CS 563: Software Maintenance And Evolution

Software Defect Localization

Oregon State University, Spring 2024

What have we learnt so far?

What, Why, When, and how?

- Software maintenance (4 types)
- Software change management (SCM coordinator, VCSs)
- Software design patterns (GangsOfFour, SOLID, MVC)
- Evolution of Software Evolution (Lehman laws's)
- Software defect prediction
  - Code complexity metrics (Control Flow Graphs)
  - ML – based prediction techniques

Today’s plan

- *Software Fault Localization*
- In-class exercise: Advanced uses of Git
  (graded exercise, team of 2, due Sunday, April 21, 11:59 PM)

Software Debugging

What are software ***faults*** and how do they differ from software ***defects***?

Software Debugging

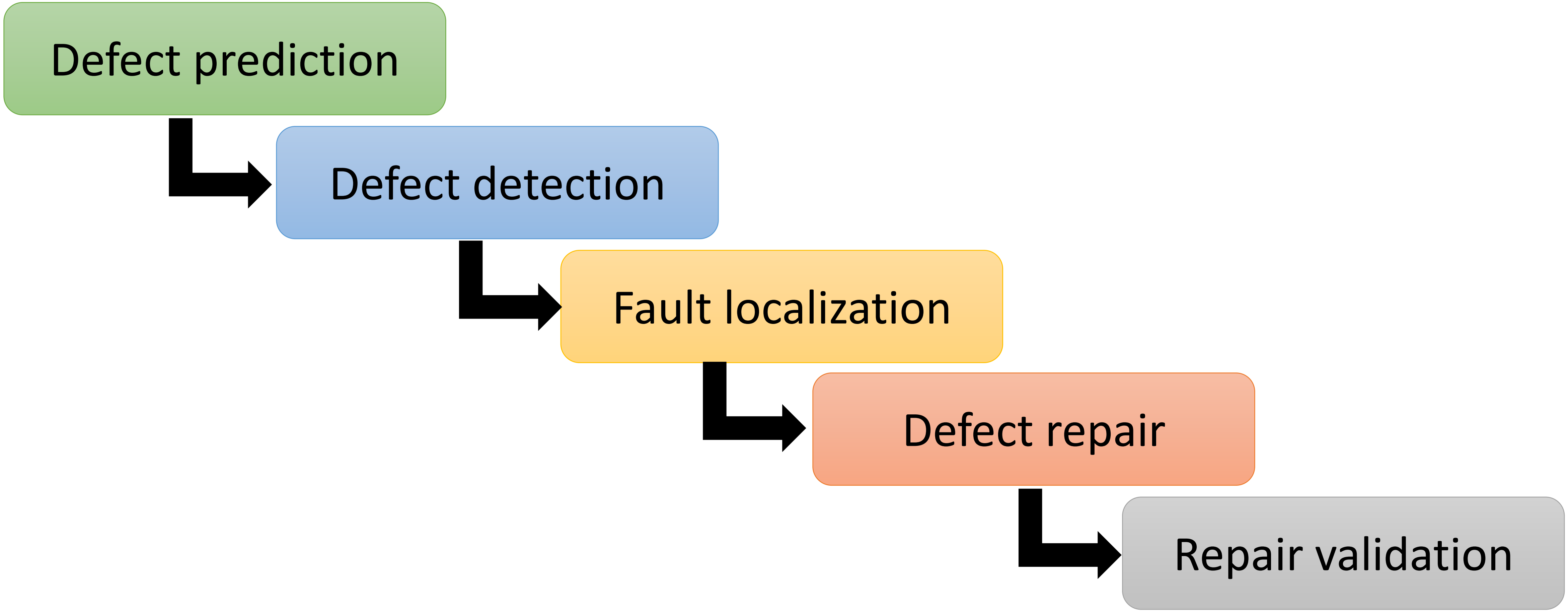
What are software ***faults*** and how do they differ from software ***defects***?

- ***Defect:*** Bugs or errors in the code, documentation, requirements, etc. where the actual behavior or software differs from its expected behavior.
- ***Fault:*** A static issue in the source code that can potentially cause failure (e.g., syntax errors, logical errors, incorrect algorithm).

Software defect is the actual manifestation of a ***fault*** that leads to observable incorrect behavior in the software

Software Debugging Process

What is software debugging and what all activities does it involve?



