths cannot be executed by any tests
- **SESE graphs** : All test paths start at a single node and end at another node
  - Single-entry, single-exit
  - N0 and Nf have exactly one node



Double-diamond graph

Four test paths

[1, 2, 4, 5, 7]
[1, 2, 4, 6, 7]
[1, 3, 4, 5, 7]
[1, 3, 4, 6, 7]

Visiting and Touring

- Visit : A test path *p* *visits* node *n* if *n* is in *p*

  A test path *p* *visits* edge e if e is in *p*
- Tour : A test path *p* *tours* subpath *q* if *q* is a subpath of *p*

Path [1, 2, 4, 5, 7]

Visits nodes 1, 2, 4, 5, 7

Visits edges (1, 2), (2, 4), (4, 5), (5, 7)

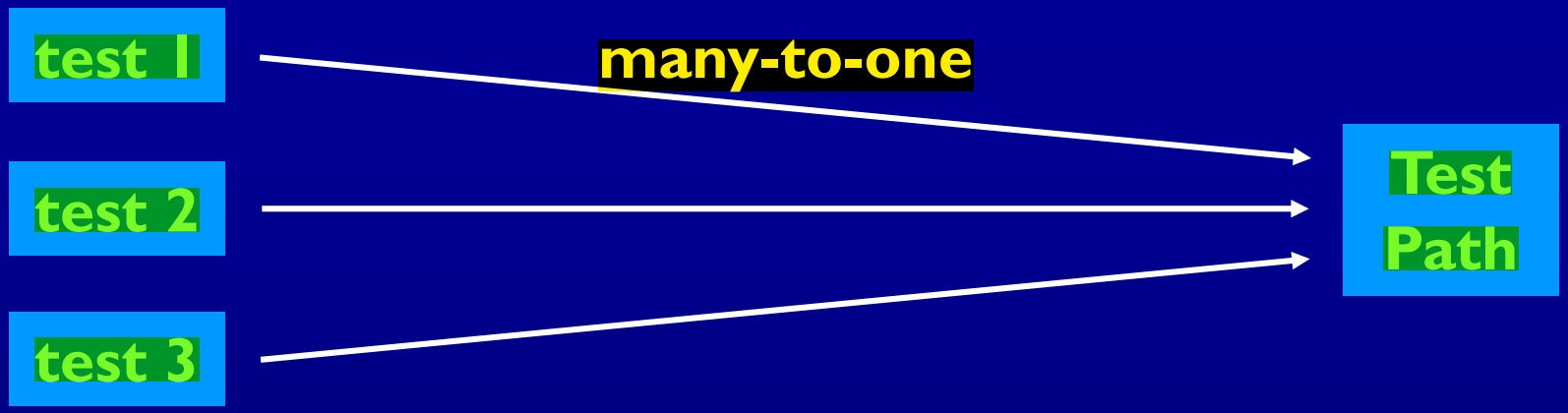
Tours subpaths [1, 2, 4], [2, 4, 5], [4, 5, 7], [1, 2, 4, 5], [2, 4, 5, 7], [1, 2, 4, 5, 7]

(***Also, each edge is technically a subpath***)

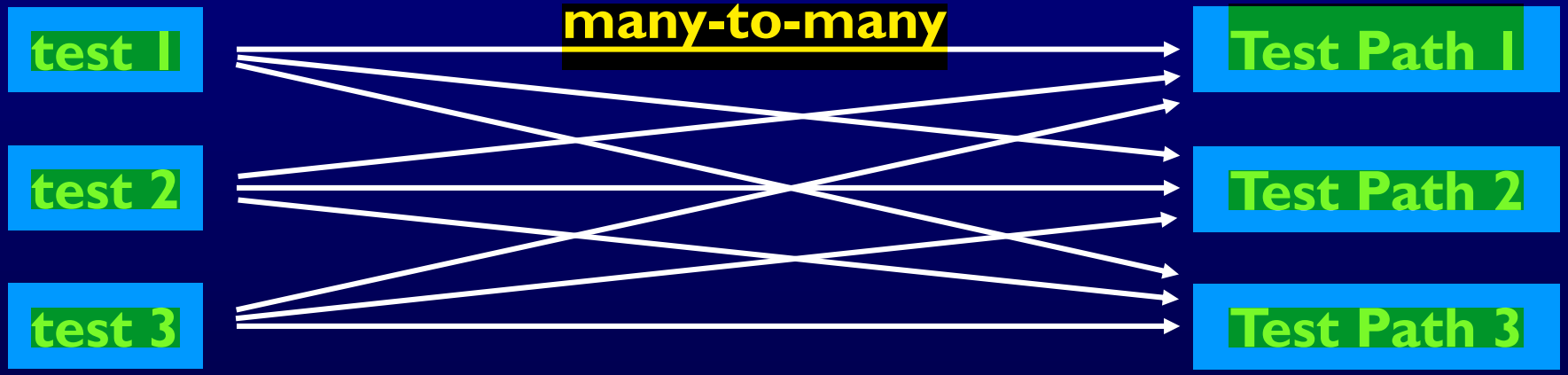
Tests and Test Paths

- path (*t*) : The test path executed by test *t*
- path (*T*) : The set of test paths executed by the set of tests *T*
- Each test executes one and only one test path
  - Complete execution from a start node to an final node
- A location in a graph (node or edge) can be reached from another location if there is a sequence of edges from the first location to the second
  - *Syntactic reach* : A subpath exists in the graph
  - *Semantic reach* : A test exists that can execute that subpath
  - This distinction will become important in section 7.3

Tests and Test Paths



Deterministic software–test always executes the same test path



Non-deterministic software–the same test can execute different test paths

Testing and Covering Graphs (7.2)

- We use graphs in testing as follows :
  - Develop a model of the software as a graph
  - Require tests to visit or tour specific sets of nodes, edges or subpaths
- Test Requirements (TR) : Describe properties of test paths
- Test Criterion : Rules that define test requirements
- Satisfaction : *Given a set TR of test requirements for a criterion C, a set of tests T satisfies C on a graph if and only if for every test requirement in TR, there is a test path in path(T) that meets the test requirement tr*

- Structural Coverage Criteria : Defined on a graph just in terms of nodes and edges
- Data Flow Coverage Criteria : Requires a graph to be annotated with references to variables

Node and Edge Coverage

- The first (and simplest) two criteria require that each node and edge in a graph be executed

Node Coverage (NC) : Test set *T* satisfies node coverage on graph *G* iff for every syntactically reachable node *n* in *N*, there is some path *p* in *path(T)* such that *p* visits *n*.

- This statement is a bit cumbersome, so we abbreviate it in terms of the set of test requirements

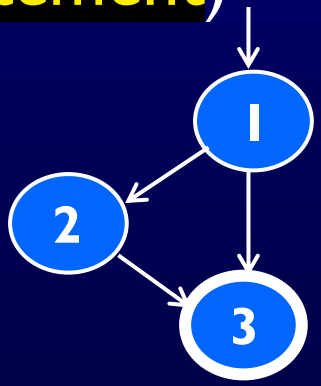
Node Coverage (NC) : TR contains each reachable node in G.

Node and Edge Coverage

- Edge coverage is slightly stronger than node coverage

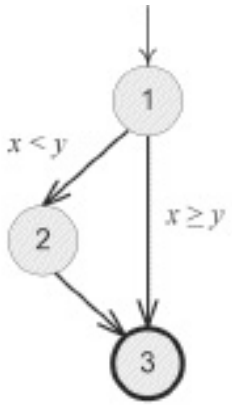
Edge Coverage (EC) : TR contains each reachable path of length up to 1, inclusive, in G.

- The phrase "*length up to 1*" allows for graphs with one node and no edges
- NC and EC are only different when there is an edge and another subpath between a pair of nodes (as in an "if-else" statement)



Node Coverage : TR = { 1, 2, 3 }
Test Path = [1, 2, 3]

Edge Coverage : TR = { (1, 2), (1, 3), (2, 3) }
Test Paths = [1, 2, 3]
[1, 3]



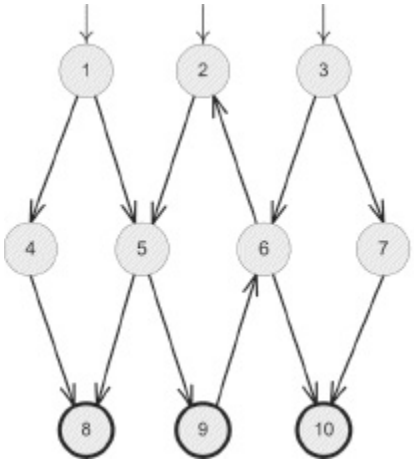
path (t_1) = [1, 2, 3]
path (t_2) = [1, 3]

$T_1 = \{ t_1 \}$
T_1 satisfies node coverage on the graph

(a) Node Coverage

$T_2 = \{ t_1 , t_2 \}$
T_2 satisfies edge coverage on the graph

(b) Edge Coverage

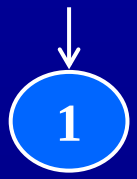


Path Examples	
1	1, 4, 8
2	2, 5, 9, 6, 2
3	3, 7, 10

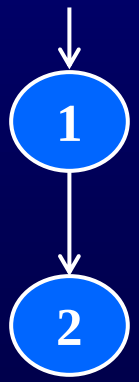
Invalid Path Examples	
1	1, 8
2	4, 5
3	3, 7, 9

Paths of Length 1 and 0

- A graph with only one node will not have any edges



- It may seem trivial, but formally, Edge Coverage needs to require Node Coverage on this graph
- Otherwise, Edge Coverage will not subsume Node Coverage
  - So we define "length up to 1" instead of simply "length 1"
- We have the same issue with graphs that only have one edge – for Edge-Pair Coverage …

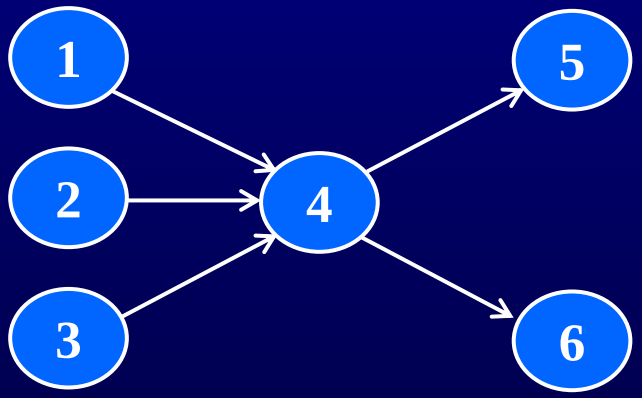


Covering Multiple Edges

- Edge-pair coverage requires pairs of edges, or subpaths of length 2

Edge-Pair Coverage (EPC) : TR contains each reachable path of length up to 2, inclusive, in G.

- The phrase "length up to 2" is used to include graphs that have less than 2 edges



Edge-Pair Coverage :

TR = { [1,4,5], [1,4,6], [2,4,5], [2,4,6], [3,4,5], [3,4,6] }

- The logical extension is to require all paths …

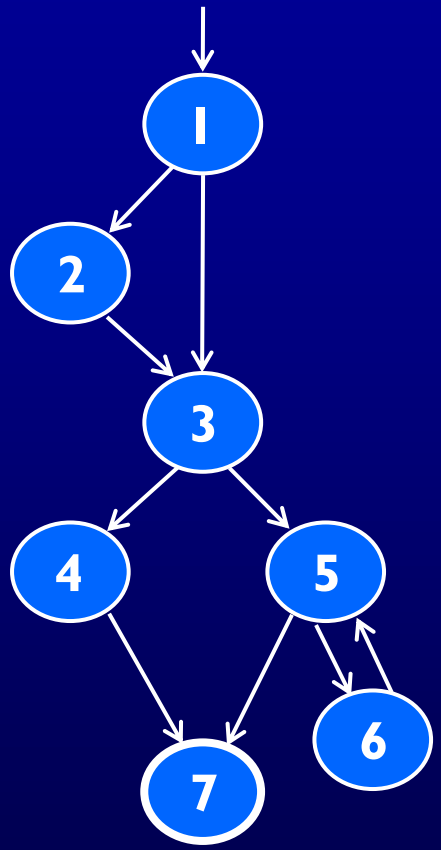
Covering Multiple Edges

Complete Path Coverage (CPC) : TR contains all paths in G.

Unfortunately, this is impossible if the graph has a loop, so a weak compromise makes the tester decide which paths:

Specified Path Coverage (SPC) : TR contains a set S of test paths, where S is supplied as a parameter.

Structural Coverage Example



Node Coverage

TR = { 1, 2, 3, 4, 5, 6, 7 }

Test Paths: [1, 2, 3, 4, 7] [1, 2, 3, 5, 6, 5, 7]

Edge Coverage

TR = { (1,2), (1, 3), (2, 3), (3, 4), (3, 5), (4, 7), (5, 6), (5, 7), (6, 5) }

Test Paths: [1, 2, 3, 4, 7] [1, 3, 5, 6, 5, 7]

Edge-Pair Coverage

TR = {[1,2,3], [1,3,4], [1,3,5], [2,3,4], [2,3,5], [3,4,7], [3,5,6], [3,5,7], [5,6,5], [6,5,6], [6,5,7] }

Test Paths: [1, 2, 3, 4, 7] [1, 2, 3, 5, 7] [1, 3, 4, 7] [1, 3, 5, 6, 5, 6, 5, 7]

Complete Path Coverage

Test Paths: [1, 2, 3, 4, 7] [1, 2, 3, 5, 7] [1, 2, 3, 5, 6, 5, 6] [1, 2, 3, 5, 6, 5, 6, 5, 7] [1, 2, 3, 5, 6, 5, 6, 5, 6, 5, 7] ...

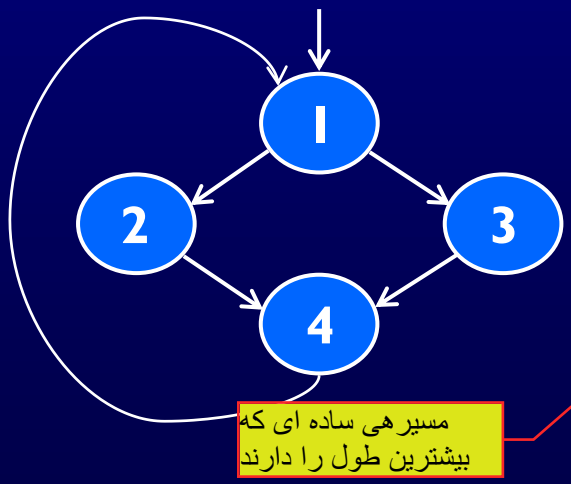
Handling Loops in Graphs

- If a graph contains a loop, it has an infinite number of paths
- Thus, CPC is not feasible
- SPC is not satisfactory because the results are subjective and vary with the tester
- Attempts to "deal with" loops:
  - 1970s : Execute cycles once ([4, 5, 4] in previous example, informal)
  - 1980s : Execute each loop, exactly once (formalized)
  - 1990s : Execute loops 0 times, once, more than once (informal description)
  - 2000s : Prime paths (touring, sidetrips, and detours)

Simple Paths and Prime Paths

- Simple Path : *A path from node ni to nj is simple if no node appears more than once, except possibly the first and last nodes are the same*
  - No internal loops
  - A loop is a simple path
- Prime Path : *A simple path that does not appear as a proper subpath of any other simple path*

در تعریف مسیرهای ساده:
مسیرهای به طول صفرکه نودهامون هستن ومسیر به طول یک که یال هامونه و ... را داریم



مسیرهی ساده ای که بیشترین طول را دارند

Simple Paths : [1,2,4,1], [1,3,4,1], [2,4,1,2], [2,4,1,3], [3,4,1,2], [3,4,1,3], [4,1,2,4], [4,1,3,4], [1,2,4], [1,3,4], [2,4,1], [3,4,1], [4,1,2], [4,1,3], [1,2], [1,3], [2,4], [3,4], [4,1], [1], [2], [3], [4]

Prime Paths : [2,4,1,2], [2,4,1,3], [1,3,4,1], [1,2,4,1], [3,4,1,2], [4,1,3,4], [4,1,2,4], [3,4,1,3]

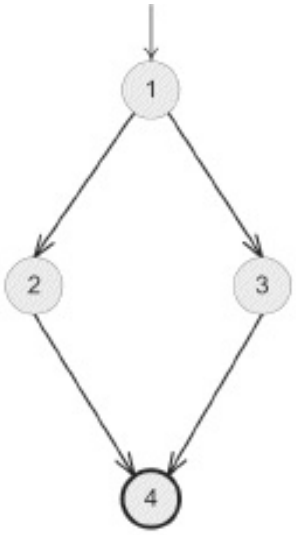
ماکس تعداد نودی که در این مسیرها میبینیم به تعدادنودهای گرافمون است

Prime Path Coverage

- A simple, elegant and finite criterion that requires loops to be executed as well as skipped

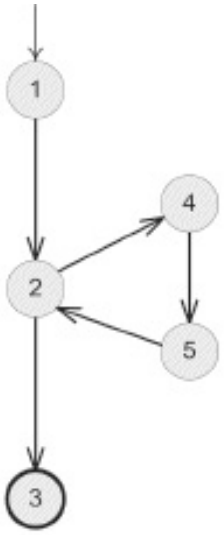
Prime Path Coverage (PPC) : TR contains each prime path in G.

- Will tour all paths of length 0, 1, …
- That is, it subsumes node and edge coverage
- PPC almost, but not quite, subsumes EPC …



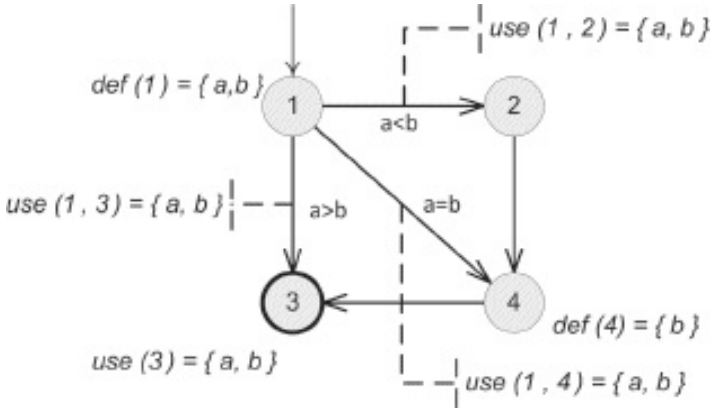
Prime Paths = { [1, 2, 4], [1, 3, 4] }
path (t_1) = [1, 2, 4]
path (t_2) = [1, 3, 4]
$T_1 = \{t_1, t_2\}$
T_1 satisfies prime path coverage on the graph

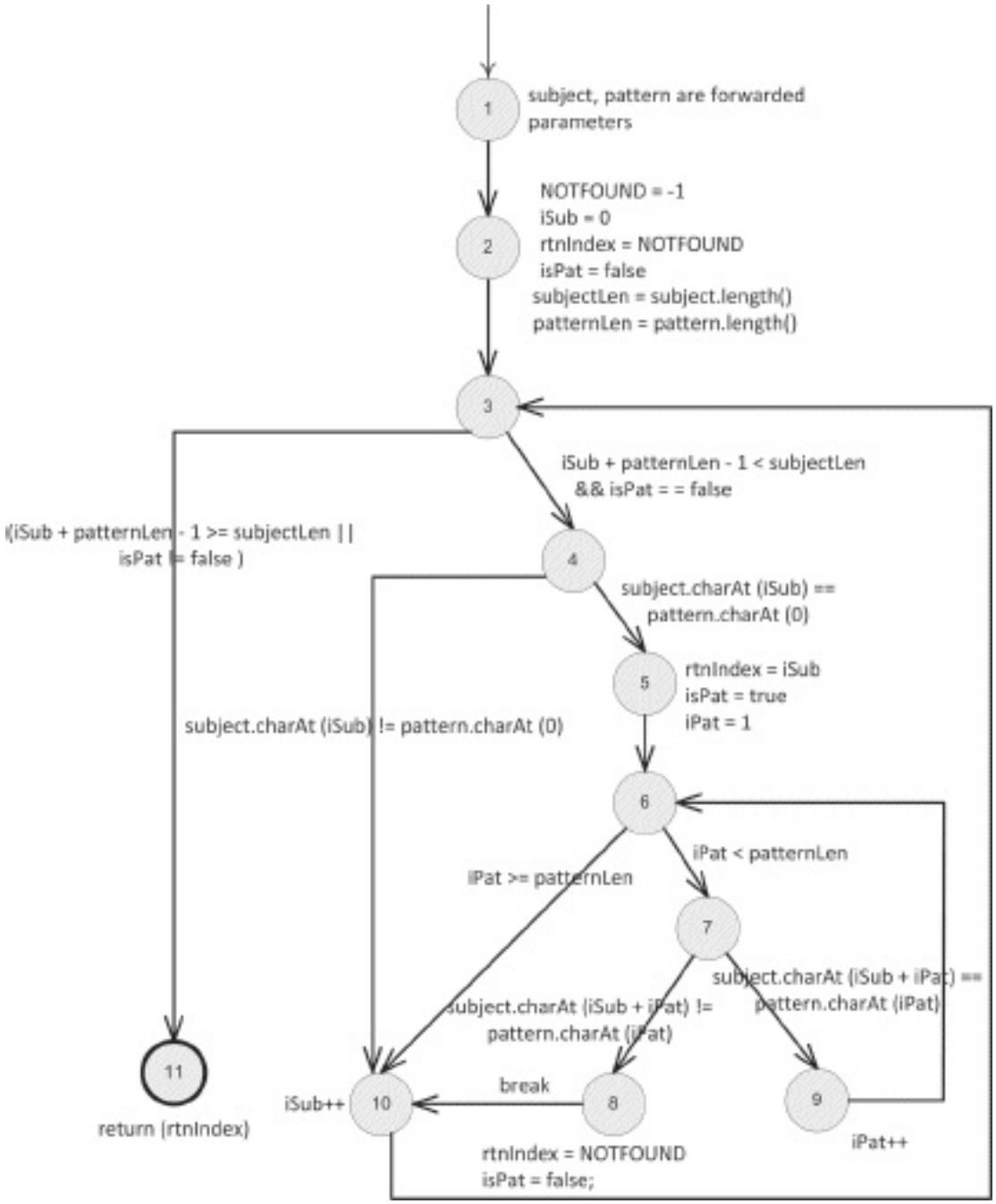
(a) Prime Path Coverage on a Graph With No Loops