Defect prediction

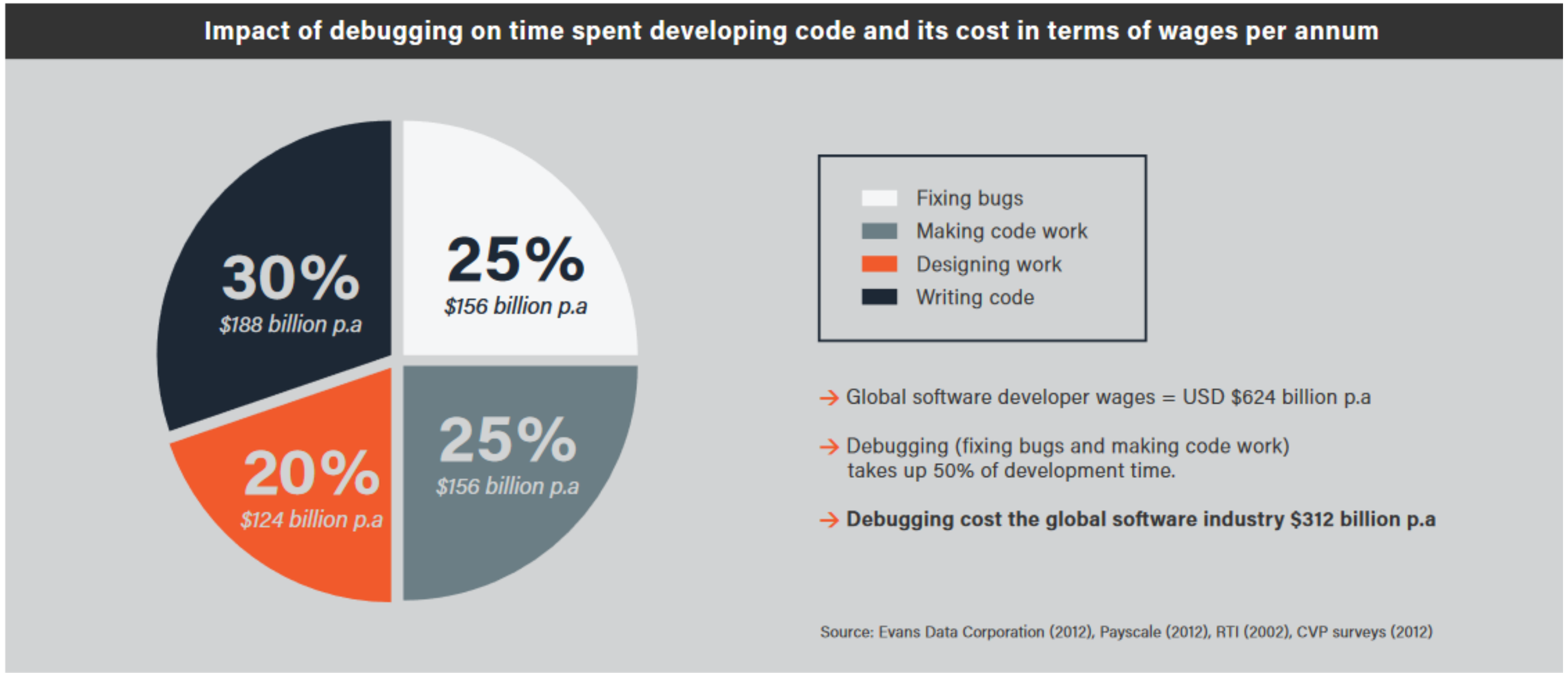
Defect detection

Fault localization

Defect repair

Repair validation

Software Debugging Is Expensive



Impact of debugging on time spent developing code and its cost in terms of wages per annum

→ Global software developer wages = USD $624 billion p.a

→ Debugging (fixing bugs and making code work) takes up 50% of development time.

→ **Debugging cost the global software industry $312 billion p.a**

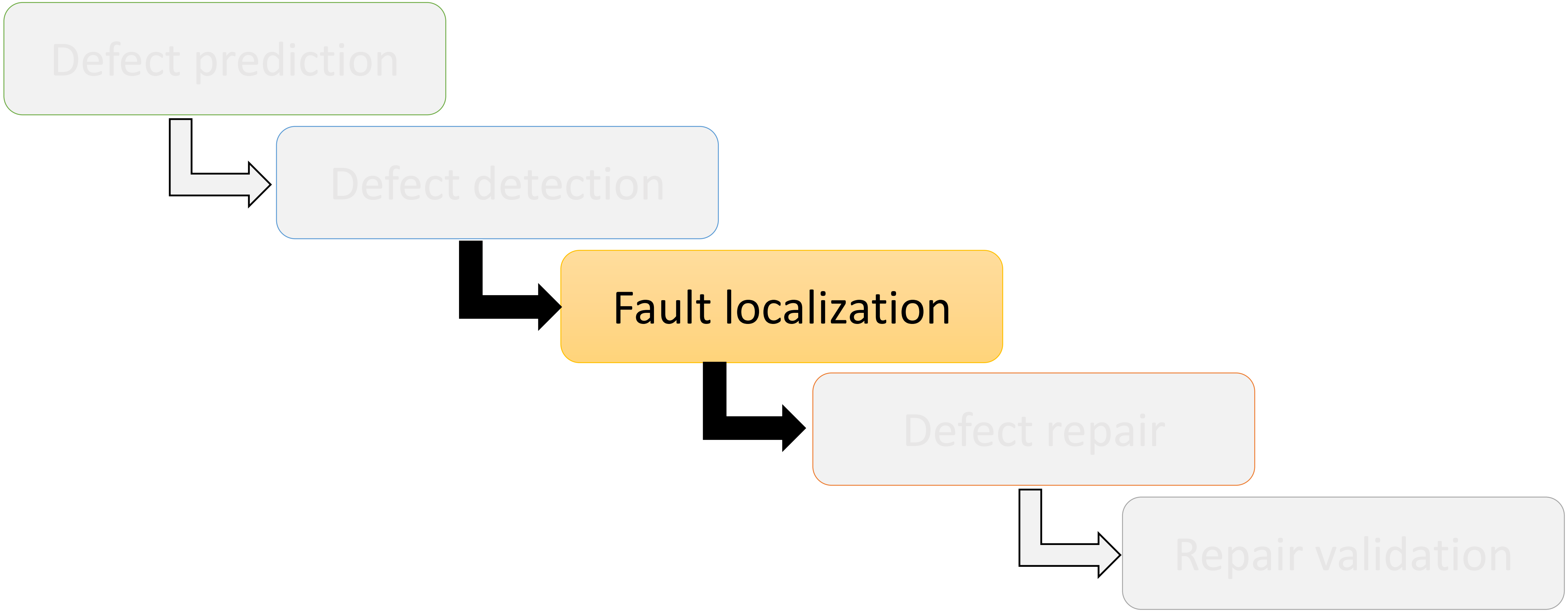
Source: Evans Data Corporation (2012), Payscale (2012), RTI (2002), CVP surveys (2012)

“Software developers spend 35-50% of their time validating and debugging software. The cost of debugging, testing, and verification is estimated to account for 50-75% of the total budget of software development projects, amounting to more than $100 billion annually.”

Source: Britton, T., Jeng, L., Carver, G., Cheak, P., Katzenellenbogen, T. 2013. Reversible debugging software. Cambridge Judge Business School; http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.444.9094&rep=rep1&type=pdf.

Software Fault Localization

What is software fault localization and why should we care about localizing software faults?



Defect prediction

Defect detection

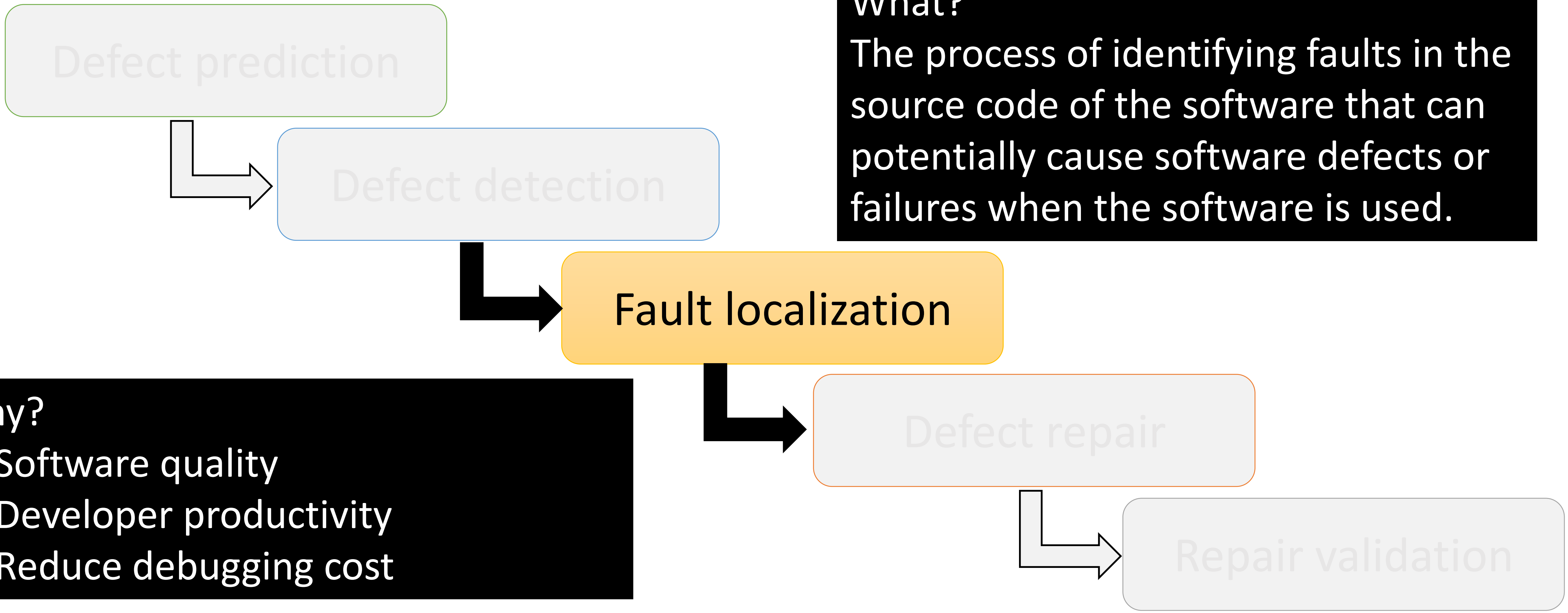
Fault localization

Defect repair

Repair validation

Software Fault Localization

What is software fault localization and **why** should we care about localizing software faults?



Defect prediction

Defect detection

Fault localization

Defect repair

Repair validation

What?
The process of identifying faults in the source code of the software that can potentially cause software defects or failures when the software is used.

Why?

- Software quality
- Developer productivity
- Reduce debugging cost

Software Fault Localization

How to know which software code elements are buggy?

Program Elements

What is a software program composed of?

- Files (or classes in OOP)
- Functions (or methods)
- Statements (remember they are different from lines!)

A bug can be localized to any of the above listed granularities. Therefore, these are called ***program elements***.

Perfect Bug Detection

A bug in an element will be detected by a programmer if that element is manually examined

- A correct element will not be mistakenly identified as a faulty element
- If the assumption does not hold, a programmer may need to examine more code than necessary to find a faulty element

Commonly used traditional techniques

Commonly used traditional techniques

- Insert *print* statements
- Add *assertions* or set *breakpoints*
- Examine *core dump* or *stack trace*

Rely on programmers’ intuition and domain expert knowledge

Commonly used traditional techniques

- Insert *print* statements
- Add *assertions* or set *breakpoints*
- Examine *core dump* or *stack trace*

Rely on programmers’ intuition and domain expert knowledge

- *How to automate it?*

Execution Slice & Dice

- **Key insight:** Faults reside in the ***execution slice*** of a test that fails on execution
  - An ***execution slice*** is the set of a program’s code (blocks, statements, decisions, c-uses, or p-uses) executed by a test

Execution Slice & Dice

- **Key insight:** Faults reside in the ***execution slice*** of a test that fails on execution
    - An ***execution slice*** is the set of a program’s code (blocks, statements, decisions, c-uses (computation uses), or p-uses (predicate uses)) executed by a test
    - An execution slice can be constructed very easily if we know the ***coverage*** of the test (*instead of reporting the coverage percentage, it reports which parts of the program are covered*).