Prime Paths = { [1, 2, 3], [1, 2, 4, 5], [2, 4, 5, 2], [4, 5, 2, 4], [5, 2, 4, 5], [4, 5, 2, 3] }
path (t_3) = [1, 2, 3]
path (t_4) = [1, 2, 4, 5, 2, 4, 5, 2, 3]
$T_2 = \{t_3, t_4\}$
T_2 satisfies prime path coverage on the graph

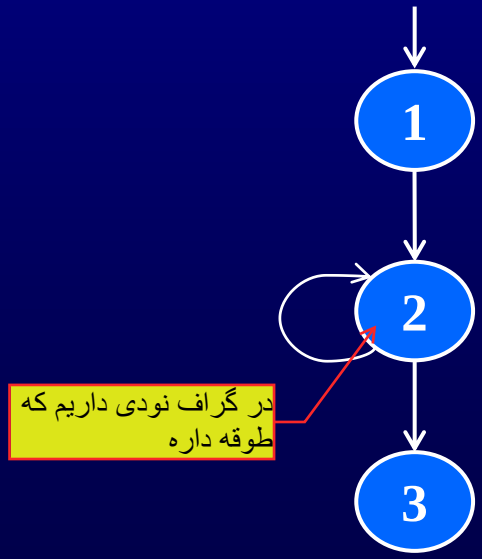
(b) Prime Path Coverage on a Graph With Loops




1
subject, pattern are forwarded
parameters
2
NOTFOUND = -1
iSub = 0
rtnIndex = NOTFOUND
isPat = false
subjectLen = subject.length()
patternLen = pattern.length()
3
iSub + patternLen - 1 < subjectLen
&& isPat = = false
(iSub + patternLen - 1 >= subjectLen | |
isPat != false)
4
subject.charAt (iSub) ==
pattern.charAt (0)
5
rtnIndex = iSub
isPat = true
iPat = 1
subject.charAt (iSub) != pattern.charAt (0)
6
iPat < patternLen
iPat >= patternLen
7
subject.charAt (iSub + iPat) ==
pattern.charAt (iPat)
subject.charAt (iSub + iPat) !=
pattern.charAt (iPat)
11
return (rtnIndex)
iSub++
10
break
8
9
iPat++
rtnIndex = NOTFOUND
isPat = false;

PPC Does Not Subsume EPC

- If a node *n* has an edge to itself (*self edge*), EPC requires [*n, n, m*] and [*m, n, n*]
- [*n, n, m*] is not prime
- Neither [*n, n, m*] nor [*m, n, n*] are simple paths (not prime)



EPC Requirements :

یال های مجاورهم رو باید داشته باشیم

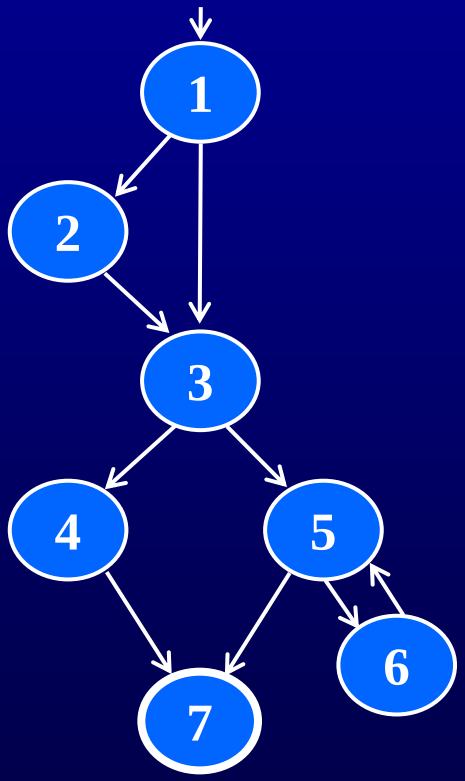
TR = { [1,2,3], [1,2,2], [2,2,3], [2,2,2] }

PPC Requirements :

TR = { [1,2,3], [2,2] }

Prime Path Example

- The previous example has 38 simple paths
- Only nine *prime paths*



Prime Paths

[1, 2, 3, 4, 7]
[1, 2, 3, 5, 7]
[1, 2, 3, 5, 6]
[1, 3, 4, 7]
[1, 3, 5, 7]
[1, 3, 5, 6]
[6, 5, 7]
[6, 5, 6]
[5, 6, 5]

Execute loop 0 times

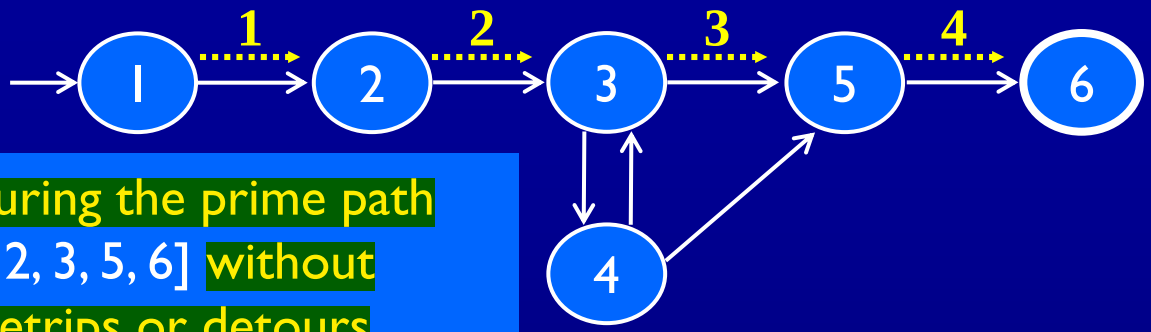
Execute loop once

Execute loop more than once

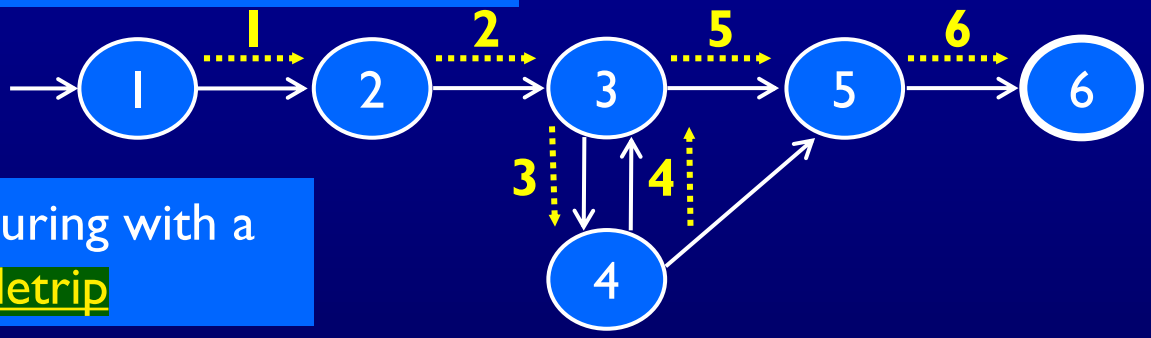
Touring, Sidetrips, and Detours

- Prime paths do not have internal loops … test paths might (page 170)
- Tour : *A test path p tours subpath q if q is a subpath of p*
- Tour With Sidetrips : *A test path p tours subpath q with sidetrips iff every edge in q is also in p in the same order*
  - The tour can include a sidetrip, as long as it comes back to the same node
- Tour With Detours : *A test path p tours subpath q with detours iff every node in q is also in p in the same order*
  - The tour can include a detour from node *ni*, as long as it comes back to the prime path at a successor of *ni*

Sidetrips and Detours Example

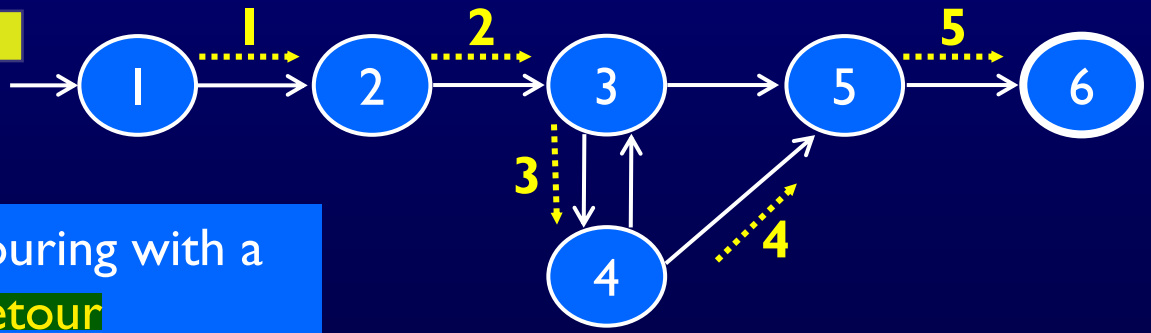


Touring the prime path [1, 2, 3, 5, 6] without sidetrips or detours



Touring with a sidetrip

حفظ ترتيب يال ها



Touring with a detour

Infeasible Test Requirements

- An infeasible test requirement cannot be satisfied
  - Unreachable statement (dead code)
  - Subpath that can only be executed with a contradiction ($X > 0$ and $X < 0$)
- Most test criteria have some infeasible test requirements
- It is usually undecidable whether all test requirements are feasible
- When sidetrips are not allowed, many structural criteria have more infeasible test requirements
- However, always allowing sidetrips weakens the test criteria

Practical recommendation—Best Effort Touring

- **Satisfy as many test requirements as possible without sidetrips**
- **Allow sidetrips to try to satisfy remaining test requirements**

Round Trips

- Round-Trip Path : *A prime path that starts and ends at the same node*

Simple Round Trip Coverage (SRTC) : TR contains at least one round-trip path for each reachable node in G that begins and ends a round-trip path.

Complete Round Trip Coverage (CRTC) : TR contains all round-trip paths for each reachable node in G.

- These criteria omit nodes and edges that are not in round trips
- Thus, they do not subsume edge-pair, edge, or node coverage

Simple & Prime Path Example

همه ی نودها در گراف

همه ی یال ها

Simple paths

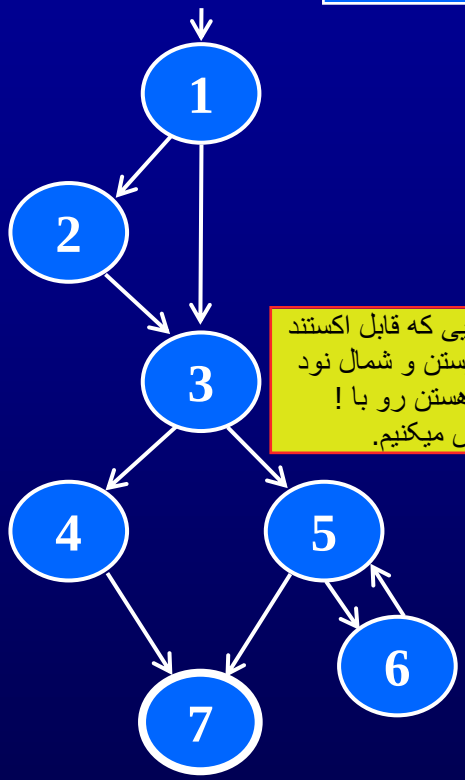
Len 0
[1]
[2]
[3]
[4]
[5]
[6]
[7] !

Len 1
[1, 2]
[1, 3]
[2, 3]
[3, 4]
[3, 5]
[4, 7] !
[5, 7] !
[5, 6]
[6, 5]

Len 2
[1, 2, 3]
[1, 3, 4]
[1, 3, 5]
[2, 3, 4]
[2, 3, 5]
[3, 4, 7] !
[3, 5, 7] !
[3, 5, 6] !
[5, 6, 5] *
[6, 5, 7] !
[6, 5, 6] *

Len 3
[1, 2, 3, 4]
[1, 2, 3, 5]
[1, 3, 4, 7] !
[1, 3, 5, 7] !
[1, 3, 5, 6] !
[2, 3, 4, 7] !
[2, 3, 5, 6] !
[2, 3, 5, 7] !

Len 4
[1, 2, 3, 4, 7] !
[1, 2, 3, 5, 7] !
[1, 2, 3, 5, 6] !



سیر هایی که قابل اکستند شدن نیستن و شمال نود فاینال هستن رو با ! مشخص میکنیم.

'!' means path terminates

'*' means path cycles

این چندتا چون زیرمسیر یه مسیر دیگه هستن نمیتونن prime path باشن.

حلقه ها هم قابل توسعه نیستند چون اگه یه نود دیگه اضافه کنیم مسیردیگه ساده نیست

این مسیر هم قابل توسعه نیست چون بعد از نود۶ فقط میشه به نود۵ بریم که دیگه مسیرمون ساده نمیشه و شرط ساده بودن نقض میشه

Prime Paths

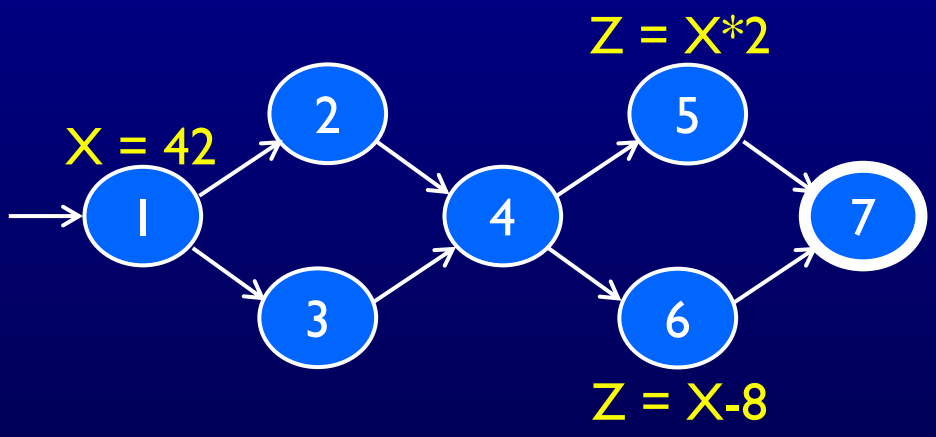
الگوریتم تعداد بی نهایت ایتریشن یا تکرار نداره چرا؟
چون ماکس طول مسیر ما مثلا اینجا ۴ است و یکی از تعداد نودهای گرافمون کمتره

برای پیداکردن prime pathها، از طولانی ترین مسیرها شروع میکنیم و این مسیرها نباید جزو مسیر دیگه ای باشند یعنی نباید sub-path مسیردیگه ای باشند پس اون مسیرهایی که قابل توسعه بودن یعنی sub-path یه مسیرساده بودن پس نمیتونن prime path باشند.

Data Flow Criteria

Goal: Try to ensure that values are computed and used correctly

- Definition (def) : A location where a value for a variable is stored into memory
- Use : A location where a variable's value is accessed



Defs: def (1) = {X}

def (5) = {Z}

def (6) = {Z}

Uses: use (5) = {X}

use (6) = {X}

The values given in defs should reach at least one, some, or all possible uses

DU Pairs and DU Paths

- def (n) or def (e) : The set of variables that are defined by node n or edge e
- use (n) or use (e) : The set of variables that are used by node n or edge e

- DU pair : A pair of locations (l_i, l_j) such that a variable v is defined at l_i and used at l_j

- Def-clear : A path from l_i to l_j is *def-clear* with respect to variable v if v is not given another value on any of the nodes or edges in the path
- Reach : If there is a def-clear path from l_i to l_j with respect to v, the def of v at l_i reaches the use at l_j

- du-path : A simple subpath that is def-clear with respect to v from a def of v to a use of v
- du (n_i, n_j, v) – the set of du-paths from n_i to n_j
- du (n_i, v) – the set of du-paths that start at n_i

Touring DU-Paths

- A test path *p* *du-tours* subpath *d* with respect to *v* if *p* tours *d* and the subpath taken is def-clear with respect to *v*
- Sidetrips can be used, just as with previous touring
- Three criteria
  - Use every def
  - Get to every use
  - Follow all du-paths

Data Flow Test Criteria

- First, we make sure every def reaches a use

All-defs coverage (ADC) : For each set of du-paths $S = du(n, v)$, TR contains at least one path d in S.

- Then we make sure that every def reaches all possible uses

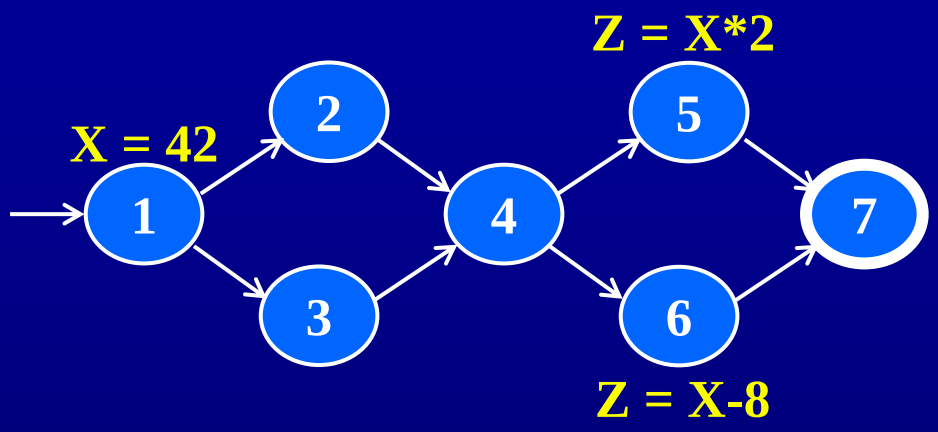
All-uses coverage (AUC) : For each set of du-paths to uses $S = du(n_i, n_j, v)$, TR contains at least one path d in S.

- Finally, we cover all the paths between defs and uses

All-du-paths coverage (ADUPC) : For each set $S = du(ni, nj, v)$, TR contains every path d in S.