Execution Slice & Dice

- **Key insight:** Faults reside in the ***execution slice*** of a test that fails on execution
  - An ***execution slice*** is the set of a program's code (blocks, statements, decisions, c-uses (computation uses), or p-uses (predicate uses)) executed by a test
  - An execution slice can be constructed very easily if we know the ***coverage*** of the test (*instead of reporting the coverage percentage, it reports which parts of the program are covered*).
  - Too much code in the slice!

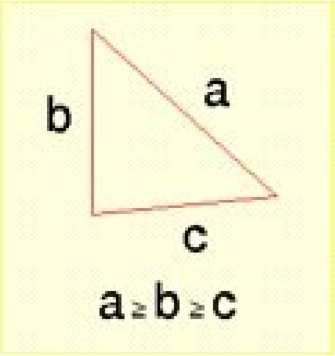
Execution Slice & Dice

- **Key insight:** Faults reside in the ***execution slice*** of a test that fails on execution
  - An ***execution slice*** is the set of a program's code (blocks, statements, decisions, c-uses (computation uses), or p-uses (predicate uses)) executed by a test
  - An execution slice can be constructed very easily if we know the ***coverage*** of the test (*instead of reporting the coverage percentage, it reports which parts of the program are covered*).
  - Too much code in the slice!
- **Another Key insight:** Narrow the search domain by ***execution dices***
  - An ***execution dice*** is obtained by subtracting *successful* execution slices from *failed* execution slices

Dice = Execution slices of failed tests – Execution slices of successful tests

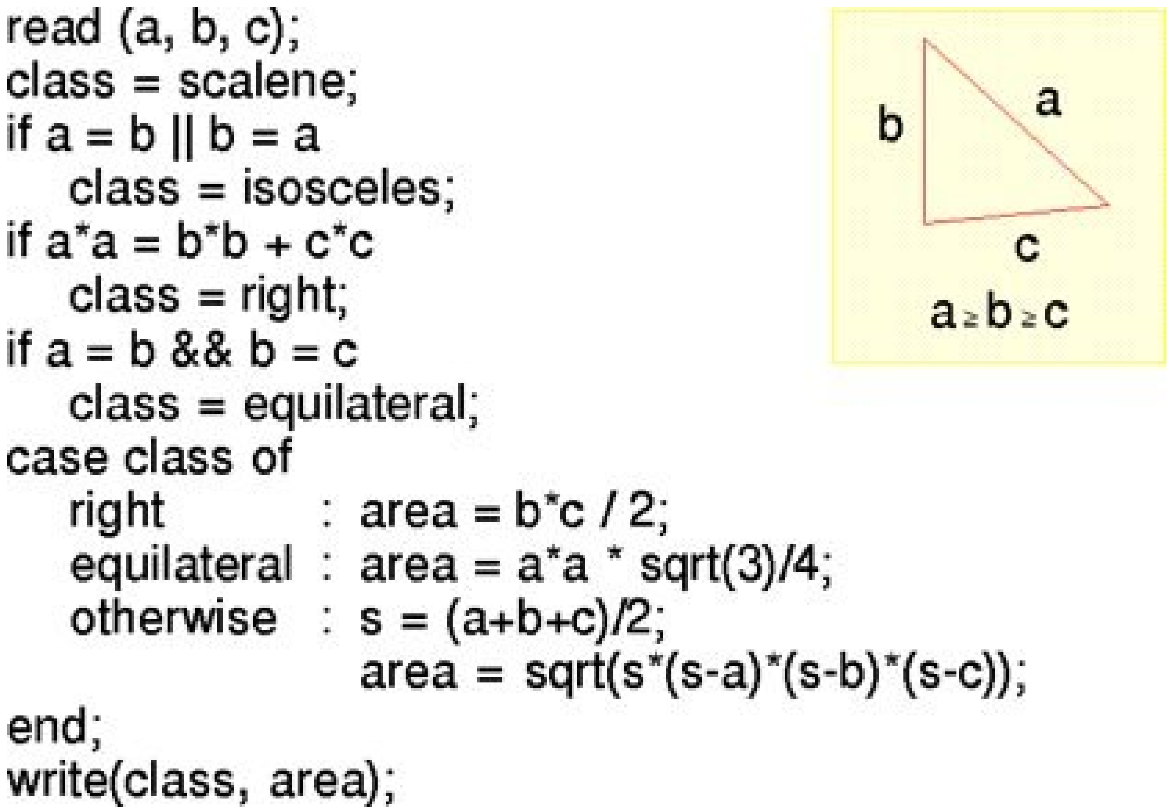
Execution Slice & Dice Example

```
read (a, b, c);
class = scalene;
if a = b || b = a
    class = isosceles;
if a*a = b*b + c*c
    class = right;
if a = b && b = c
    class = equilateral;
case class of
    right        :  area = b*c / 2;
    equilateral  :  area = a*a * sqrt(3)/4;
    otherwise    :  s = (a+b+c)/2;
                    area = sqrt(s*(s-a)*(s-b)*(s-c));
end;
write(class, area);
```

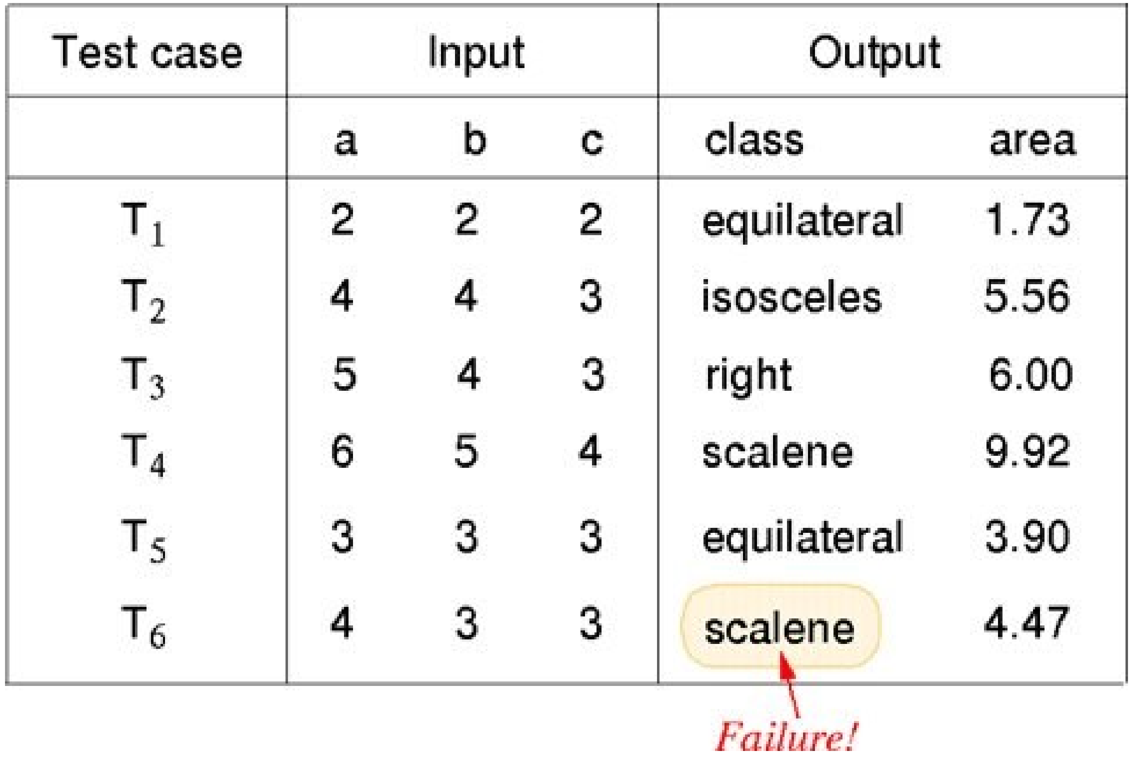


Test case	Input			Output	
	a	b	c	class	area
T_1	2	2	2	equilateral	1.73
T_2	4	4	3	isosceles	5.56
T_3	5	4	3	right	6.00
T_4	6	5	4	scalene	9.92
T_5	3	3	3	equilateral	3.90

Execution Slice & Dice Example



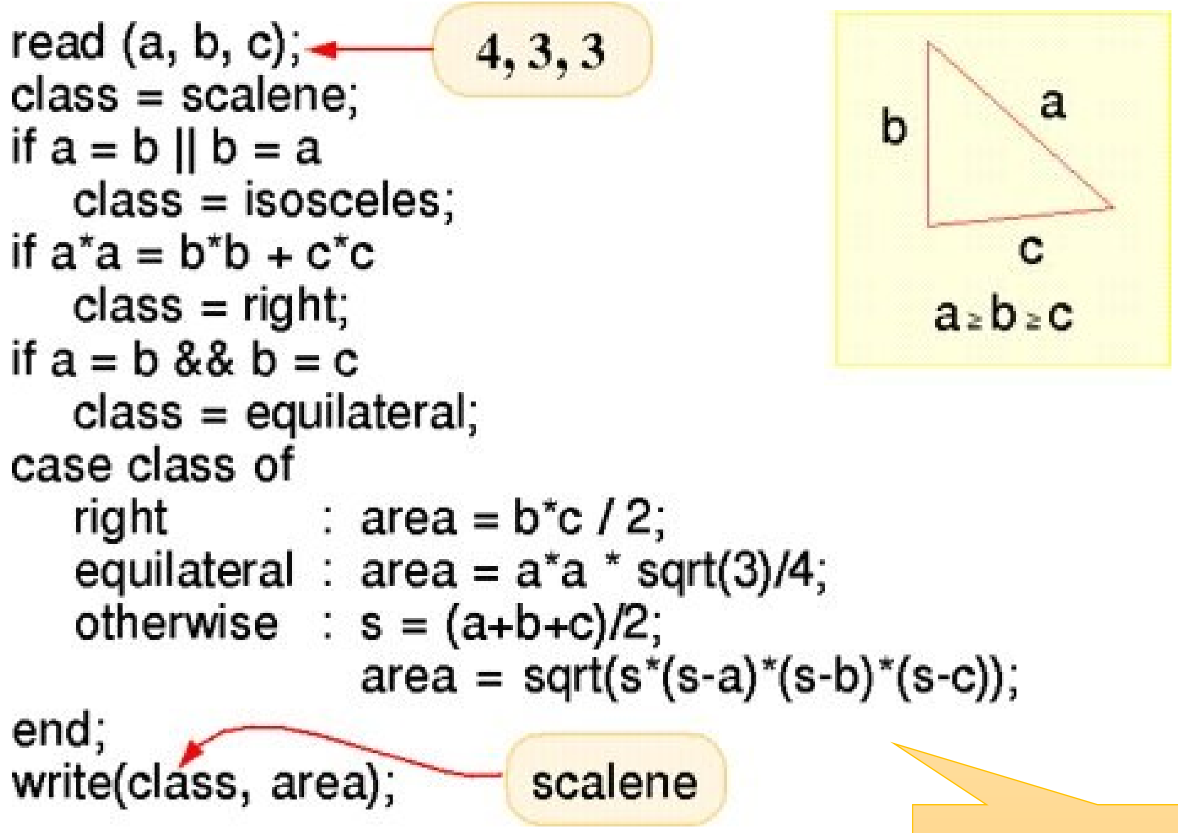
```
read (a, b, c);
class = scalene;
if a = b || b = a
    class = isosceles;
if a*a = b*b + c*c
    class = right;
if a = b && b = c
    class = equilateral;
case class of
    right       :  area = b*c / 2;
    equilateral :  area = a*a * sqrt(3)/4;
    otherwise   :  s = (a+b+c)/2;
                   area = sqrt(s*(s-a)*(s-b)*(s-c));
end;
write(class, area);
```