هر defای به هر useای که برای یک متغیر است دسترسی و مسیر داشته باشه

Data Flow Testing Example



All-defs for *X*
[1, 2, 4, 5]

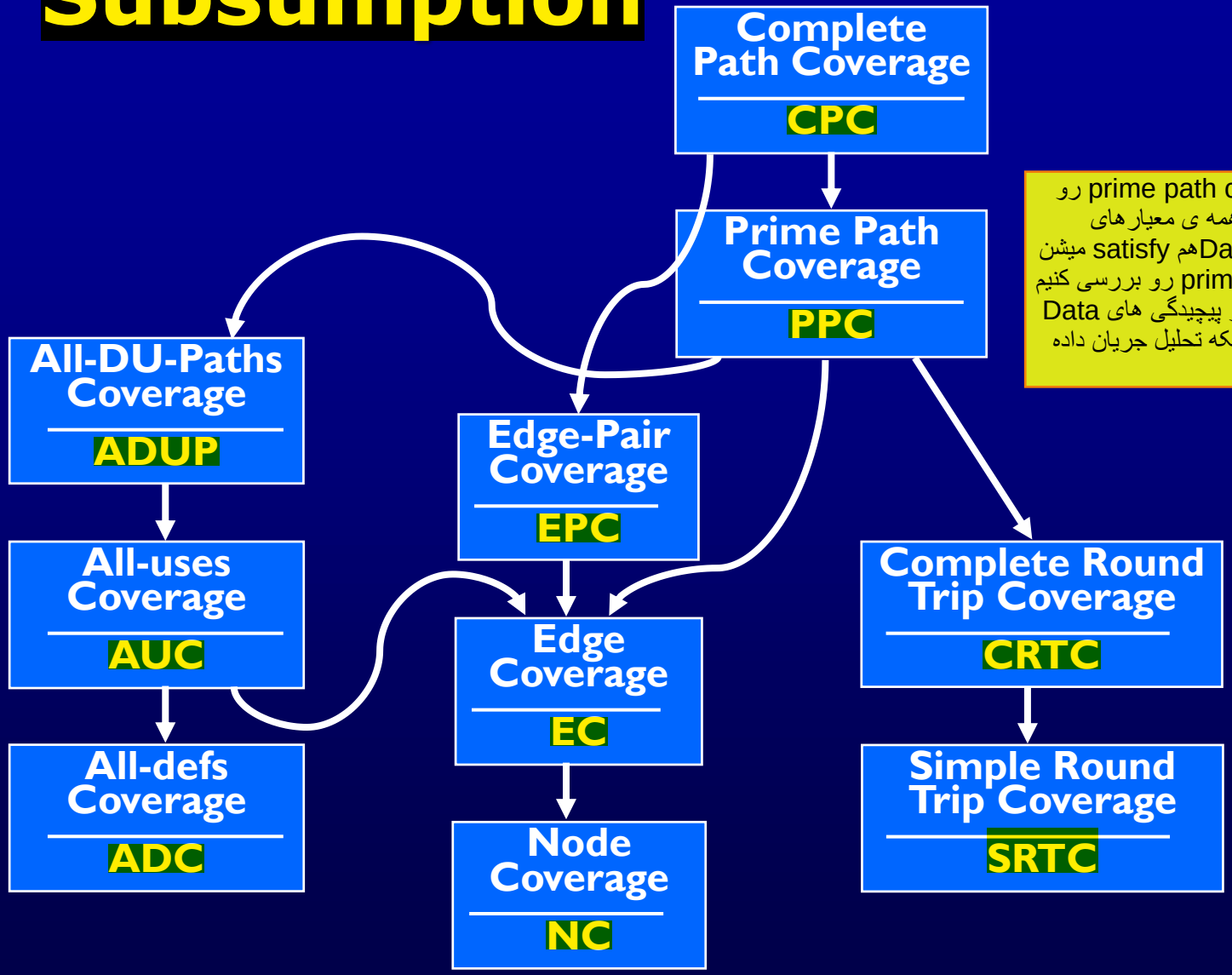
برای هر def حداقل یک use بررسی شه

All-uses for *X*
[1, 2, 4, 5]
[1, 2, 4, 6]

برای هر def همه ی use ها بررسی بشن

All-du-paths for *X*
[1, 2, 4, 5]
[1, 3, 4, 5]
[1, 2, 4, 6]
[1, 3, 4, 6]

Graph Coverage Criteria Subsumption



اگه ما prime path coverage رو برقرار کنیم دیگه همه ی معیارهای مربوط به Data flowهم satisfy میشن پس وقتی prime path رو بررسی کنیم میتونیم دیگه درگیر پیچیدگی های Data flow نشیم مگه اینکه تحلیل جریان داده مهم باشه

Summary 7.1-7.2

- Graphs are a very powerful abstraction for designing tests
- The various criteria allow lots of cost / benefit tradeoffs
- These two sections are entirely at the "design abstraction level" from chapter 2
- Graphs appear in many situations in software
  - As discussed in the rest of chapter 7

Introduction to Software Testing
Chapter 7.3
Graph Coverage for Source Code

Paul Ammann & Jeff Offutt

http://www.cs.gmu.edu/~offutt/softwaretest/

Update March 2016

Overview

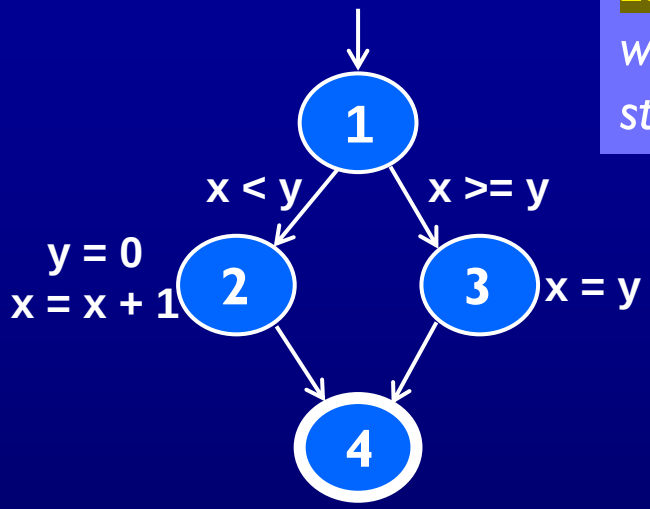
- A common application of graph criteria is to program source
- Graph : Usually the control flow graph (CFG)
- Node coverage : Execute every statement
- Edge coverage : Execute every branch
- Loops : Looping structures such as for loops, while loops, etc.
- Data flow coverage : Augment the CFG
  - defs are statements that assign values to variables
  - uses are statements that use variables

Control Flow Graphs

- A CFG models all executions of a method by describing control structures
- Nodes : Statements or sequences of statements (basic blocks)
- Edges : Transfers of control
- Basic Block : A sequence of statements such that if the first statement is executed, all statements will be (no branches)
- CFGs are sometimes annotated with extra information
  - branch predicates
  - defs
  - uses
- Rules for translating statements into graphs …

CFG : The if Statement

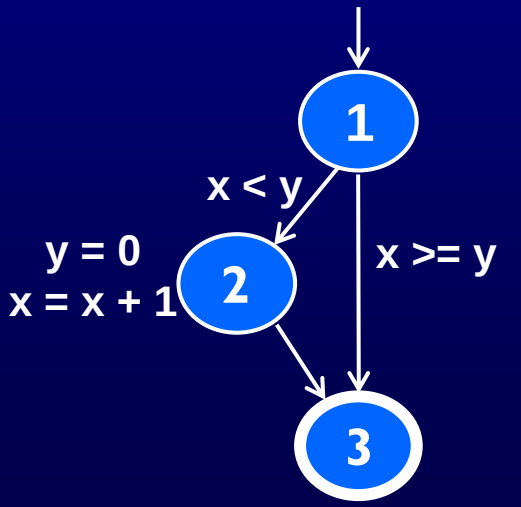
```
if (x < y)
{
   y = 0;
   x = x + 1;
}
else
{
   x = y;
}
```



Draw the graph. Label the edges with the Java statements.

```
if (x < y)
{
   y = 0;
   x = x + 1;
}
```

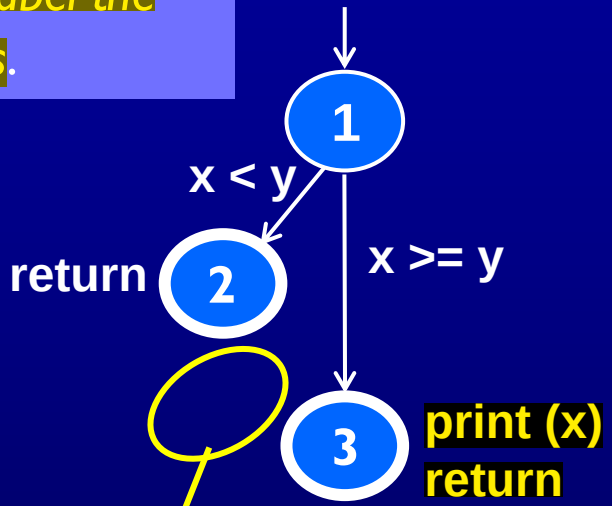
Draw the graph and label the edges.



CFG : The if-Return Statement

```
if (x < y)
{
   return;
}
print (x);
return;
```

Draw the graph and label the edges.



No edge from node 2 to 3.
The return nodes must be distinct.

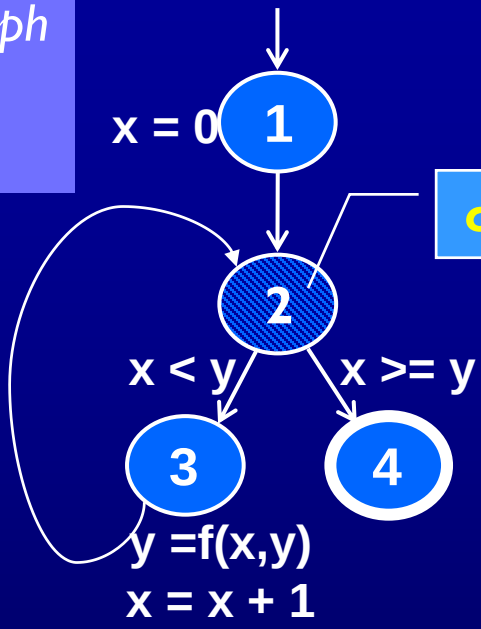
Loops

- Loops require "*extra*" nodes to be added
- Nodes that do not represent statements or basic blocks

CFG : while and for Loops

Draw the graph and label the edges.

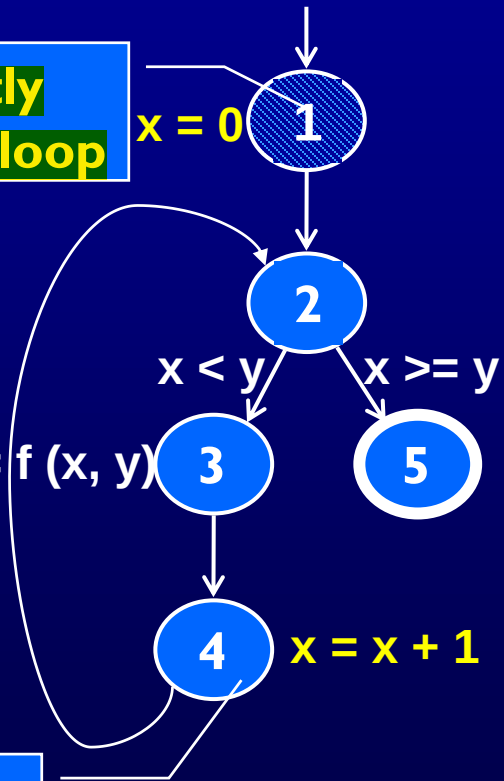
```
x = 0;
while (x < y)
{
   y = f (x, y);
   x = x + 1;
}
return (x);
```



dummy node

Draw the graph and label the edges.

implicitly initializes loop



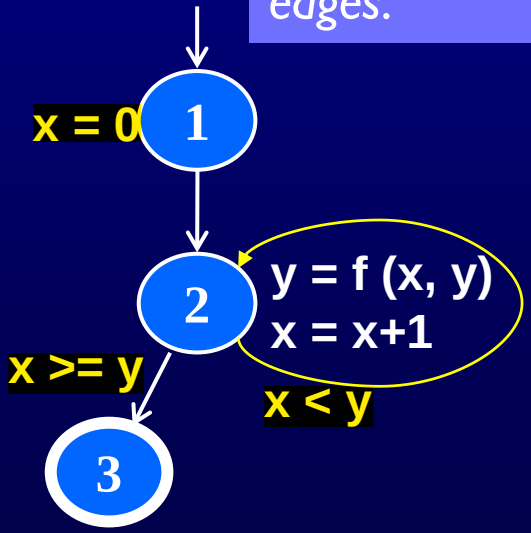

```
for (x = 0; x < y; x++)
{
   y = f (x, y);
}
return (x);
```

implicitly increments loop

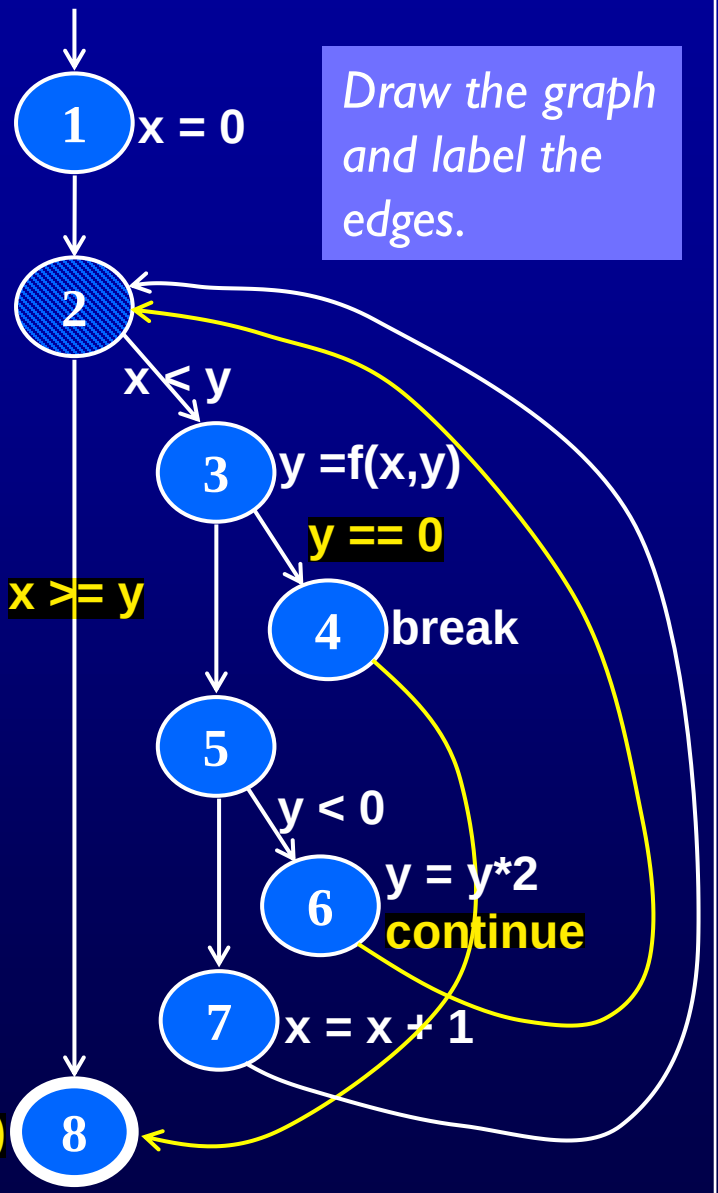
CFG : do Loop, break and continue

```
x = 0;
do
{
   y = f (x, y);
   x = x + 1;
} while (x < y);
return (y);
```

Draw the graph and label the edges.




```
x = 0;
while (x < y)
{
   y = f (x, y);
   if (y == 0)
   {
      break;
   } else if (y < 0)
   {
      y = y*2;
      continue;
   }
 else
      x = x + 1;
}
return (y);
```

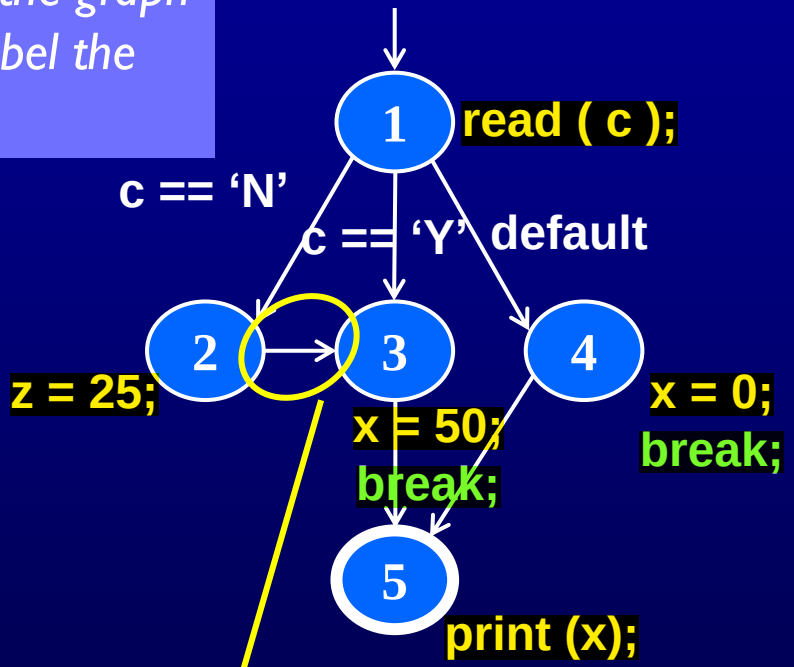


Draw the graph and label the edges.

CFG : The case (switch) Structure

```
read ( c) ;
switch ( c )
{
   case 'N':
      z = 25;
   case 'Y':
      x = 50;
      break;
   default:
      x = 0;
      break;
}
print (x);
```

Draw the graph and label the edges.

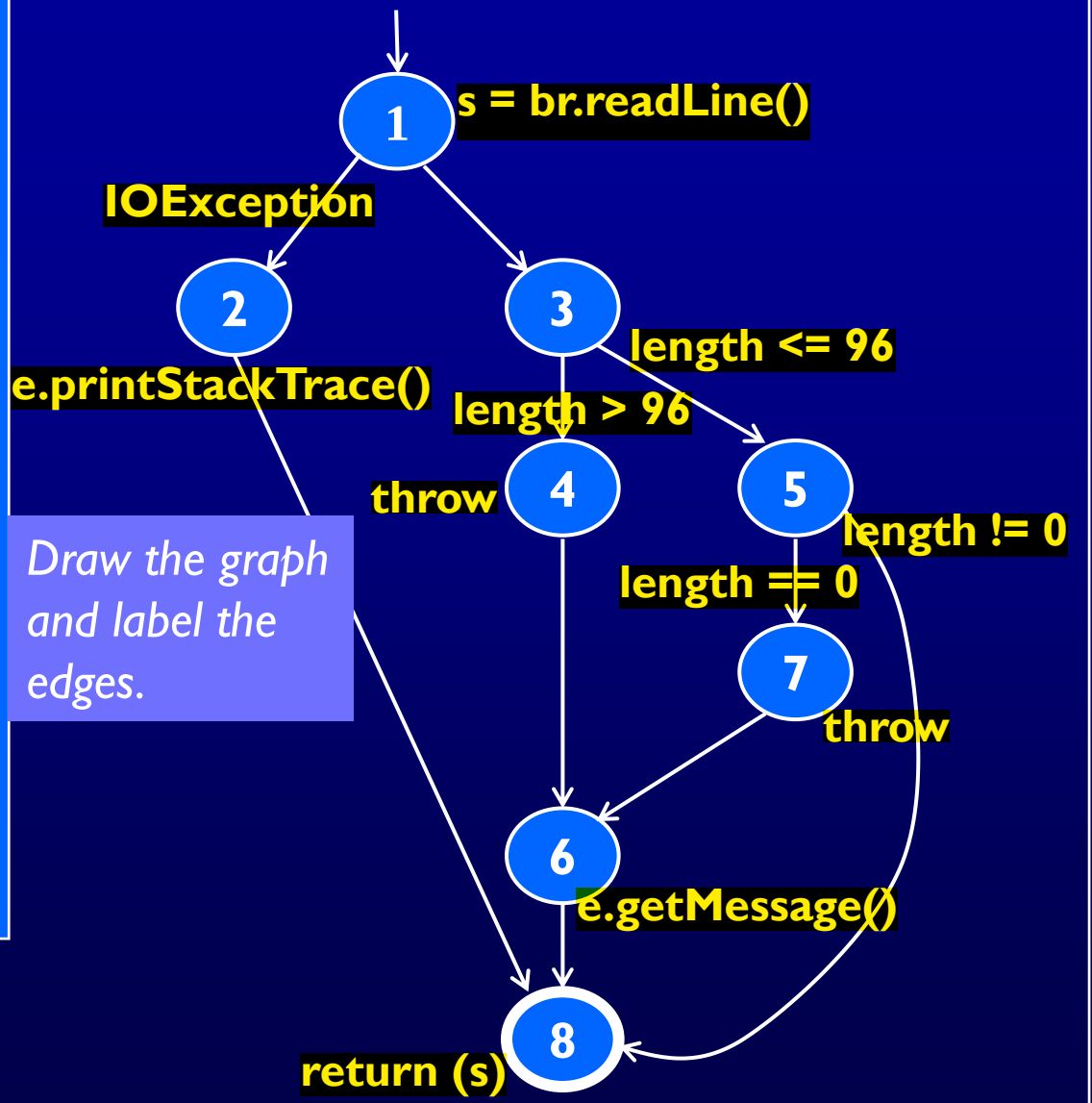


Cases without breaks fall through to the next case

CFG : Exceptions (try-catch)

```
try
{
   s = br.readLine();
   if (s.length() > 96)
      throw new Exception
         ("too long");
   if (s.length() == 0)
      throw new Exception
         ("too short");
} (catch IOException e) {
   e.printStackTrace();
} (catch Exception e) {
   e.getMessage();
}
return (s);
```

Draw the graph and label the edges.