Test case	Input			Output	
	a	b	c	class	area
T_1	2	2	2	equilateral	1.73
T_2	4	4	3	isosceles	5.56
T_3	5	4	3	right	6.00
T_4	6	5	4	scalene	9.92
T_5	3	3	3	equilateral	3.90
T_6	4	3	3	scalene	4.47

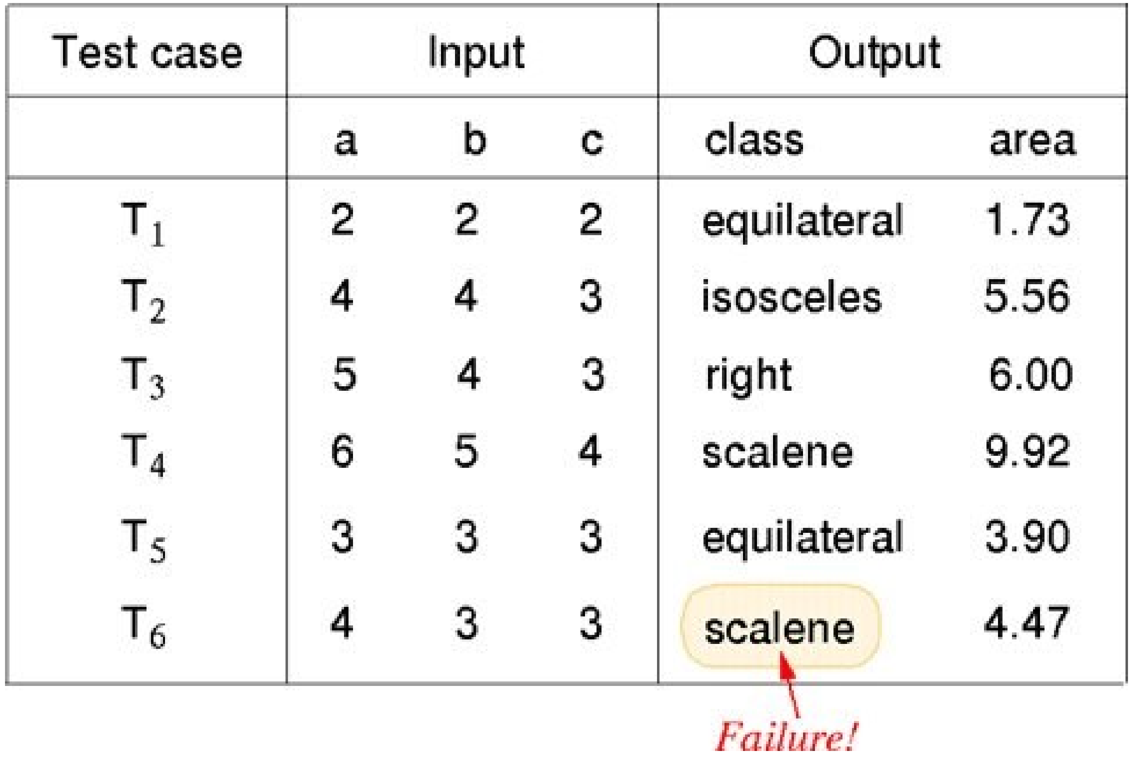
Failure!

Execution Slice & Dice Example



```
read (a, b, c);
class = scalene;
if a = b || b = a
    class = isosceles;
if a*a = b*b + c*c
    class = right;
if a = b && b = c
    class = equilateral;
case class of
    right       : area = b*c / 2;
    equilateral : area = a*a * sqrt(3)/4;
    otherwise   : s = (a+b+c)/2;
                  area = sqrt(s*(s-a)*(s-b)*(s-c));
end;
write(class, area);
```

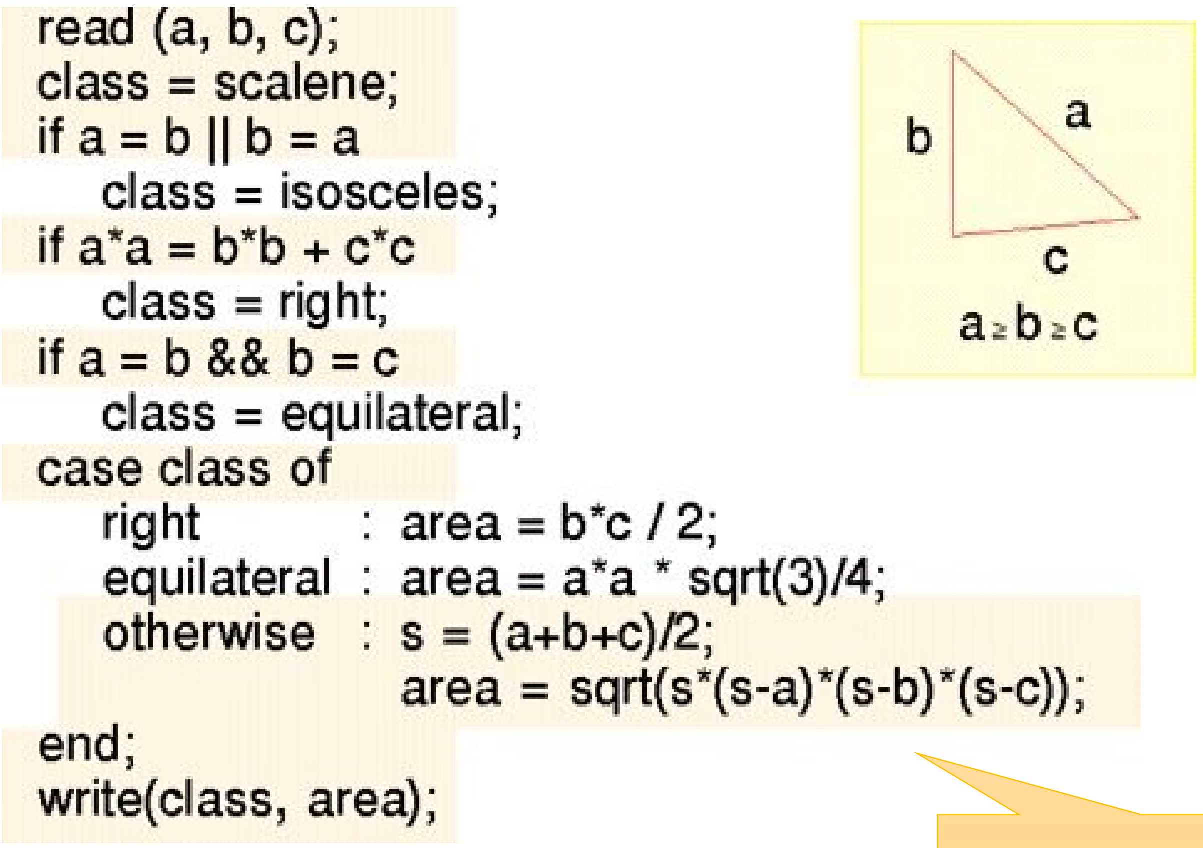
4, 3, 3
scalene
b
a
c
a ≥ b ≥ c



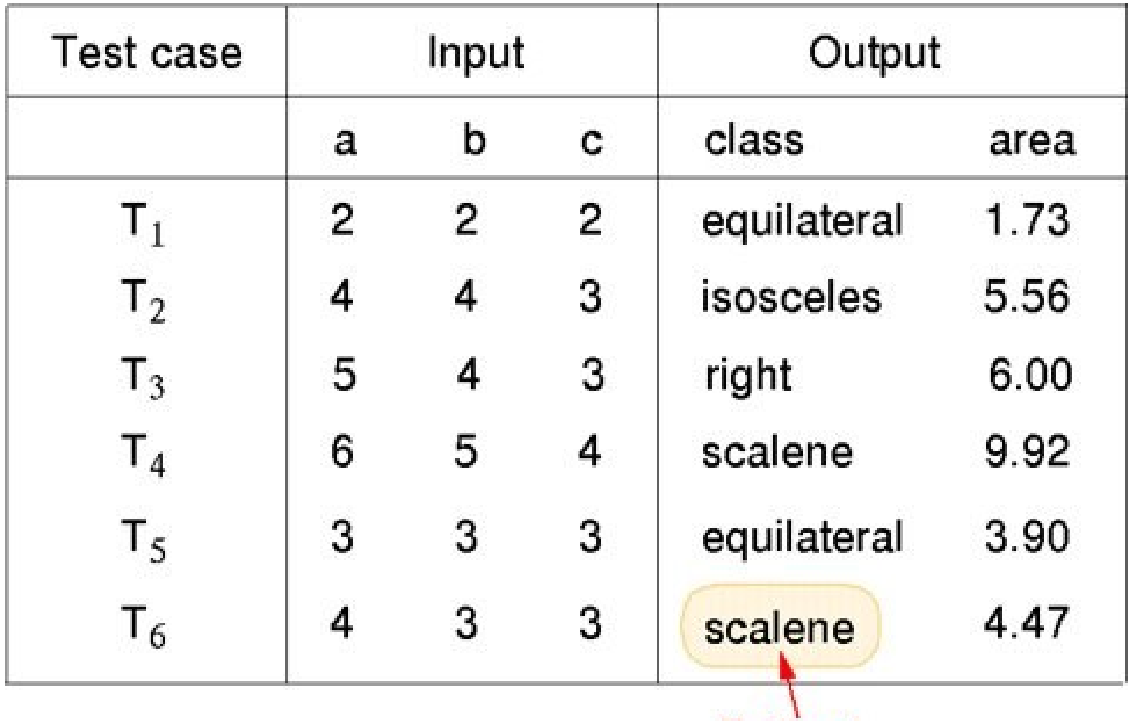
Test case	Input			Output	
	a	b	c	class	area
T_1	2	2	2	equilateral	1.73
T_2	4	4	3	isosceles	5.56
T_3	5	4	3	right	6.00
T_4	6	5	4	scalene	9.92
T_5	3	3	3	equilateral	3.90
T_6	4	3	3	scalene	4.47

Can you identify the bug?