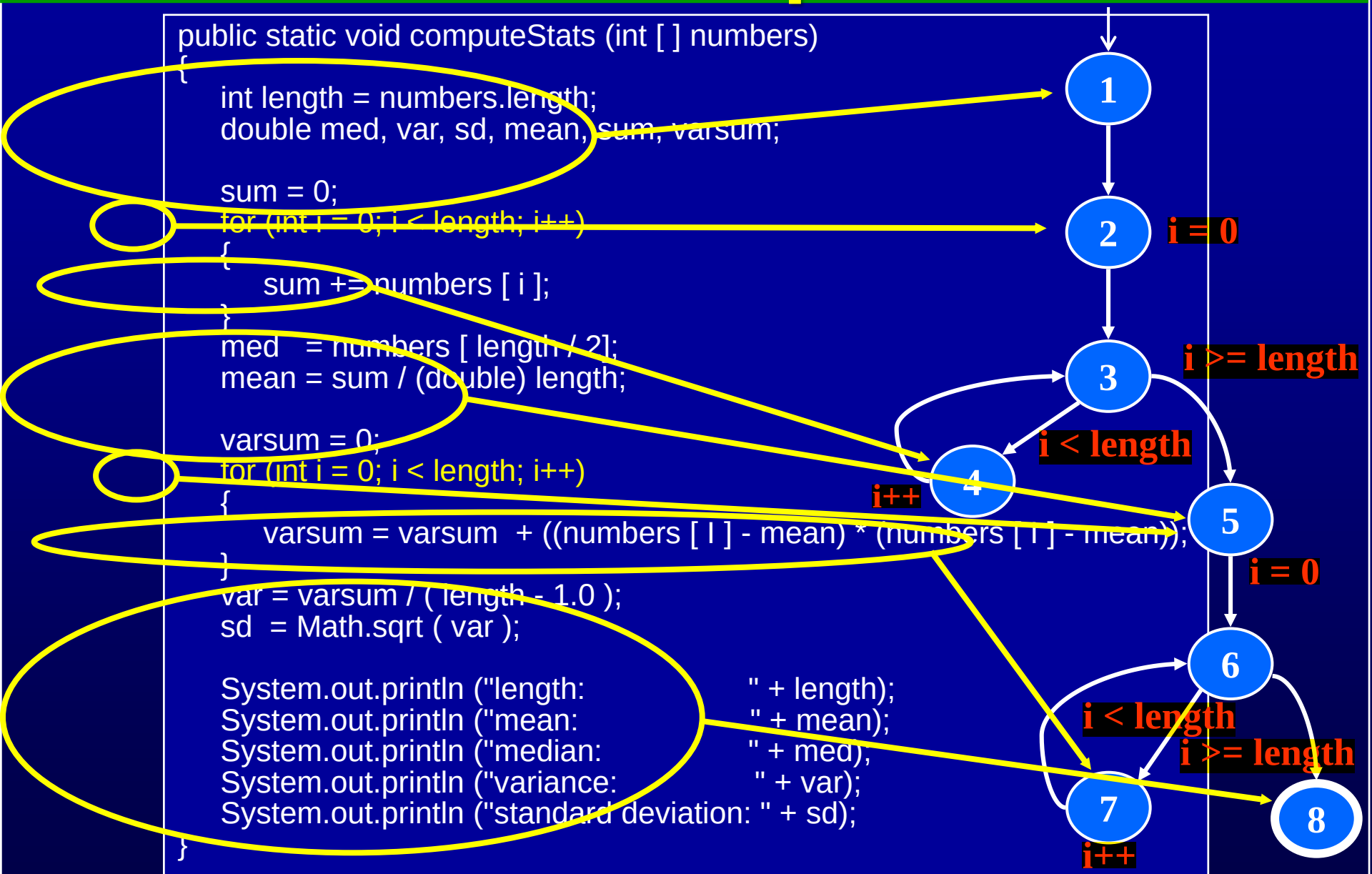
Example Control Flow – Stats

```
public static void computeStats (int [ ] numbers)
{
    int length = numbers.length;
    double med, var, sd, mean, sum, varsum;

    sum = 0;
    for (int i = 0; i < length; i++)
    {
        sum += numbers [ i ];
    }
    med   = numbers [ length / 2];
    mean = sum / (double) length;

    varsum = 0;
    for (int i = 0; i < length; i++)
    {
        varsum = varsum  + ((numbers [ i ] - mean) * (numbers [ i ] - mean));
    }
    var = varsum / ( length - 1.0 );
    sd  = Math.sqrt ( var );

    System.out.println ("length:                " + length);
    System.out.println ("mean:                  " + mean);
    System.out.println ("median:                " + med);
    System.out.println ("variance:               " + var);
    System.out.println ("standard deviation: " + sd);
}
```

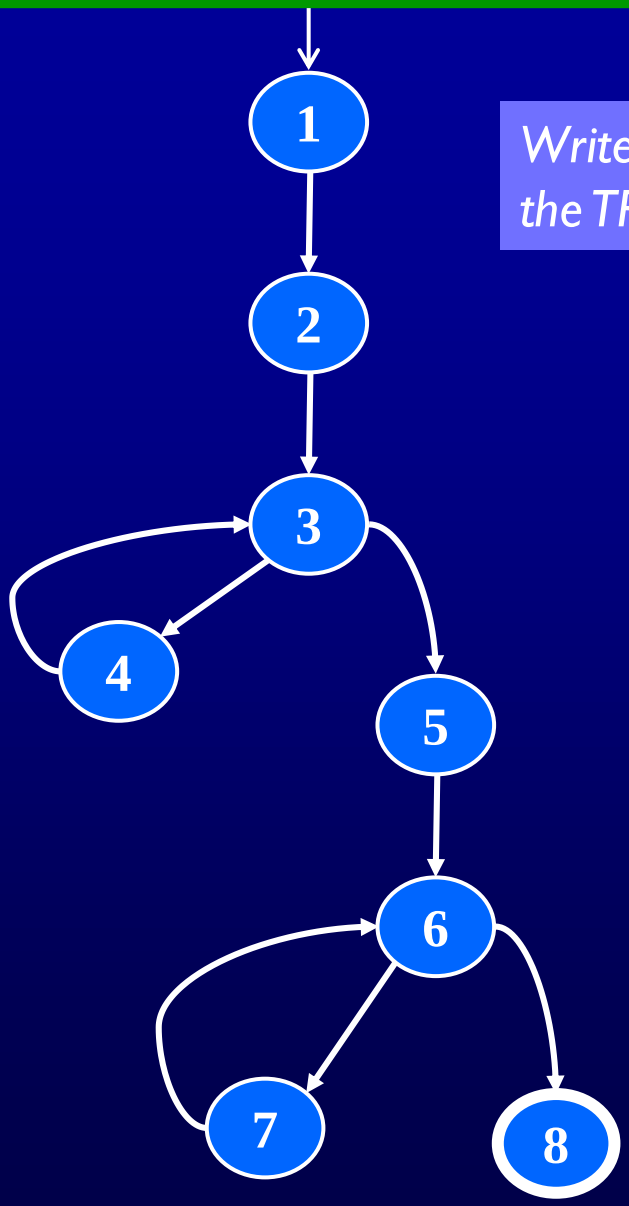
Draw the graph and label the edges.

Control Flow Graph for Stats

```
public static void computeStats (int [ ] numbers)
{
    int length = numbers.length;
    double med, var, sd, mean, sum, varsum;

    sum = 0;
    for (int i = 0; i < length; i++)
    {
        sum += numbers [ i ];
    }
    med   = numbers [ length / 2];
    mean = sum / (double) length;

    varsum = 0;
    for (int i = 0; i < length; i++)
    {
        varsum = varsum  + ((numbers [ I ] - mean) * (numbers [ I ] - mean));
    }
    var = varsum / ( length - 1.0 );
    sd  = Math.sqrt ( var );

    System.out.println ("length:                " + length);
    System.out.println ("mean:                  " + mean);
    System.out.println ("median:                " + med);
    System.out.println ("variance:              " + var);
    System.out.println ("standard deviation: " + sd);
}
```



Control Flow TRs and Test Paths—EC



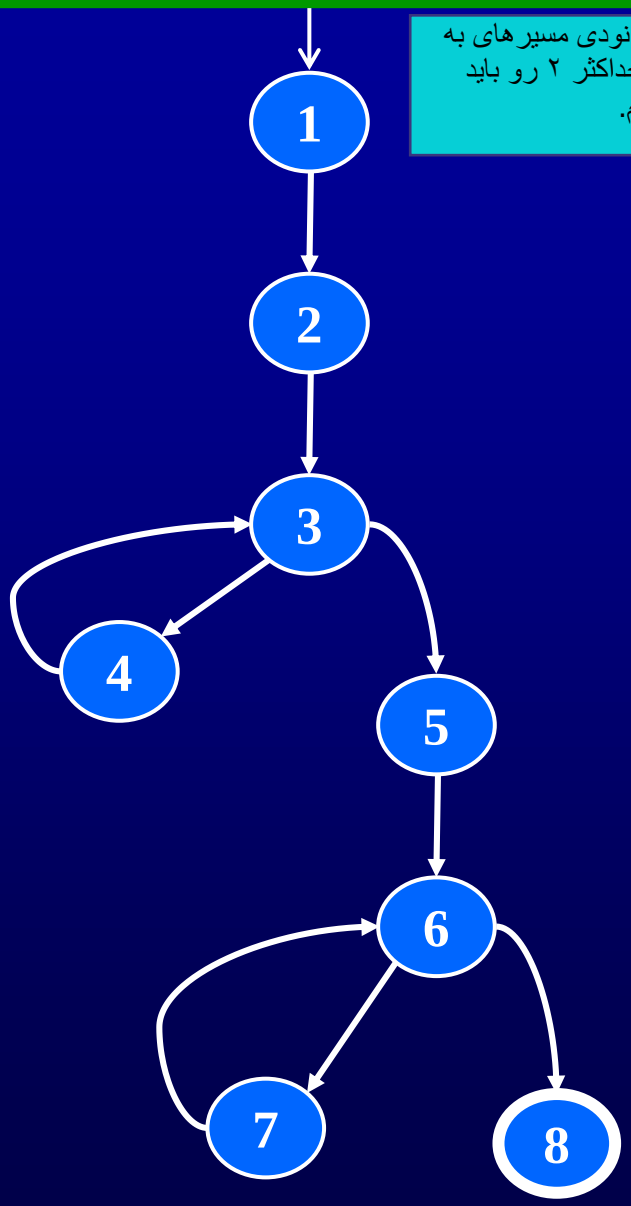
Write down the TRs for EC.

Edge Coverage

TR	Test Path
A. [1, 2]	[1, 2, 3, 4, 3, 5, 6, 7, 6, 8]
B. [2, 3]	
C. [3, 4]	
D. [3, 5]	
E. [4, 3]	
F. [5, 6]	
G. [6, 7]	
H. [6, 8]	
I. [7, 6]	

Write down test paths that tour all edges.

Control Flow TRs and Test Paths—EPC



از هر نودی مسیرهای به طول حداکثر ۲ رو باید بنویسیم.

Edge-Pair Coverage

TR

A. [1, 2, 3]
B. [2, 3, 4]
C. [2, 3, 5]
D. [3, 4, 3]
E. [3, 5, 6]
F. [4, 3, 5]
G. [5, 6, 7]
H. [5, 6, 8]
I. [6, 7, 6]
J. [7, 6, 8]
K. [4, 3, 4]
L. [7, 6, 7]

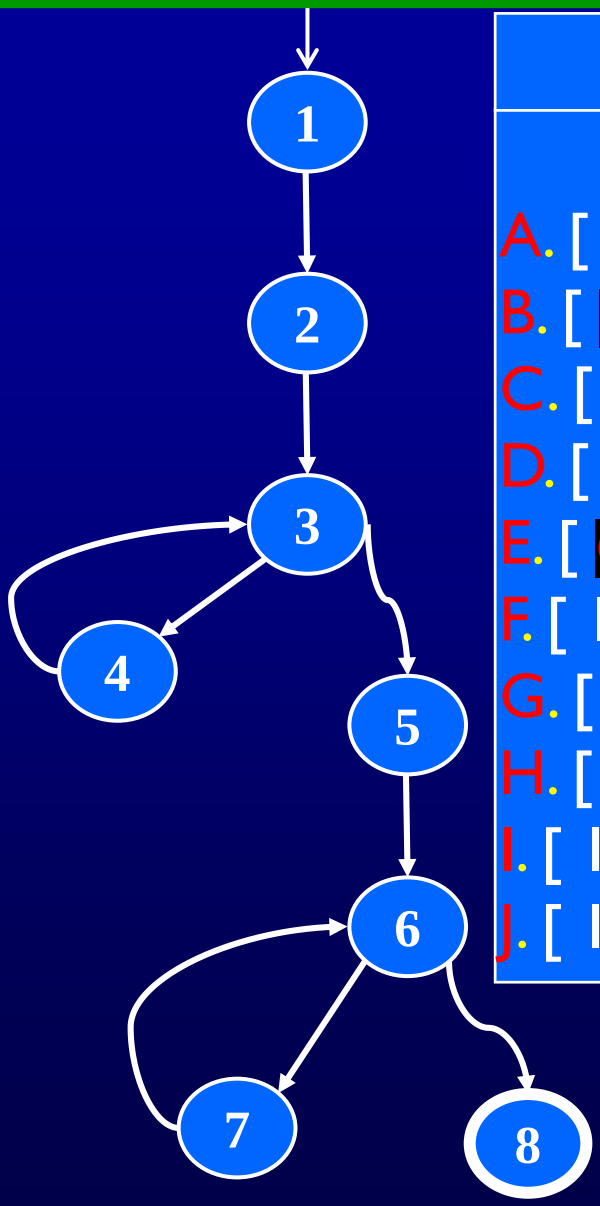
Test Paths

i. [1, 2, 3, 4, 3, 5, 6, 7, 6, 8]
ii. [1, 2, 3, 5, 6, 8]
iii. [1, 2, 3, 4, 3, 4, 3, 5, 6, 7, 6, 7, 6, 8]

TP	TRs toured	*sidetrips*
~~i~~	~~A, B, D, E, F, G, I, J~~	~~C, H~~
ii	A, C, E, H	
iii	A, B, D, E, F, G, I, J, K, L	C, H

TP iii makes TP i redundant. A *minimal* set of TPs is cheaper.

Control Flow TRs and Test Paths—PPC



Prime Path Coverage

TR

A. [3, 4, 3]
B. [4, 3, 4]
C. [7, 6, 7]
D. [7, 6, 8]
E. [6, 7, 6]
F. [1, 2, 3, 4]
G. [4, 3, 5, 6, 7]
H. [4, 3, 5, 6, 8]
I. [1, 2, 3, 5, 6, 7]
J. [1, 2, 3, 5, 6, 8]

Test Paths

i. [1, 2, 3, 4, 3, 5, 6, 7, 6, 8]
ii. [1, 2, 3, 4, 3, 4, 3, 5, 6, 7, 6, 7, 6, 8]
iii. [1, 2, 3, 4, 3, 5, 6, 8]
iv. [1, 2, 3, 5, 6, 7, 6, 8]
v. [1, 2, 3, 5, 6, 8]

TP	TRs toured	*sidetrips*
~~i~~	~~A, D, E, F, G~~	~~H, I, J~~
ii	A, B, C, D, E, F, G,	H, I, J
iii	A, F, H	J
iv	D, E, F, I	J
v	J	

TP ii makes TP i redundant.

Data Flow Coverage for Source

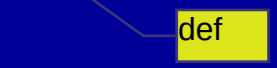
- def : a location where a value is stored into memory
  - x appears on the left side of an assignment (x = 44;)
  - x is an actual parameter in a call and the method changes its value
  - x is a formal parameter of a method (implicit def when method starts)
  - x is an input to a program
- use : a location where variable's value is accessed
  - x appears on the right side of an assignment
  - x appears in a conditional test
  - x is an actual parameter to a method
  - x is an output of the program
  - x is an output of a method in a return statement

یه پارامتری را به متدی پاس بدیم

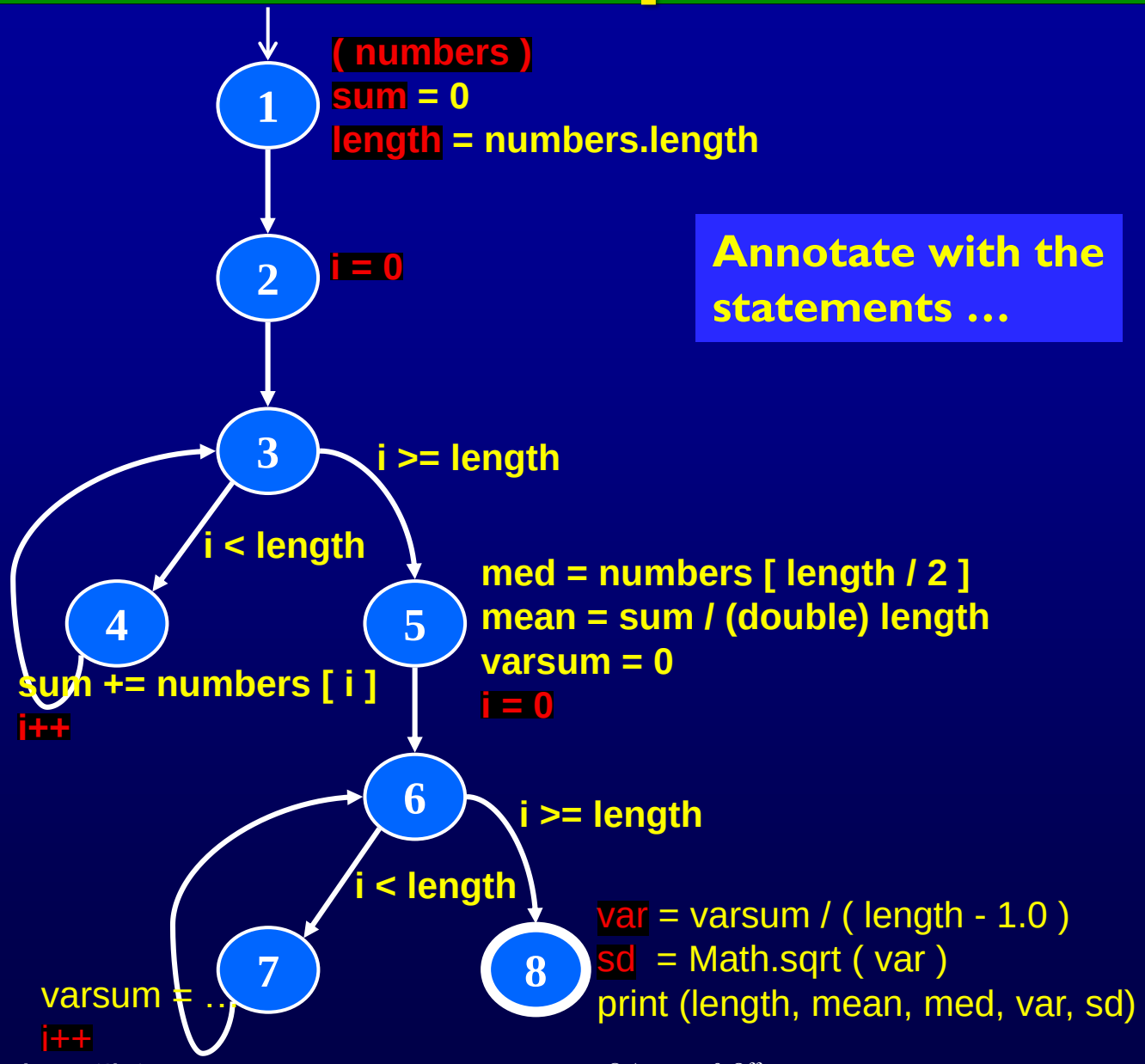
- If a def and a use appear on the same node, then it is only a DU-pair if the def occurs after the use and the node is in a loop

Example Data Flow – Stats

```
public static void computeStats (int [ ] numbers)
{
    int length = numbers.length;
    double med, var, sd, mean, sum, varsum;

    sum = 0.0;
    for (int i = 0; i < length; i++)
    {
        sum += numbers [ i ];
    }
    med   = numbers [ length / 2 ];
    mean = sum / (double) length;

    varsum = 0.o;
    for (int i = 0; i < length; i++)
    {
        varsum = varsum  + ((numbers [ i ] - mean) * (numbers [ i ] - mean));
    }
    var = varsum / ( length - 1 );
    sd  = Math.sqrt ( var );

    System.out.println ("length:                " + length);
    System.out.println ("mean:                  " + mean);
    System.out.println ("median:                " + med);
    System.out.println ("variance:               " + var);
    System.out.println ("standard deviation: " + sd);
}
```

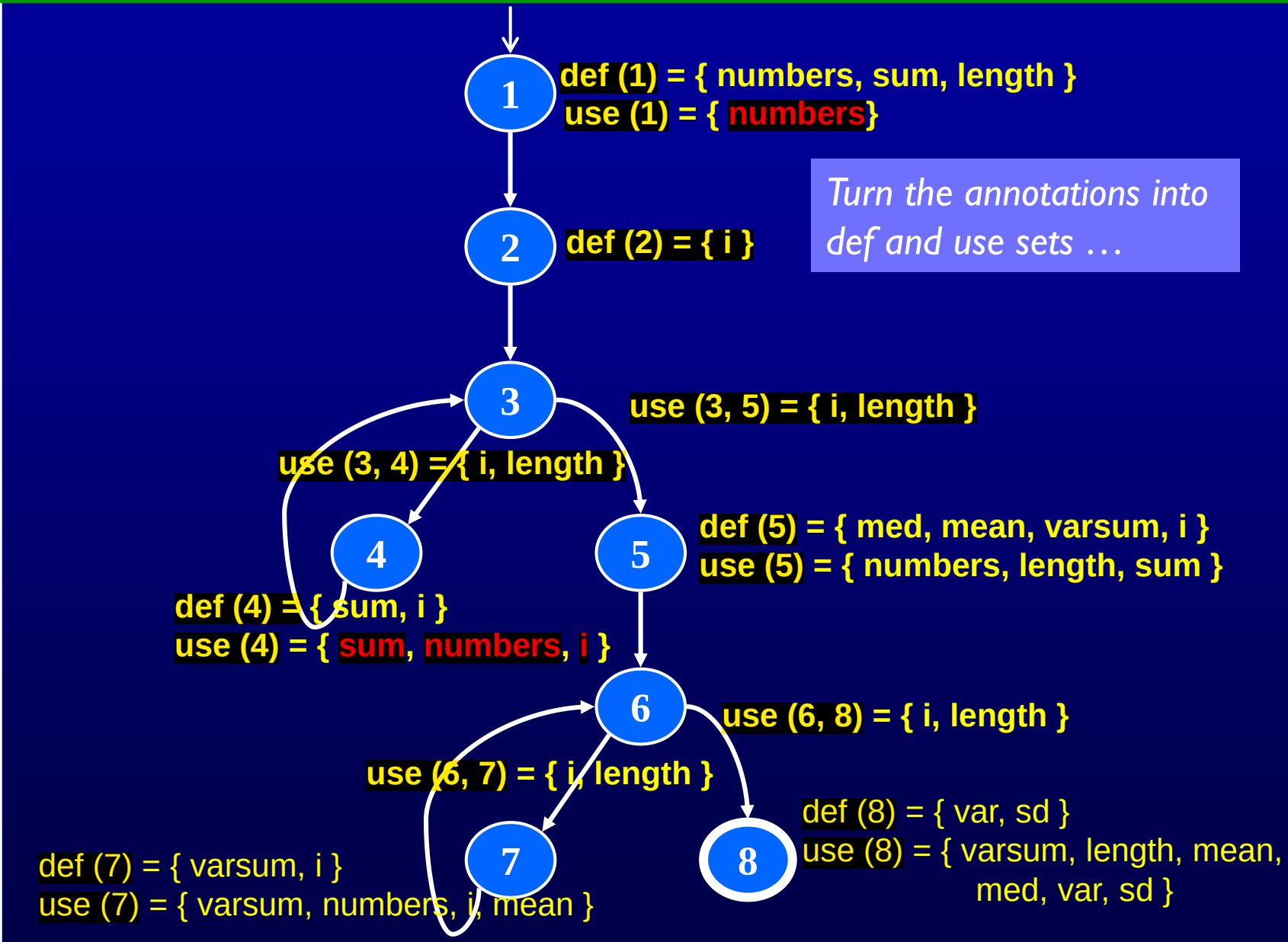


Control Flow Graph for Stats



Annotate with the statements …

CFG for Stats – With Defs & Uses



Defs and Uses Tables for Stats

Node	Def	Use
1	{ numbers, sum, length }	{ numbers }
2	{ i }	
3		
4	{ sum, i }	{ numbers, i, sum }
5	{ med, mean, varsum, i }	{ numbers, length, sum }
6		
7	{ varsum, i }	{ varsum, numbers, i, mean }
8	{ var, sd }	{ varsum, length, var, mean, med, var, sd }

Edge	Use
(1, 2)	
(2, 3)	
(3, 4)	{ i, length }
(4, 3)	
(3, 5)	{ i, length }
(5, 6)	
(6, 7)	{ i, length }
(7, 6)	
(6, 8)	{ i, length }

DU Pairs for Stats

variable	DU Pairs
numbers	(1, 4) (1, 5) (1, 7)
length	(1, 5) (1, 8) (1, (3,4)) (1, (3,5)) (1, (6,7)) (1, (6,8))
med	(5, 8)
var	(8, 8)
sd	(8, 8)
mean	(5, 7) (5, 8)
sum	(1, 4) (1, 5) (4, 4) (4, 5)
varsum	(5, 7) (5, 8) (7, 7) (7, 8)
i	(2, 4) (2, (3,4)) (2, (3,5)) (2, 7) (2, (6,7)) (2, (6,8)) (4, 4) (4, (3,4)) (4, (3,5)) (4, 7) (4, (6,7)) (4, (6,8)) (5, 7) (5, (6,7)) (5, (6,8)) (7, 7) (7, (6,7)) (7, (6,8))

defs come before uses, do not count as DU pairs

defs after use in loop, these are valid DU pairs

No def-clear path … different scope for i

A simple subpath that is def-clear with respect to v from a def of v to a use of v

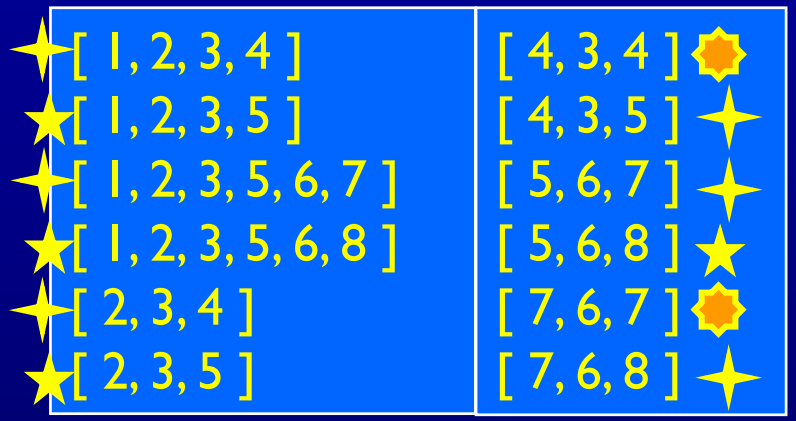
DU Paths for Stats

variable	DU Pairs	DU Paths
numbers	(1, 4) (1, 5) (1, 7)	[1, 2, 3, 4] [1, 2, 3, 5] [1, 2, 3, 5, 6, 7]
length	(1, 5) (1, 8) (1, (3,4)) (1, (3,5)) (1, (6,7)) (1, (6,8))	[1, 2, 3, 5] [1, 2, 3, 5, 6, 8] [1, 2, 3, 4] [1, 2, 3, 5] [1, 2, 3, 5, 6, 7] [1, 2, 3, 5, 6, 8]
med	(5, 8)	[5, 6, 8]
var	(8, 8)	*No path needed*
sd	(8, 8)	*No path needed*
sum	(1, 4) (1, 5) (4, 4) (4, 5)	[1, 2, 3, 4] [1, 2, 3, 5] [4, 3, 4] [4, 3, 5]

variable	DU Pairs	DU Paths
mean	(5, 7) (5, 8)	[5, 6, 7] [5, 6, 8]
varsum	(5, 7) (5, 8) (7, 7) (7, 8)	[5, 6, 7] [5, 6, 8] [7, 6, 7] [7, 6, 8]
i	(2, 4) (2, (3,4)) (2, (3,5)) (4, 4) (4, (3,4)) (4, (3,5)) (5, 7) (5, (6,7)) (5, (6,8)) (7, 7) (7, (6,7)) (7, (6,8))	[2, 3, 4] [2, 3, 4] [2, 3, 5] [4, 3, 4] [4, 3, 4] [4, 3, 5] [5, 6, 7] [5, 6, 7] [5, 6, 8] [7, 6, 7] [7, 6, 7] [7, 6, 8]

DU Paths for Stats—No Duplicates

There are 38 DU paths for Stats, but only 12 unique



4 expect a loop not to be "entered"

6 require at least one iteration of a loop

2 require at least two iterations of a loop

Test Cases and Test Paths

Test Case : numbers = (44) ; length = 1

Test Path : [1, 2, 3, 4, 3, 5, 6, 7, 6, 8]

Additional DU Paths covered (no sidetrips)

[1, 2, 3, 4] [2, 3, 4] [4, 3, 5] [5, 6, 7] [7, 6, 8]

The five stars ✦ that require at least one iteration of a loop

Test Case : numbers = (2, 10, 15) ; length = 3

Test Path : [1, 2, 3, 4, 3, 4, 3, 4, 3, 5, 6, 7, 6, 7, 6, 7, 6, 8]

DU Paths covered (no sidetrips)

[4, 3, 4] [7, 6, 7]

The two stars ✹ that require at least two iterations of a loop

Other DU paths ★ require arrays with length 0 to skip loops

But the method fails with index out of bounds exception…

```
med = numbers [length / 2];
```

A fault was found

Summary

- Applying the graph test criteria to control flow graphs is relatively straightforward
  - Most of the developmental research work was done with CFGs
- A few subtle decisions must be made to translate control structures into the graph
- Some tools will assign each statement to a unique node
  - These slides and the book uses basic blocks

Introduction to Software Testing
(*2nd edition*)
Chapter 7.4

Graph Coverage for Design Elements

Paul Ammann & Jeff Offutt

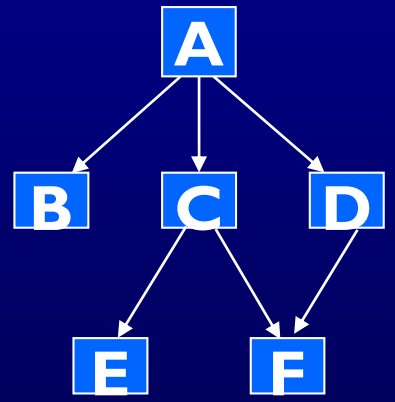
http://www.cs.gmu.edu/~offutt/softwaretest/

OO Software and Designs

- Emphasis on modularity and reuse puts complexity in the design connections
- Testing design relationships is more important than before
- Graphs are based on the connections among the software components
  - Connections are dependency relations, also called couplings

Call Graph

- The most common graph for structural design testing
- Nodes : Units (in Java – methods)
- Edges : Calls to units



Example call graph

Node coverage : call every unit at least once (method coverage)

Edge coverage : execute every call at least once (call coverage)

اون ترنزیشن یا کالینگه باید تحت تست قرار بگیره

باید حتما دوبار کال بشه از سمت c,D

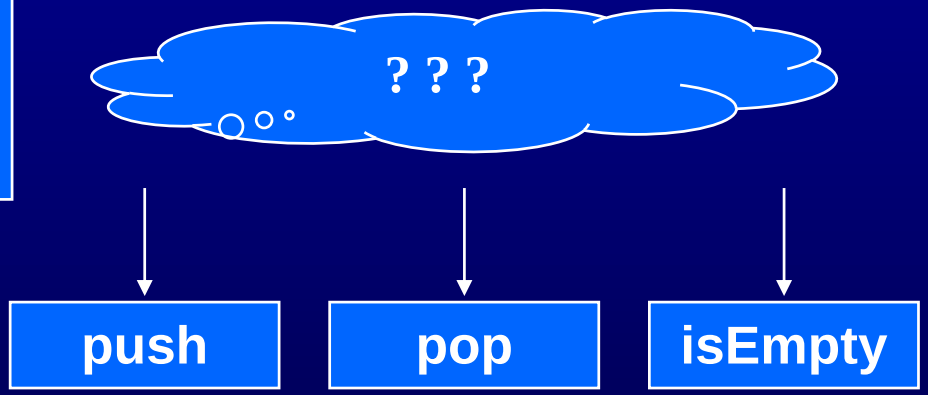
Call Graphs on Classes

- Node and edge coverage of class call graphs often do not work very well
- Individual methods might not call each other at all!

این نوع تست برای تست کردن متدهای این کلاس مناسب نیست چون همدیگر را صدا نمیزنند

Class stack

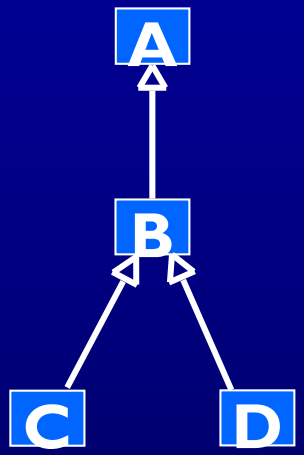
public void push (Object o)
public Object pop ( )
public boolean isEmpty (Object o)



Other types of testing are needed – do not use graph criteria

Inheritance & Polymorphism

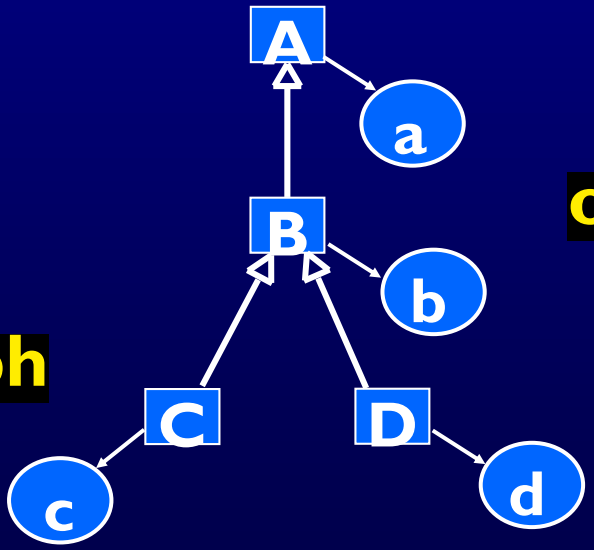
Caution : Ideas are preliminary and not widely used



Classes are not executable, so this graph is not directly testable

We need objects

Example inheritance hierarchy graph



objects

What is coverage on this graph ?

Coverage on Inheritance Graph

- Create an object for each class ?
  - This seems weak because there is no execution
- Create an object for each class and apply call coverage?

OO Call Coverage : TR contains each reachable node in the call graph of an object instantiated for each class in the class hierarchy.

OO Object Call Coverage : TR contains each reachable node in the call graph of every object instantiated for each class in the class hierarchy.

- Data flow is probably more appropriate …

Data Flow at the Design Level

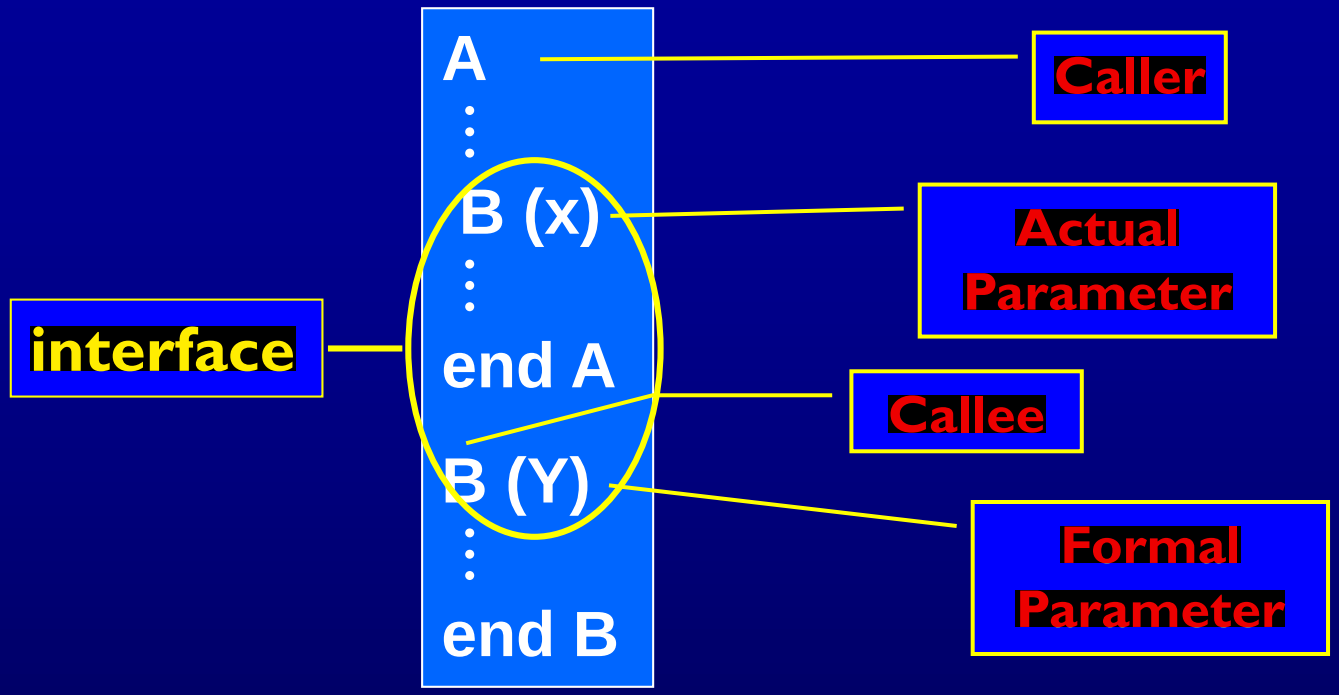
- Data flow couplings among units and classes are more complicated than control flow couplings
  - When values are passed, they "change names"
  - Many different ways to share data
  - Finding defs and uses can be difficult – finding which uses a def can reach is very difficult
- When software gets complicated … testers should get interested
  - That's where the faults are!

Preliminary Definitions

- Caller : A unit that invokes another unit
- Callee : The unit that is called
- Callsite : Statement or node where the call appears
- Actual parameter : Variable in the caller
- Formal parameter : Variable in the callee

متغیر هایی که به کالی پاس داده میشن

Example Call Site



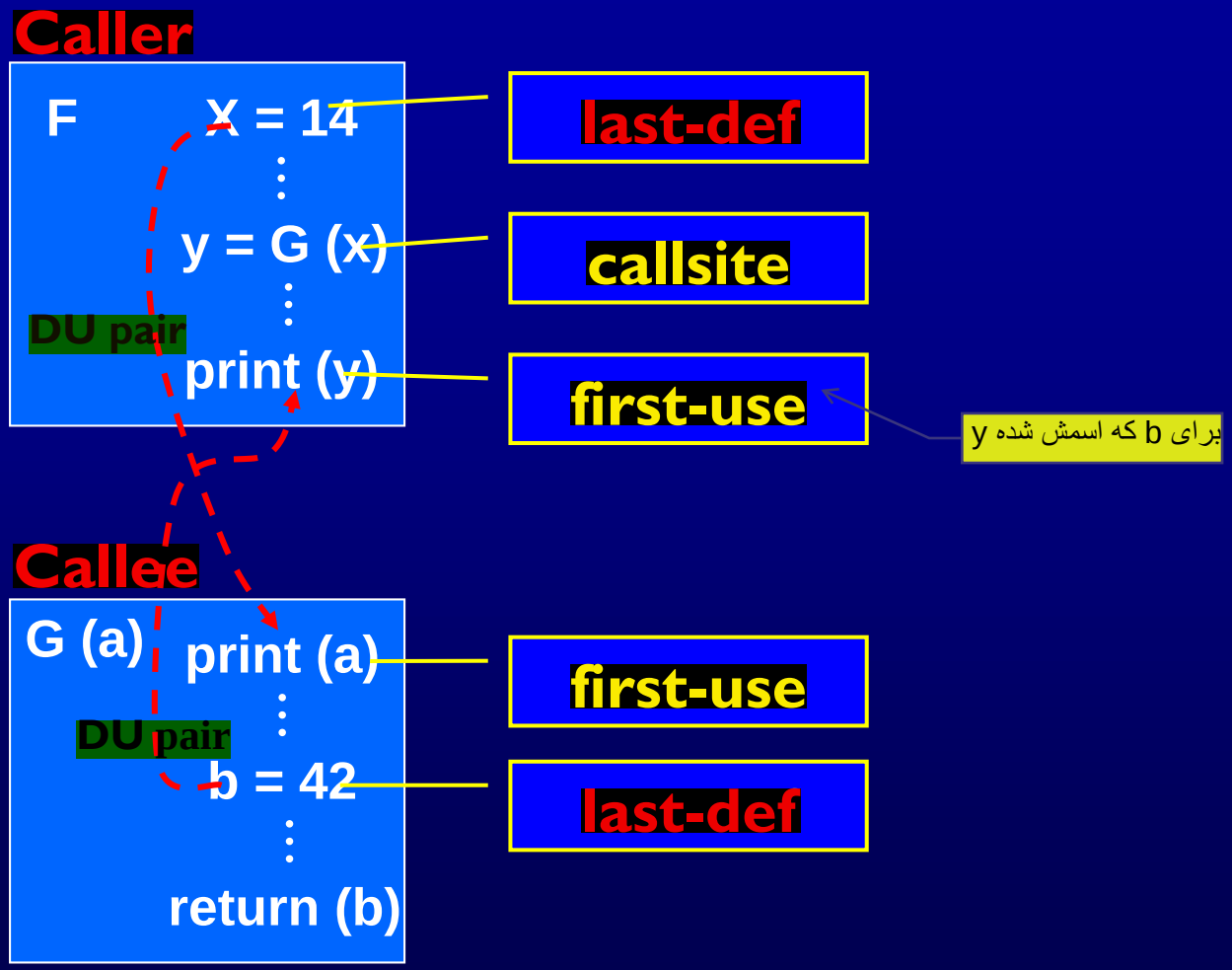
- Applying data flow criteria to def-use pairs between units is too expensive
- Too many possibilities
- But this is integration testing, and we really only care about the interface …

Inter-procedural DU Pairs

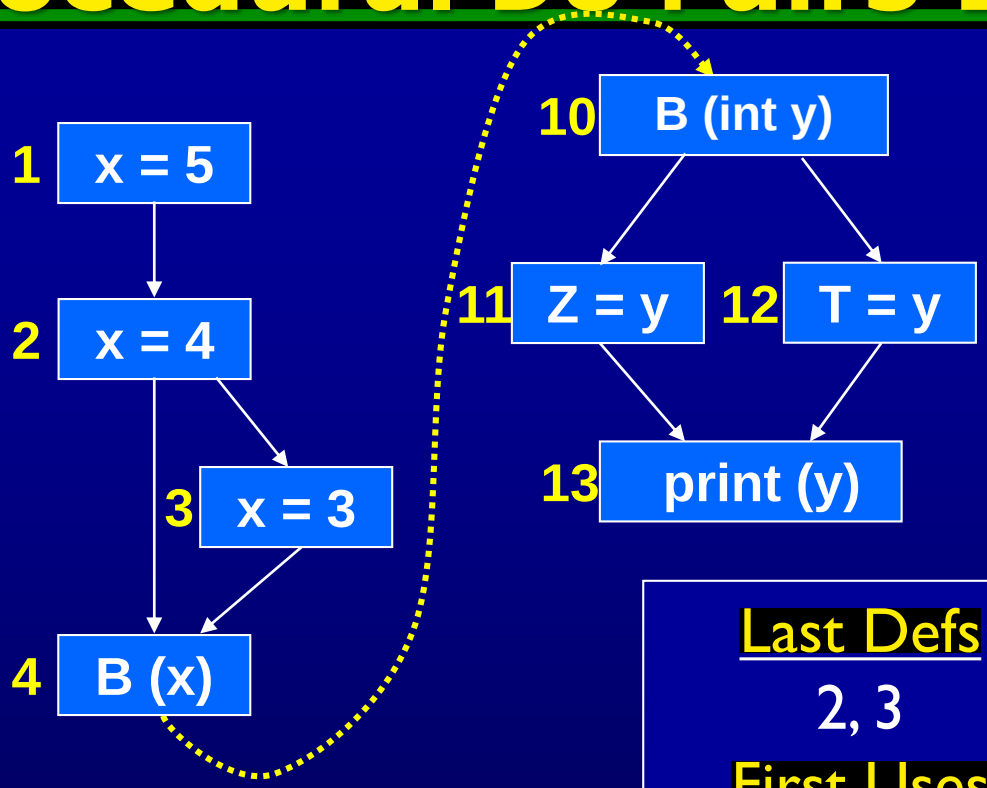
- If we focus on the interface, then we just need to consider the last definitions of variables before calls and returns and first uses inside units and after calls
- Last-def : The set of nodes that define a variable *x* and has a def-clear path from the node through a callsite to a use in the other unit
  - Can be from caller to callee (parameter or shared variable) or from callee to caller as a return value
- First-use : The set of nodes that have uses of a variable *y* and for which there is a def-clear and use-clear path from the callsite to the nodes.

A path from n_i to n_j is *use-clear* with respect to variable *v* if for every node n_k on the path, $k \neq i$ and $k \neq j$, *v* is not in $use(n_k)$.

Inter-procedural DU Pairs Example



Inter-procedural DU Pairs Example



Last Defs

2, 3

First Uses

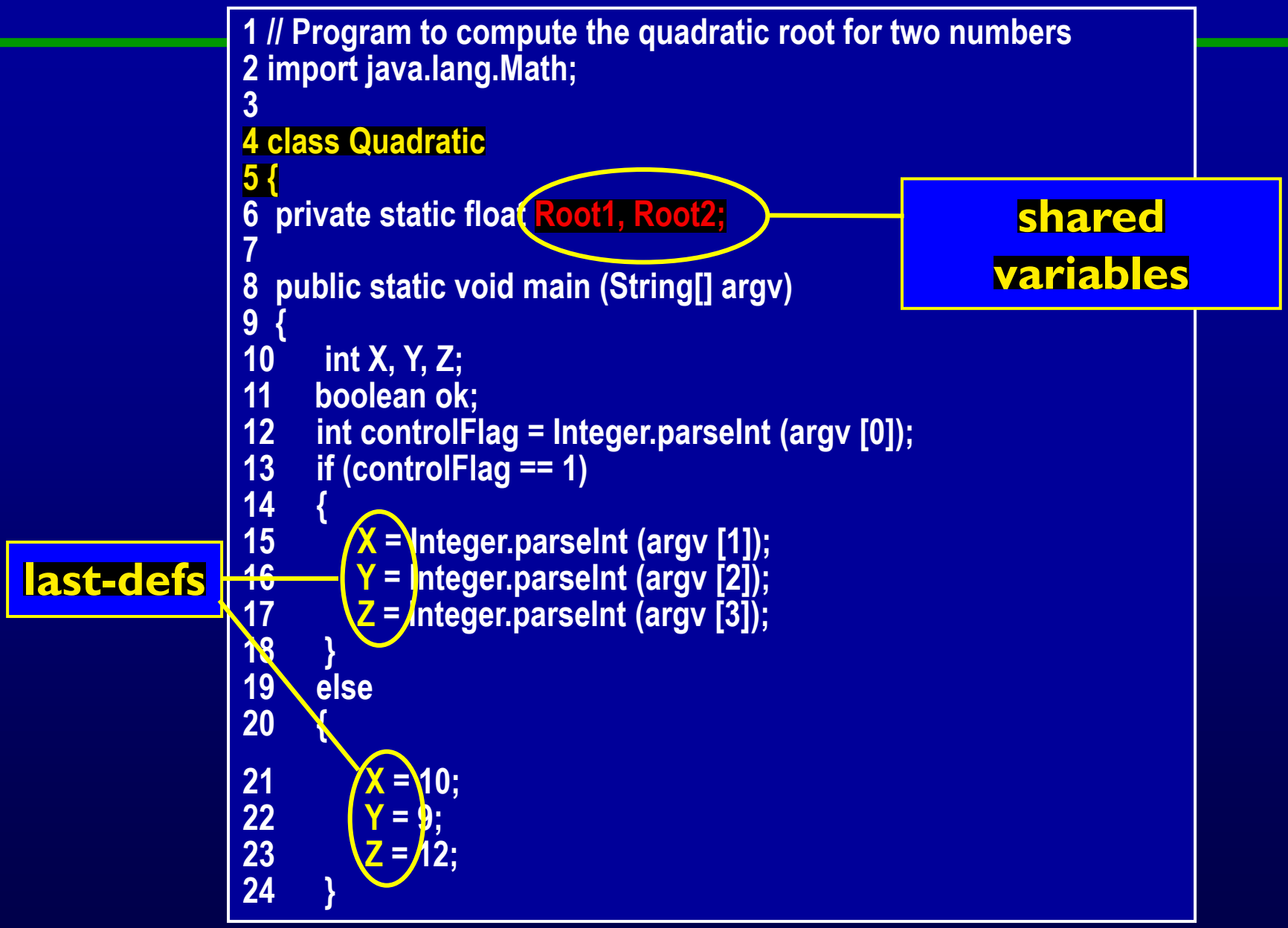
11, 12

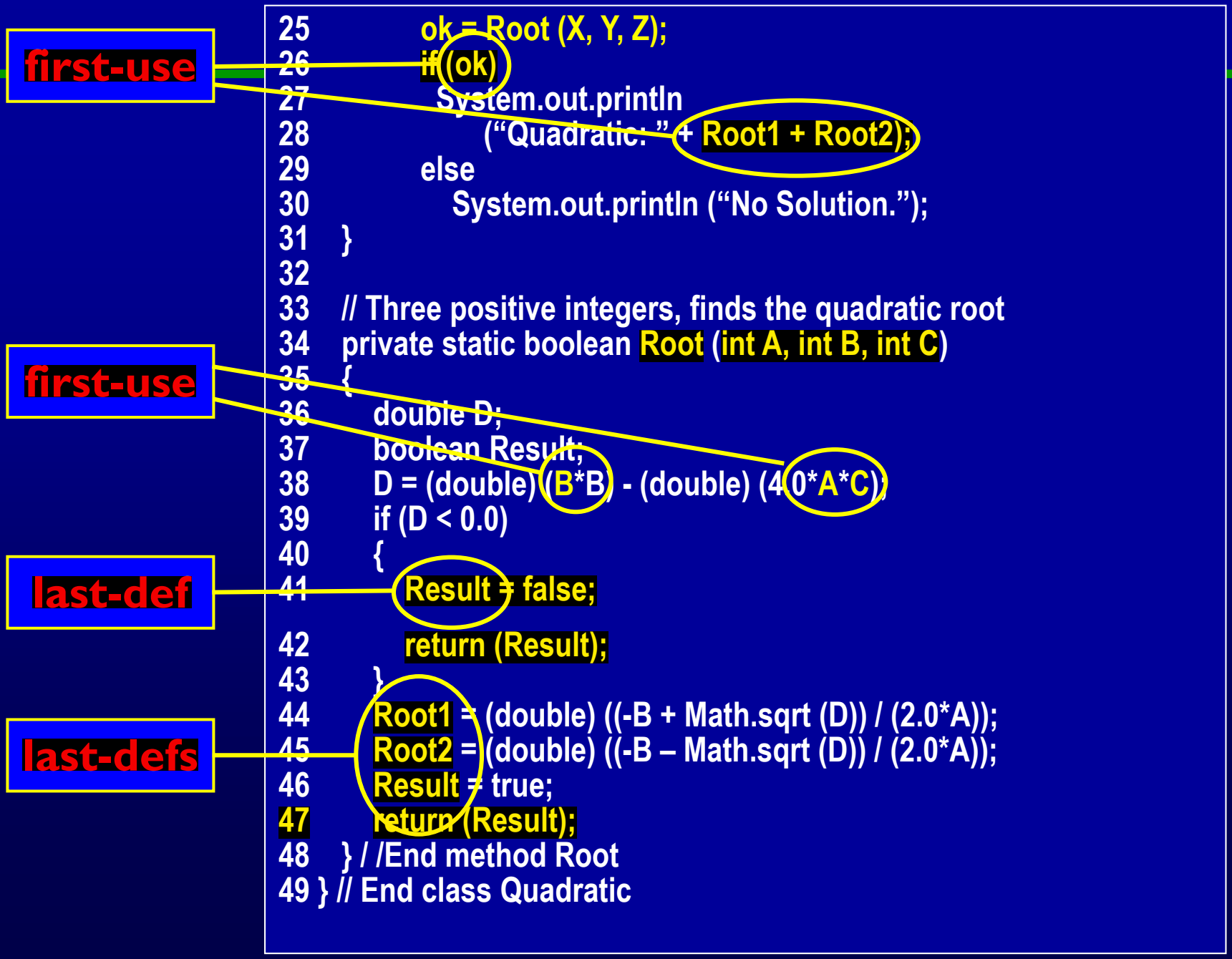
DU Pairs

(A, x, 2)—(B, y, 11)
(A, x, 2)—(B, y, 12)
(A, x, 3)—(B, y, 11)
(A, x, 3)—(B, y, 12)

Example – Quadratic

```
1 // Program to compute the quadratic root for two
numbers
2 import java.lang.Math;
3
4 class Quadratic
5 {
6  private static float Root1, Root2;
7
8  public static void main (String[] argv)
9  {
10    int X, Y, Z;
11    boolean ok;
12    int controlFlag = Integer.parseInt (argv[0]);
13    if (controlFlag == 1)
14    {
15       X = Integer.parseInt (argv[1]);
16       Y = Integer.parseInt (argv[2]);
17       Z = Integer.parseInt (argv[3]);
18    }
19    else
20    {

21       X = 10;
22       Y = 9;
23       Z = 12;
24    }
25       ok = Root (X, Y, Z);
26       if (ok)
27         System.out.println
28            ("Quadratic: " + Root1 + Root2);
29       else
30          System.out.println ("No Solution.");
31  }
32
33 // Three positive integers, finds quadratic root
34   private static boolean Root (int A, int B, int C)
35   {
36      double D;
37      boolean Result;
38      D = (double) (B*B) -  (double) (4.0*A*C );
39      if (D < 0.0)
40      {
41         Result = false;

42         return (Result);
43      }
44      Root1 = (double) ((-B + Math.sqrt(D))/(2.0*A));
45      Root2 = (double) ((-B – Math.sqrt(D))/(2.0*A));
46      Result = true;
47      return (Result);
48   } // End method Root
49 } // End class Quadratic
```


1 // Program to compute the quadratic root for two numbers
2 import java.lang.Math;
3
4 class Quadratic
5 {
6 private static float Root1, Root2;
7
8 public static void main (String[] argv)
9 {
10 int X, Y, Z;
11 boolean ok;
12 int controlFlag = Integer.parseInt (argv [0]);
13 if (controlFlag == 1)
14 {
15 X = Integer.parseInt (argv [1]);
16 Y = Integer.parseInt (argv [2]);
17 Z = Integer.parseInt (argv [3]);
18 }
19 else
20 {
21 X = 10;
22 Y = 9;
23 Z = 12;
24 }
shared variables
last-defs


first-use
first-use
last-def
last-defs
25 ok = Root (X, Y, Z);
26 if (ok)
27 System.out.println
28 ("Quadratic: " + Root1 + Root2);
29 else
30 System.out.println ("No Solution.");
31 }
32
33 // Three positive integers, finds the quadratic root
34 private static boolean Root (int A, int B, int C)
35 {
36 double D;
37 boolean Result;
38 D = (double) (B*B) - (double) (4.0*A*C);
39 if (D < 0.0)
40 {
41 Result = false;
42 return (Result);
43 }
44 Root1 = (double) ((-B + Math.sqrt (D)) / (2.0*A));
45 Root2 = (double) ((-B – Math.sqrt (D)) / (2.0*A));
46 Result = true;
47 return (Result);
48 } / /End method Root
49 } // End class Quadratic

Quadratic – Coupling DU-pairs

Pairs of locations: method name, variable name, statement

(main (), X, 15) – (Root (), A, 38)

(main (), Y, 16) – (Root (), B, 38)

(main (), Z, 17) – (Root (), C, 38)

(main (), X, 21) – (Root (), A, 38)

(main (), Y, 22) – (Root (), B, 38)

(main (), Z, 23) – (Root (), C, 38)

(Root (), Root1, 44) – (main (), Root1, 28)

(Root (), Root2, 45) – (main (), Root2, 28)

(Root (), Result, 41) – (main (), ok, 26)

(Root (), Result, 46) – (main (), ok, 26)

Coupling Data Flow Notes

- Only variables that are used or defined in the callee
- Implicit initializations of class and global variables
- Transitive DU-pairs are too expensive to handle
  - A calls B, B calls C, and there is a variable defined in A and used in C
- Arrays : a reference to one element is considered to be a reference to all elements

Inheritance, Polymorphism & Dynamic Binding

- Additional control and data connections make data flow analysis more complex
- The defining and using units may be in different call hierarchies
- When inheritance hierarchies are used, a def in one unit could reach uses in any class in the inheritance hierarchy
- With dynamic binding, the same location can reach different uses depending on the current type of the using object
- The same location can have different definitions or uses at different points in the execution !

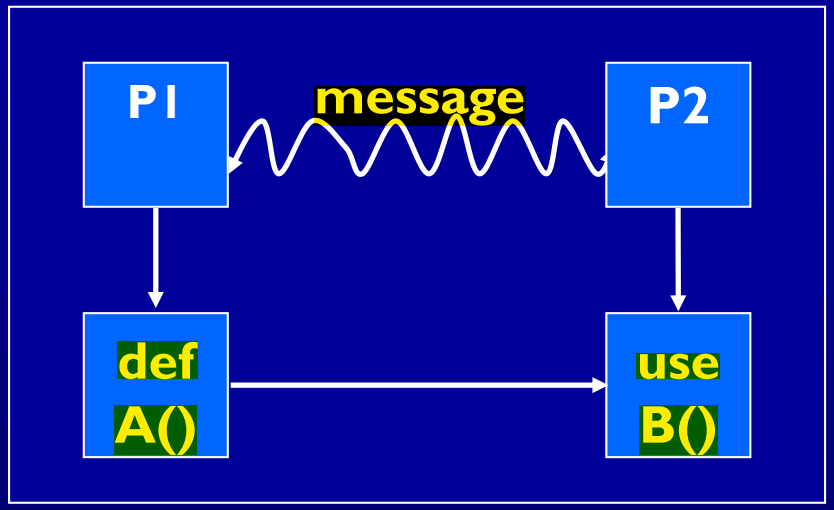
OO Data Flow Summary

- The defs and uses could be in the same class, or different classes

- Researchers have applied data flow testing to the direct coupling OO situation
  - Has not been used in practice
  - No tools available

- Indirect coupling data flow testing has not been tried either in research or in practice
  - Analysis cost may be prohibitive

Web Applications and Other Distributed Software



distributed software data flow

- "message" could be HTTP, RMI, or other mechanism
- A() and B() could be in the same class or accessing a persistent variable such as in a web session
- Beyond current technologies

Summary—What Works?

- Call graphs are common and very useful ways to design integration tests
- Inter-procedural data flow is relatively easy to compute and results in effective integration tests
- The ideas for OO software and web applications are preliminary and have not been used much in practice

Introduction to Software Testing
(2nd edition)
Chapter 7.5

Graph Coverage for Specifications

Paul Ammann & Jeff Offutt

http://www.cs.gmu.edu/~offutt/softwaretest/

Design Specifications

- A design specification describes aspects of what behavior software should exhibit

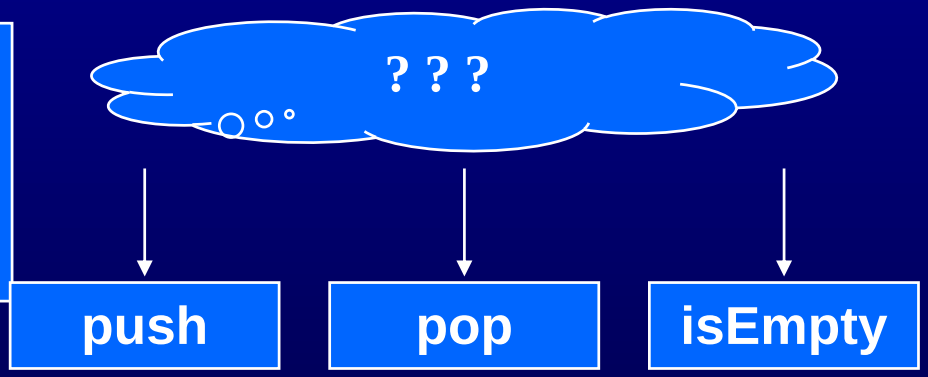
- A design specification may or may not reflect the implementation
  - More accurately – the implementation may not exactly reflect the spec
  - Design specifications are often called models of the software

- Two types of descriptions are used in this chapter
  1. Sequencing constraints on class methods
  2. State behavior descriptions of software

Sequencing Constraints

- Sequencing constraints are rules that impose constraints on the order in which methods may be called
- They can be encoded as preconditions or other specifications
- Section 7.4 said that classes often have methods that do not call each other

Class stack
public void push (Object o)
public Object pop ( )
public boolean isEmpty ( )



- Tests can be created for these classes as sequences of method calls
- Sequencing constraints give an easy and effective way to choose which sequences to use

Sequencing Constraints Overview

- Sequencing constraints might be
  - Expressed explicitly
  - Expressed implicitly
  - Not expressed at all
- Testers should derive them if they do not exist
  - Look at existing design documents
  - Look at requirements documents
  - Ask the developers
  - Last choice : Look at the implementation
- If they don't exist, expect to find more faults !
- Share with designers before designing tests
- Sequencing constraints do not capture all behavior

Queue Example

```
public int deQueue()
{
  // Pre: At least one element must be on the queue.
  … …

public enQueue (int e)
{
  // Post: e is on the end of the queue.
```

- Sequencing constraints are implicitly embedded in the pre and postconditions
  - enQueue () must be called before deQueue ()
- Does not include the requirement that we must have at least as many enQueue () calls as deQueue () calls
  - Can be handled by state behavior techniques

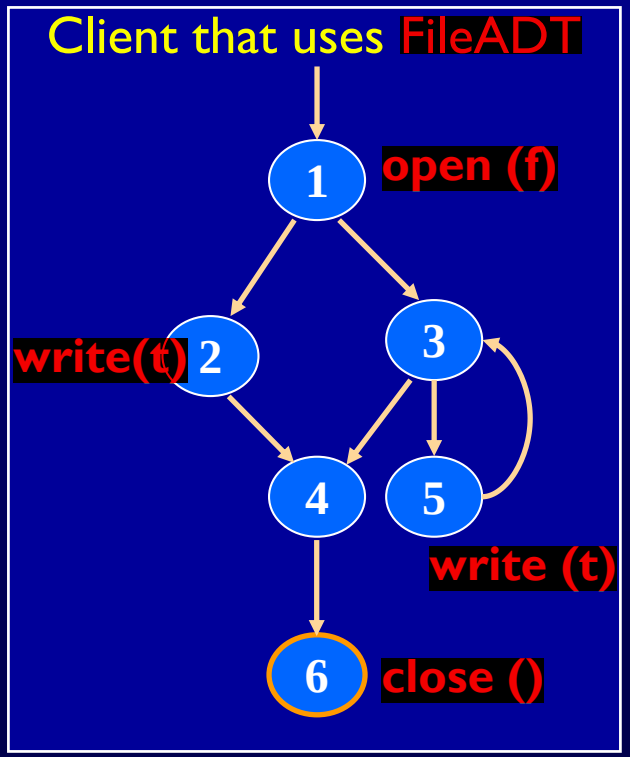
File ADT Example

class FileADT has three methods:

- **open (String fName)** // Opens file with name fName
- **close ()** // Closes the file and makes it unavailable
- **write (String textLine)** // Writes a line of text to the file

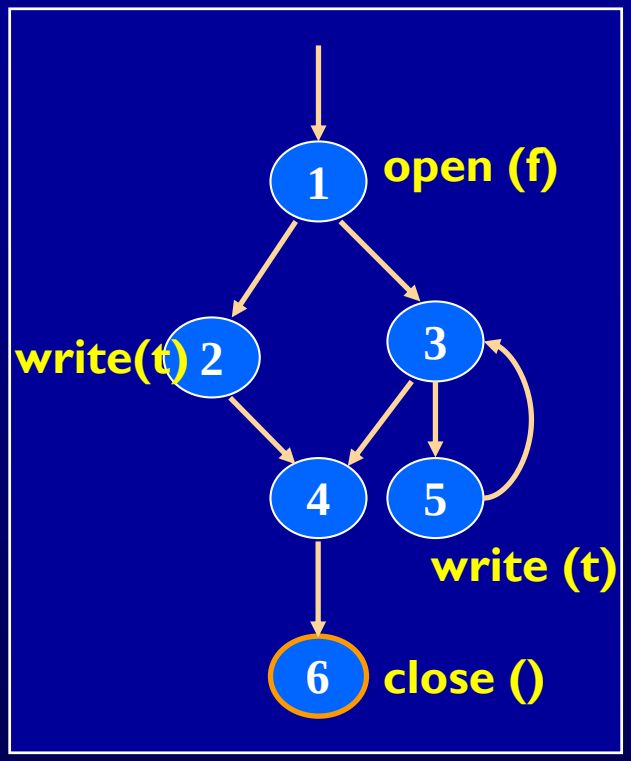
Valid sequencing constraints on FileADT:

1. An open (f) must be executed before every write (t)
2. An open (f) must be executed before every close ()
3. A write (f) may not be executed after a close () unless there is an open (f) in between
4. A write (t) should be executed before every close ()



Static Checking

Is there a path that violates any of the sequencing constraints ?

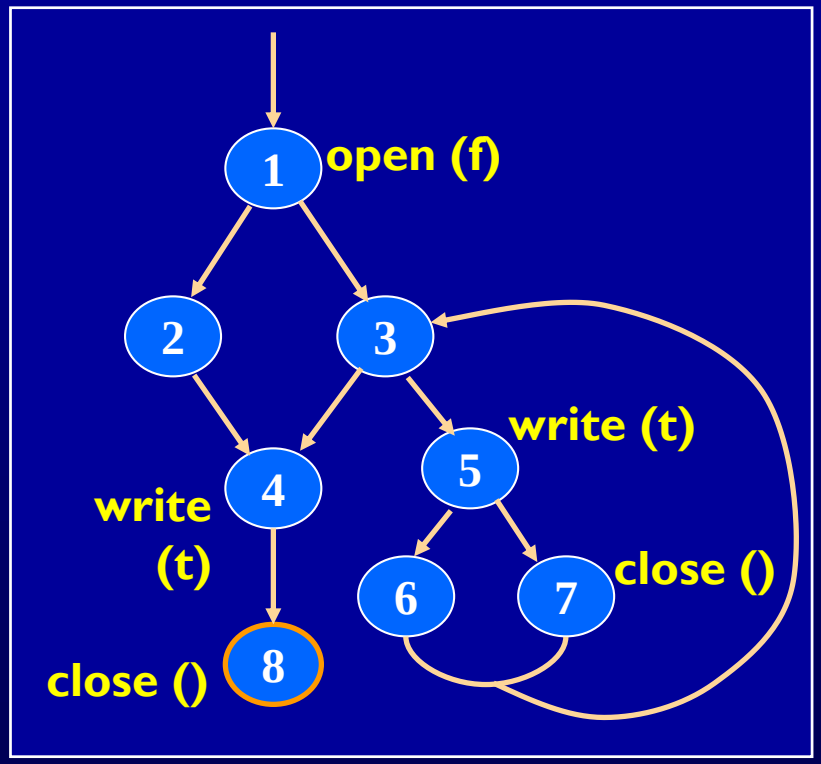


- Is there a path to a write() that does not go through an open() ?
- Is there a path to a close() that does not go through an open() ?
- Is there a path from a close() to a write()?
- Is there a path from an open() to a close() that does not go through a write() ? ("write-clear" path)

[1, 3, 4, 6] – ADT use anomaly!

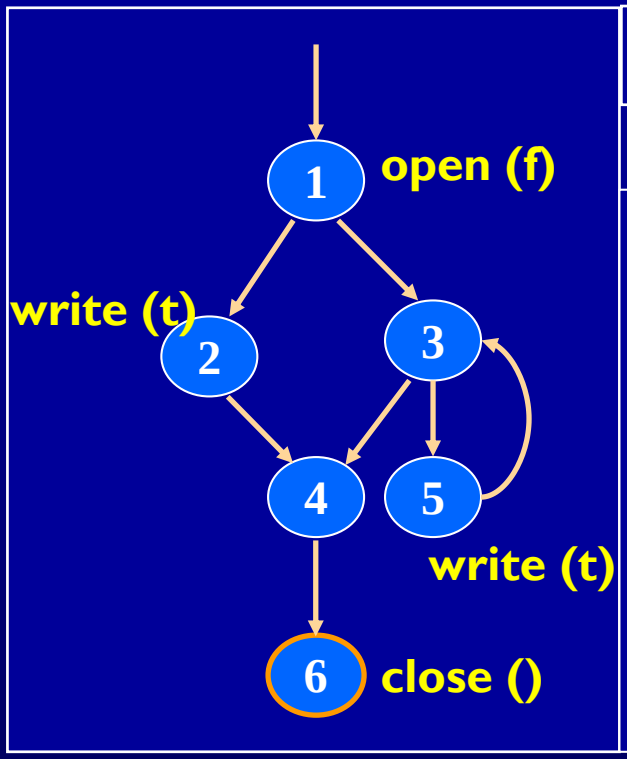
Static Checking

Consider the following graph :



[7, 3, 4] – close () before write () !

Generating Test Requirements



[1, 3, 4, 6] – ADT use anomaly!

- But it is possible that the logic of the program does not allow the pair of edges [1, 3, 4]
- That is – the loop body must be taken at least once
- Determining this is undecidable – so static methods are not enough

- Use the sequencing constraints to generate test requirements
- The goal is to violate every sequencing constraint

Test Requirements for FileADT

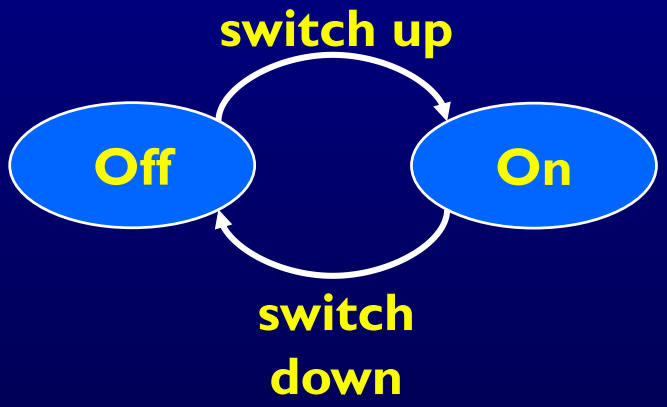
Apply to all programs that use FileADT

1. Cover every path from the start node to every node that contains a write() such that the path does not go through a node containing an open()
2. Cover every path from the start node to every node that contains a close() such that the path does not go through a node containing an open()
3. Cover every path from every node that contains a close() to every node that contains a write()
4. Cover every path from every node that contains an open() to every node that contains a close() such that the path does not go through a node containing a write()

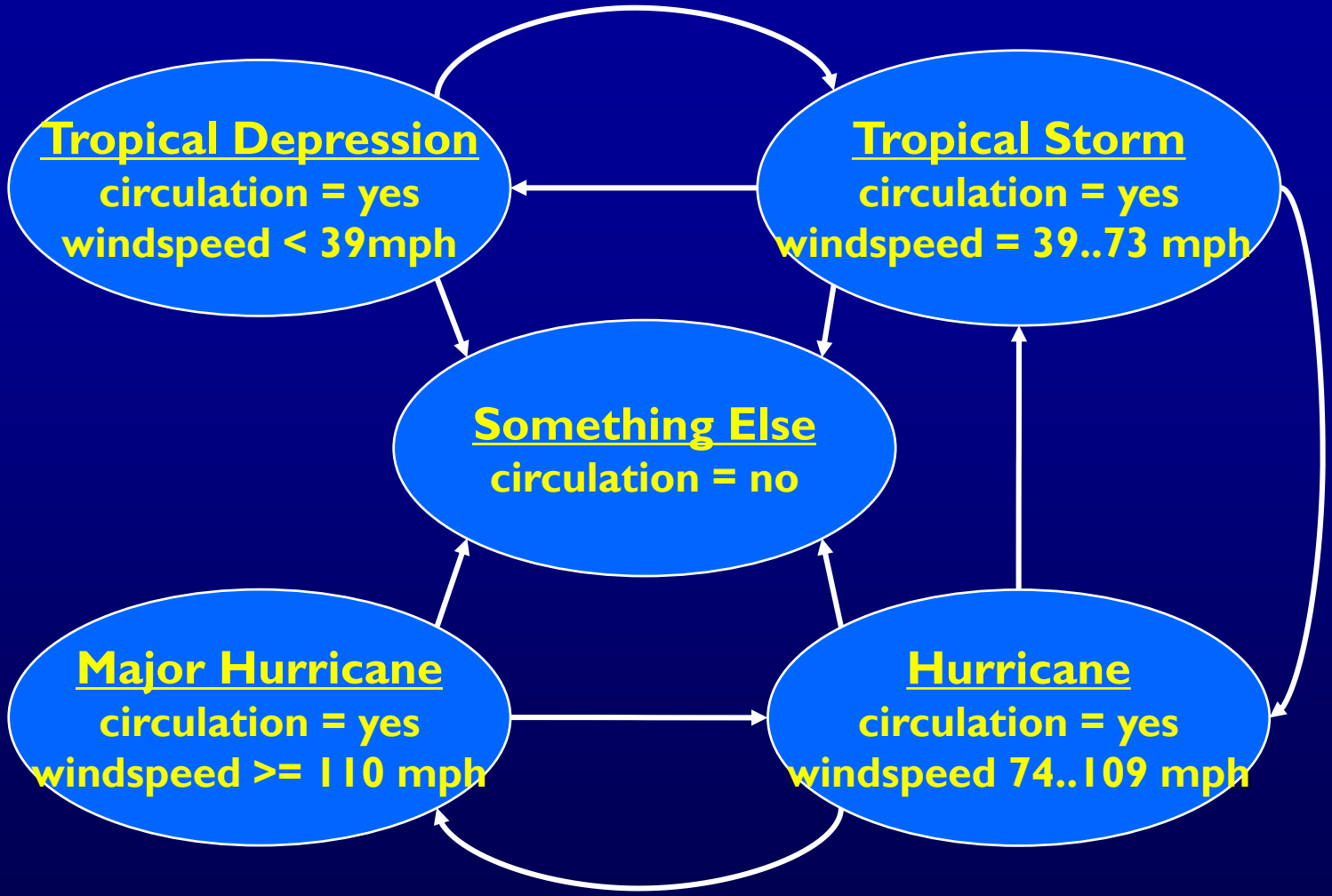
- If program is correct, all test requirements will be infeasible
- Any tests created will almost definitely find faults

Testing State Behavior

- A finite state machine (FSM) is a graph that describes how software variables are modified during execution
- Nodes : States, representing sets of values for key variables
- Edges : Transitions, possible changes in the state



Finite State Machine—Two Variables



Other variables may exist but not be part of state

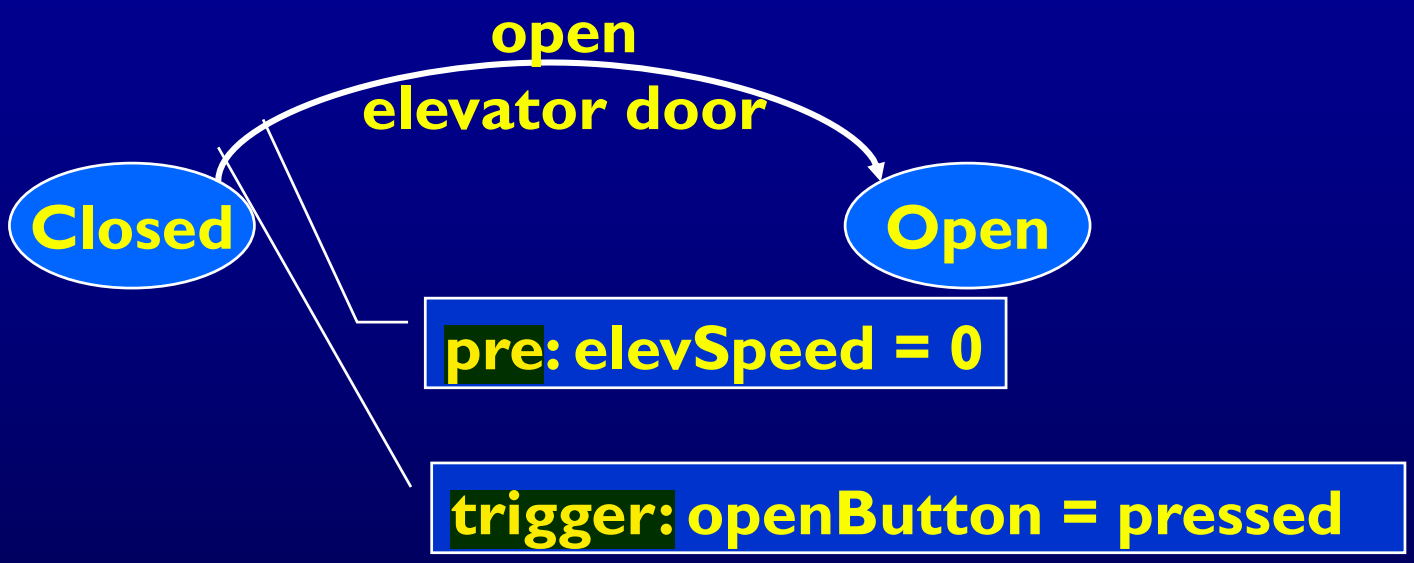
Finite State Machines are Common

- FSMs can accurately model many kinds of software
  - Embedded and control software (think electronic gadgets)
  - Abstract data types
  - Compilers and operating systems
  - Web applications
- Creating FSMs can help find software problems
- Numerous languages for expressing FSMs
  - UML statecharts
  - Automata
  - State tables (SCR)
  - Petri nets
- Limitation : FSMs are not always practical for programs that have lots of states (for example, GUIs)

Annotations on FSMs

- FSMs can be annotated with different types of actions
  - Actions on transitions
  - Entry actions to nodes
  - Exit actions on nodes
- Actions can express changes to variables or conditions on variables
- These slides use the basics:
  - Preconditions (guards) : conditions that must be true for transitions to be taken
  - Triggering events : changes to variables that cause transitions to be taken
- This is close to the UML Statecharts, but not exactly the same

Example Annotations



Covering FSMs

- Node coverage : execute every state (*state coverage*)
- Edge coverage : execute every transition (*transition coverage*)
- Edge-pair coverage : execute every pair of transitions (*transition-pair*)
- Data flow:
  - Nodes often do not include defs or uses of variables
  - Defs of variables in triggers are used immediately (the next state)
  - Defs and uses are usually computed for guards, or states are extended
  - FSMs typically only model a subset of the variables
- Generating FSMs is often harder than covering them …

Deriving FSMs

- With some projects, an FSM (such as a statechart) was created during design
  - Tester should check to see if the FSM is still current with respect to the implementation
- If not, it is very helpful for the tester to derive the FSM
- Strategies for deriving FSMs from a program:
  1. Combining control flow graphs (*wrong*)
  2. Using the software structure (*wrong*)
  3. Modeling state variables
- Example based on a digital watch …
  - Class Watch uses class Time

Class Watch

```
class Watch
// Constant values for the button (inputs)
private static final int NEXT = 0;
private static final int UP   = 1;
private static final int DOWN = 2;
// Constant values for the state
private static final int TIME      = 5;
private static final int STOPWATCH = 6;
private static final int ALARM     = 7;
// Primary state variable
private int mode = TIME;
// Three separate times, one for each state
private Time watch, stopwatch, alarm;

public Watch () // Constructor
public void doTransition (int button) // Handles inputs
public String toString ()  // Converts values
```

```
class Time   ( inner class )
private int hour   = 0;
private int minute = 0;

public void changeTime (int button)
public String toString ()
```

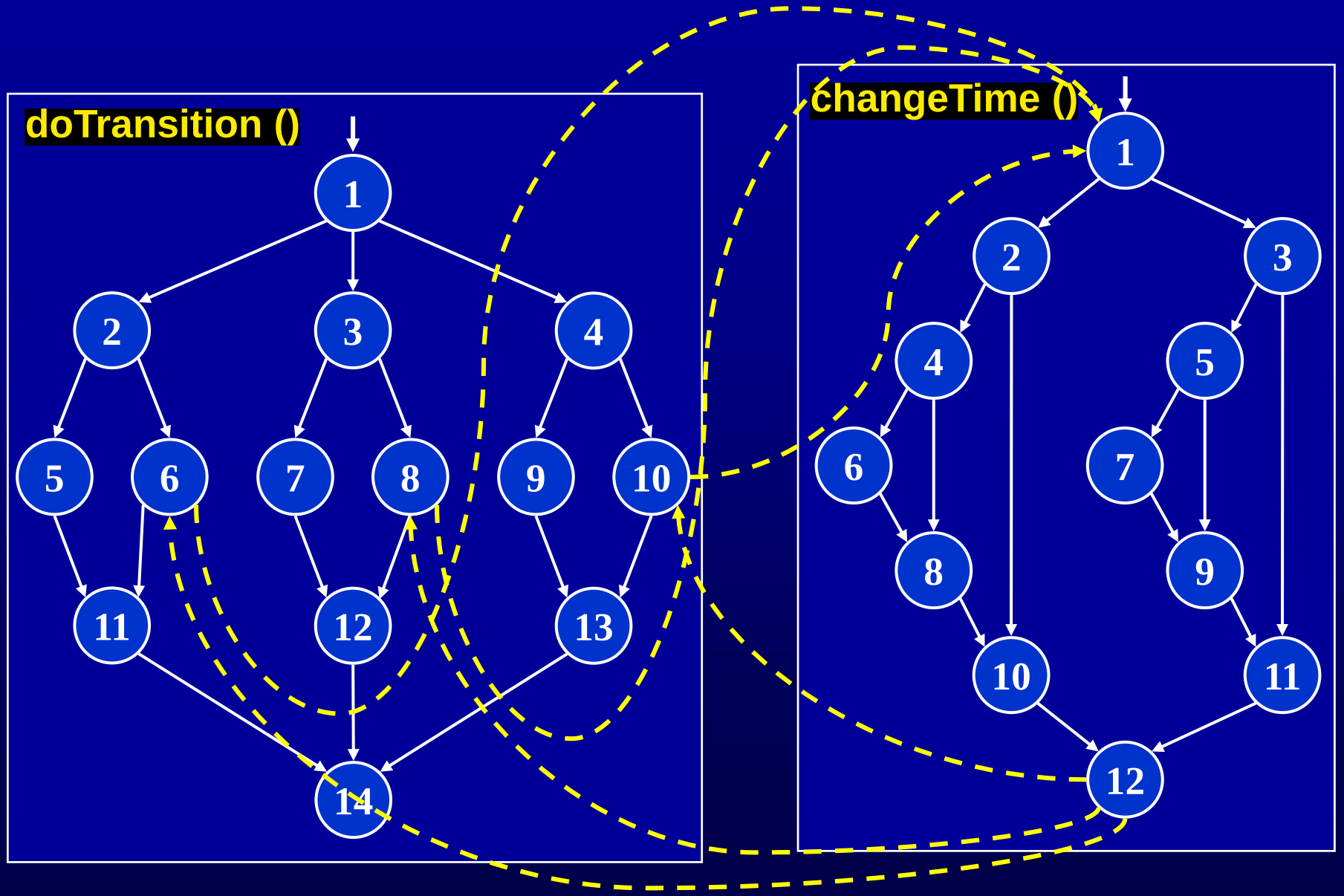
```
// Takes the appropriate transition when a button is pushed.
public void doTransition (int button)
{
   switch ( mode )
   {
      case TIME:
         if (button == NEXT)
            mode = STOPWATCH;
         else
            watch.changeTime (button);
         break;
      case STOPWATCH:
         if (button == NEXT)
            mode = ALARM;
         else
            stopwatch.changeTime (button);
         break;
      case ALARM:
         if (button == NEXT)
            mode = TIME;
         else
            alarm.changeTime (button);
         break;
      default:
         break;
   }
} // end doTransition()
```

```
// Increases or decreases the time.
// Rolls around when necessary.
public void changeTime (int button)
{
   if (button == UP)
   {
      minute += 1;
      if (minute >= 60)
      {
         minute = 0;
         hour += 1;
         if (hour > 12)
            hour = 1;
      }
   }
   else if (button == DOWN)
   {
      minute -= 1;
      if (minute < 0)
      {
         minute = 59;
         hour -= 1;
         if (hour <= 0)
            hour = 12;
      }
   }
} // end changeTime()
```

1. Combining Control Flow Graphs

- The first instinct for inexperienced developers is to draw CFGs and link them together
- This is really not an FSM
- Several problems
  - Methods must return to correct callsites—implicit nondeterminism
  - Implementation must be available before graph can be built
  - This graph does not scale up
- Watch example …

CFGs for Watch



2. Using the Software Structure

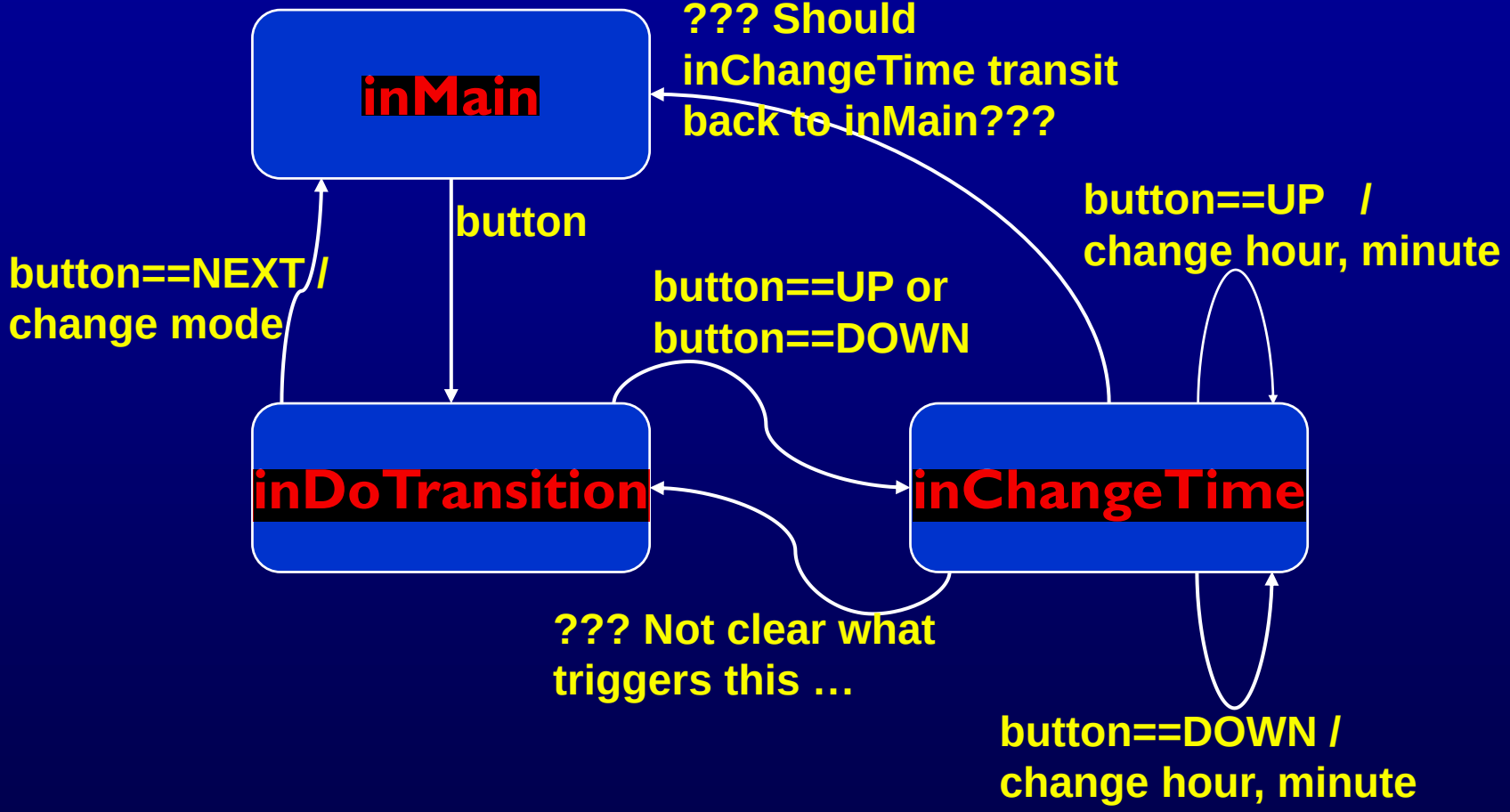
- A more experienced programmer may map methods to states

- These are really not states

- Problems
  - Subjective—different testers get different graphs
  - Requires in-depth knowledge of implementation
  - Detailed design must be present

- Watch example …

Software Structure for Watch



3. Modeling State Variables

- More mechanical
- State variables are usually defined early
- First identify all state variables, then choose which are relevant
- In theory, every combination of values for the state variables defines a different state
- In practice, we must identify ranges, or sets of values, that are all in one state
- Some states may not be feasible

State Variables in Watch

Constants

- ~~NEXT, UP, DOWN~~
- ~~TIME, STOPWATCH, ALARM~~

Not relevant, really just values

Non-constant variables in class Watch

- int mode (values: TIME, STOPWATCH, ALARM)
- Time watch, stopwatch, alarm

State Variables in Time

Non-constant variables in class Time

- int hour (values: 1..12)
- int minute (values: 0 .. 59)

12 X 60 values is 720 states

Clearly, that is too many

Combine values into ranges of similar values :

- hour : 1.. 11, 12
- minute : 0, 1.. 59

Clumsy ... Not sequential …

let's combine hour and minute …

Four states : (1..11, 0); (12, 0); (1..11, 1.. 59); (12, 1 .. 59)

Time : 12:00, 12:01..12:59, 01:00 .. 11:59

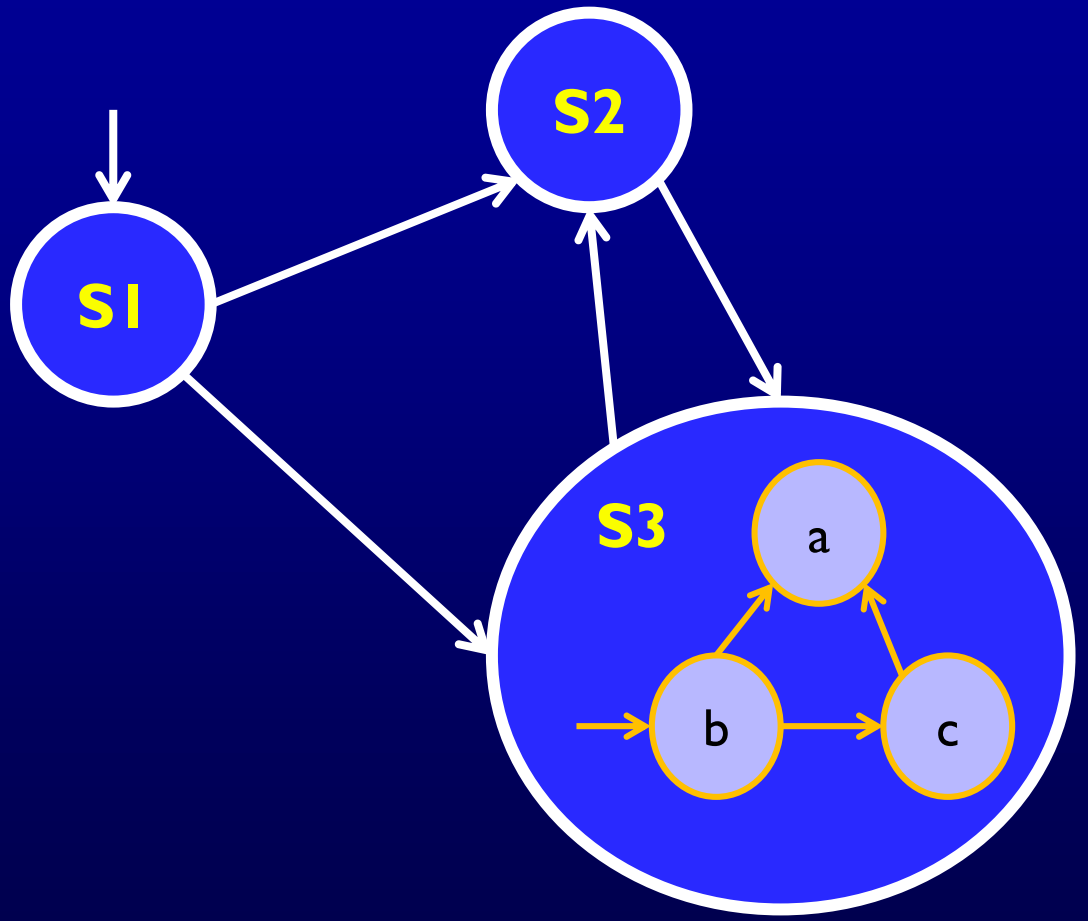
These require lots of thought and semantic domain knowledge of the program

Hierarchical FSMs

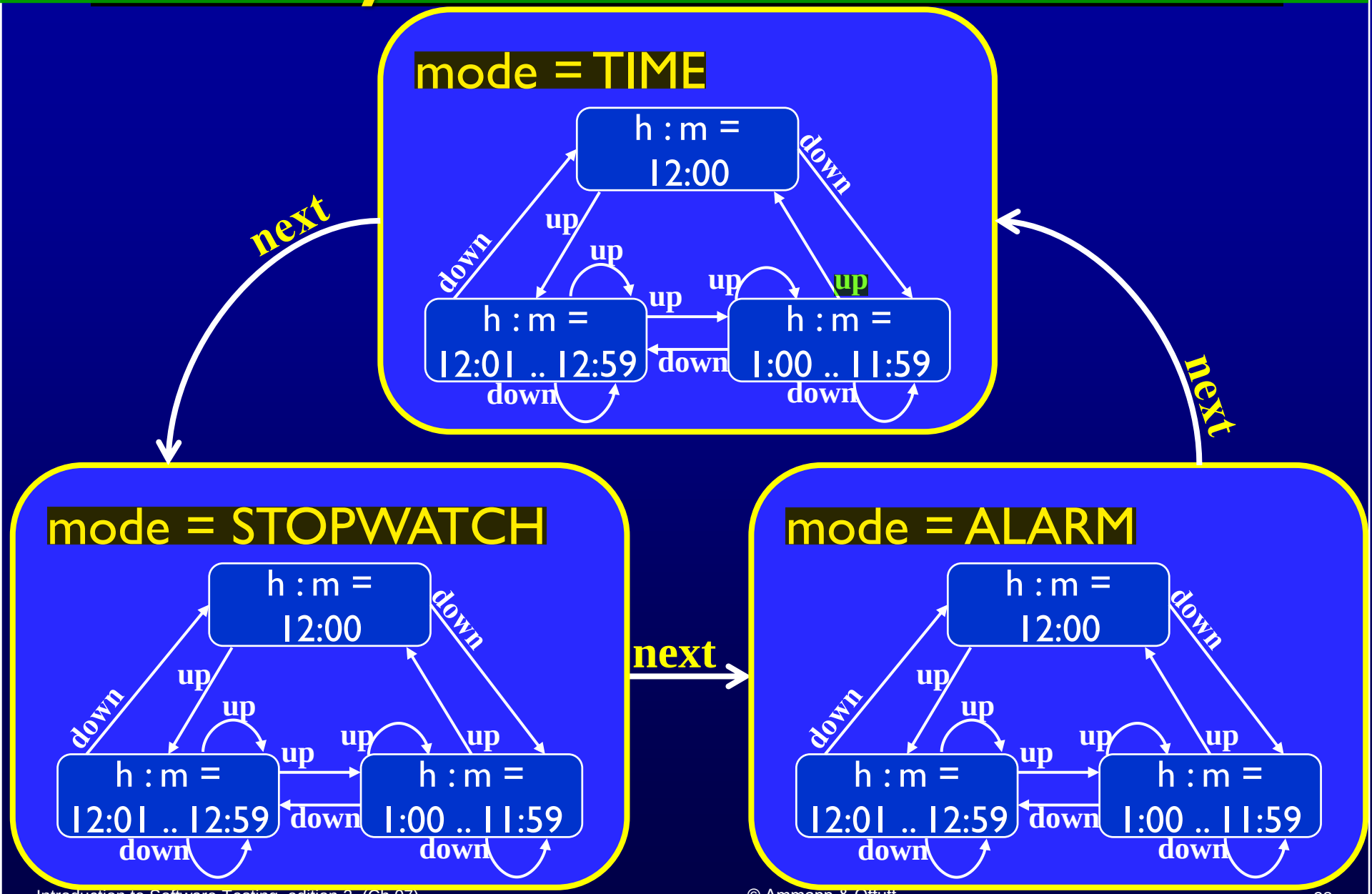
One FSM is contained within the other

Class Watch uses class Time

How can we model two classes—one that uses another?



Watch / Time Hierarchical FSM



Summary–Tradeoffs in Applying Graph Coverage Criteria to FSMs

- Two advantages
  1. Tests can be designed before implementation
  2. Analyzing FSMs is much easier than analyzing source

- Three disadvantages
  1. Some implementation decisions are not modeled in the FSM
  2. There is some variation in the results because of the subjective nature of deriving FSMs
  3. Tests have to be "mapped" to actual inputs to the program – the names that appear in the FSM may not be the same as the names in the program

Introduction to Software Testing
(2nd edition)
Chapter 7.6

Graph Coverage for Use Cases

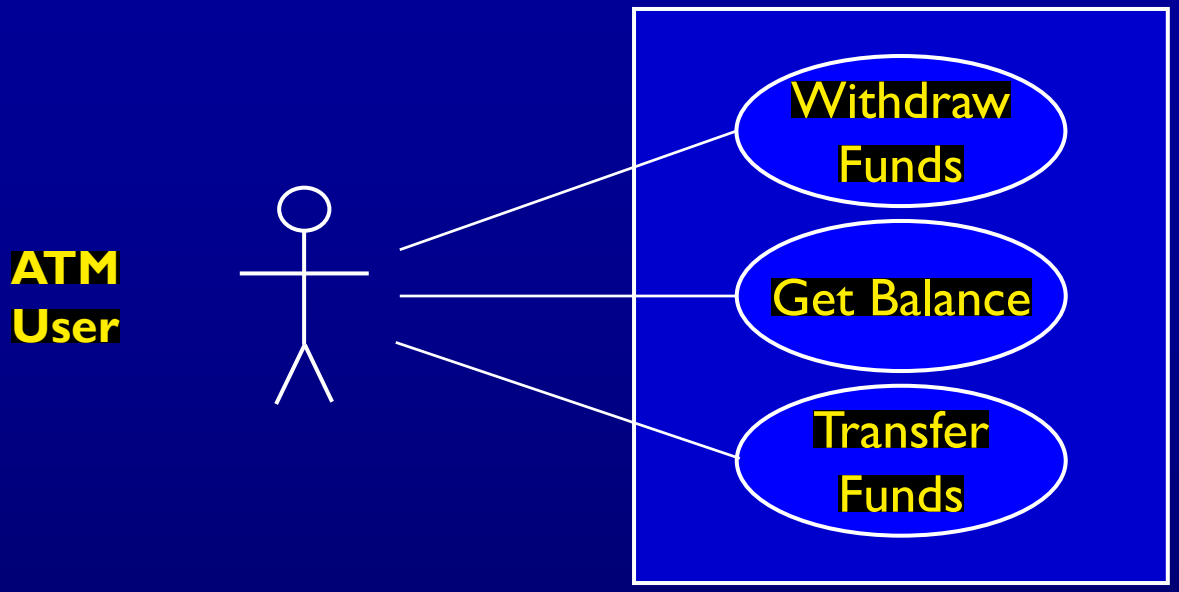
Paul Ammann & Jeff Offutt

http://www.cs.gmu.edu/~offutt/softwaretest/

UML Use Cases

- UML use cases are often used to express software requirements
- They help express computer application workflow
- We won't teach use cases, but show examples

Simple Use Case Example



- Actors : Humans or software components that use the software being modeled
- Use cases : Shown as circles or ovals
- Node Coverage : Try each use case once …

Use case graphs, by themselves, are not useful for testing

Elaboration

- Use cases are commonly elaborated (or documented)
- Elaboration is first written textually
  - Details of operation
  - Alternatives model choices and conditions during execution

Elaboration of ATM Use Case

- Use Case Name : Withdraw Funds
- Summary : Customer uses a valid card to withdraw funds from a valid bank account.
- Actor : ATM Customer
- Precondition : ATM is displaying the idle welcome message
- Description :
  - Customer inserts an ATM Card into the ATM Card Reader.
  - If the system can recognize the card, it reads the card number.
  - System prompts the customer for a PIN.
  - Customer enters PIN.
  - System checks the card's expiration date and whether the card has been stolen or lost.
  - If the card is valid, the system checks if the entered PIN matches the card PIN.
  - If the PINs match, the system finds out what accounts the card can access.
  - System displays customer accounts and prompts the customer to choose a type of transaction. There are three types of transactions, Withdraw Funds, Get Balance and Transfer Funds. (The previous eight steps are part of all three use cases; the following steps are unique to the Withdraw Funds use case.)

Elaboration of ATM Use Case–(2/3)

- Description (continued) :
  - Customer selects Withdraw Funds, selects the account number, and enters the amount.
  - System checks that the account is valid, makes sure that customer has enough funds in the account, makes sure that the daily limit has not been exceeded, and checks that the ATM has enough funds.
  - If all four checks are successful, the system dispenses the cash.
  - System prints a receipt with a transaction number, the transaction type, the amount withdrawn, and the new account balance.
  - System ejects card.
  - System displays the idle welcome message.

Elaboration of ATM Use Case–(3/3)

- Alternatives : exceptional flow
  - If the system cannot recognize the card, it is ejected and the welcome message is displayed.
  - If the current date is past the card's expiration date, the card is confiscated and the welcome message is displayed.
  - If the card has been reported lost or stolen, it is confiscated and the welcome message is displayed.
  - If the customer entered PIN does not match the PIN for the card, the system prompts for a new PIN.
  - If the customer enters an incorrect PIN three times, the card is confiscated and the welcome message is displayed.
  - If the account number entered by the user is invalid, the system displays an error message, ejects the card and the welcome message is displayed.
  - If the request for withdraw exceeds the maximum allowable daily withdrawal amount, the system displays an apology message, ejects the card and the welcome message is displayed.
  - If the request for withdraw exceeds the amount of funds in the ATM, the system displays an apology message, ejects the card and the welcome message is displayed.
  - If the customer enters Cancel, the system cancels the transaction, ejects the card and the welcome message is displayed.
- Postcondition :
  - Funds have been withdrawn from the customer's account.

Wait A Minute ...

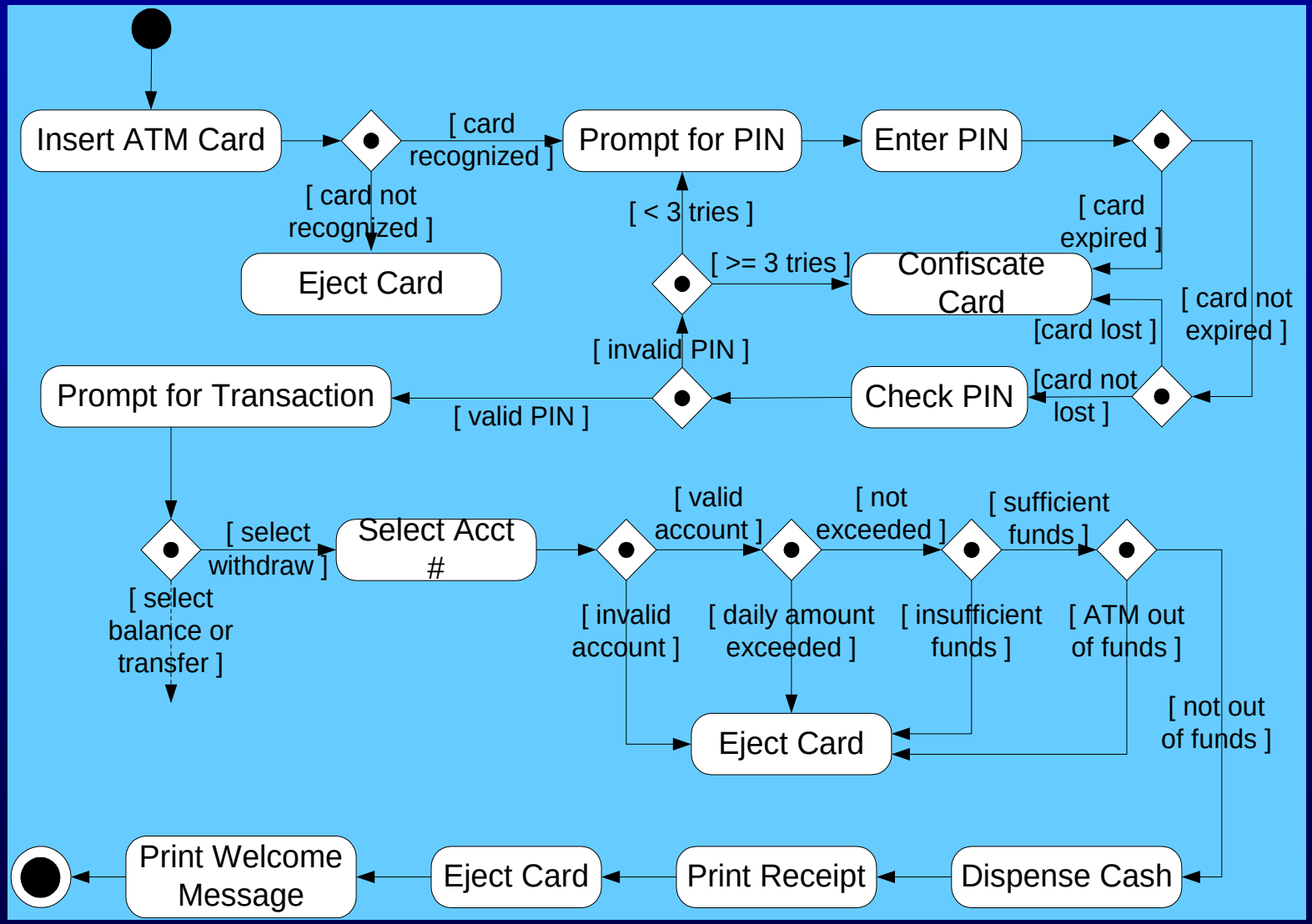
- What does this have to do with testing ?
- Specifically, what does this have to do with graphs ???
- Remember our admonition : Find a graph, then cover it!
- Beizer suggested "Transaction Flow Graphs" in his book
- UML has something very similar :

Activity Diagrams

Use Cases to Activity Diagrams

- Activity diagrams indicate flow among activities
- Activities should model user level steps
- Two kinds of nodes:
  - Action states
  - Sequential branches
- Use case descriptions become action state nodes in the activity diagram
- Alternatives are sequential branch nodes
- Flow among steps are edges
- Activity diagrams usually have some helpful characteristics:
  - Few loops
  - Simple predicates
  - No obvious DU pairs

ATM Withdraw Activity Graph



Covering Activity Graphs

- Node Coverage
  - Inputs to the software are derived from labels on nodes and predicates
  - Used to form test case values
- Edge Coverage
- Data flow techniques do not apply
- Scenario Testing
  - Scenario : A complete path through a use case activity graph
  - Should make semantic sense to the users
  - Number of paths often finite
  - If not, scenarios defined based on domain knowledge
  - Use "specified path coverage," where the set S of paths is the set of scenarios
  - Note that specified path coverage does not necessarily subsume edge coverage, but scenarios should be defined so that it does

Summary of Use Case Testing

- Use cases are defined at the requirements level
- Can be very high level
- UML Activity Diagrams encode use cases in graphs
  - Graphs usually have a fairly simple structure
- Requirements-based testing can use graph coverage
  - Straightforward to do by hand
  - Specified path coverage makes sense for these graphs