Execution Slice & Dice Example



```
read (a, b, c);
class = scalene;
if a = b || b = a
    class = isosceles;
if a*a = b*b + c*c
    class = right;
if a = b && b = c
    class = equilateral;
case class of
    right        :  area = b*c / 2;
    equilateral  :  area = a*a * sqrt(3)/4;
    otherwise    :  s = (a+b+c)/2;
                    area = sqrt(s*(s-a)*(s-b)*(s-c));
end;
write(class, area);
```



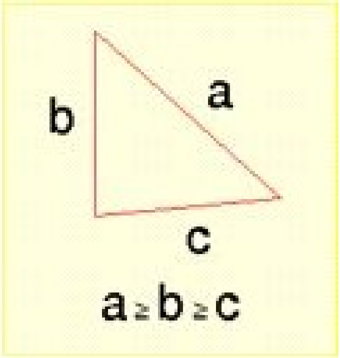
Test case	Input			Output	
	a	b	c	class	area
T_1	2	2	2	equilateral	1.73
T_2	4	4	3	isosceles	5.56
T_3	5	4	3	right	6.00
T_4	6	5	4	scalene	9.92
T_5	3	3	3	equilateral	3.90
T_6	4	3	3	scalene	4.47

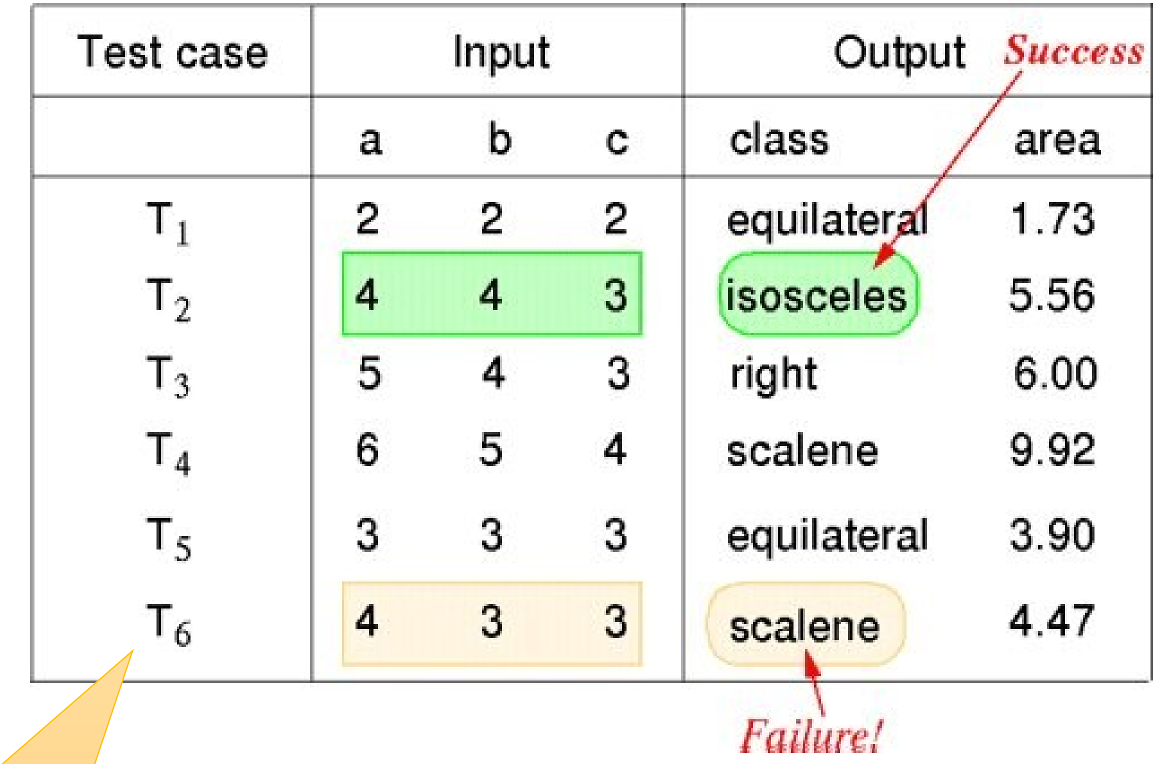
Failure!

Too much code needs to be examined!

Execution Slice & Dice Example

```
read (a, b, c);
class = scalene;
if a = b || b = a
     class = isosceles;
if a*a = b*b + c*c
     class = right;
if a = b && b = c
     class = equilateral;
case class of
     right          :  area = b*c / 2;
     equilateral    :  area = a*a * sqrt(3)/4;
     otherwise      :  s = (a+b+c)/2;
                       area = sqrt(s*(s-a)*(s-b)*(s-c));
end;
write(class, area);
```





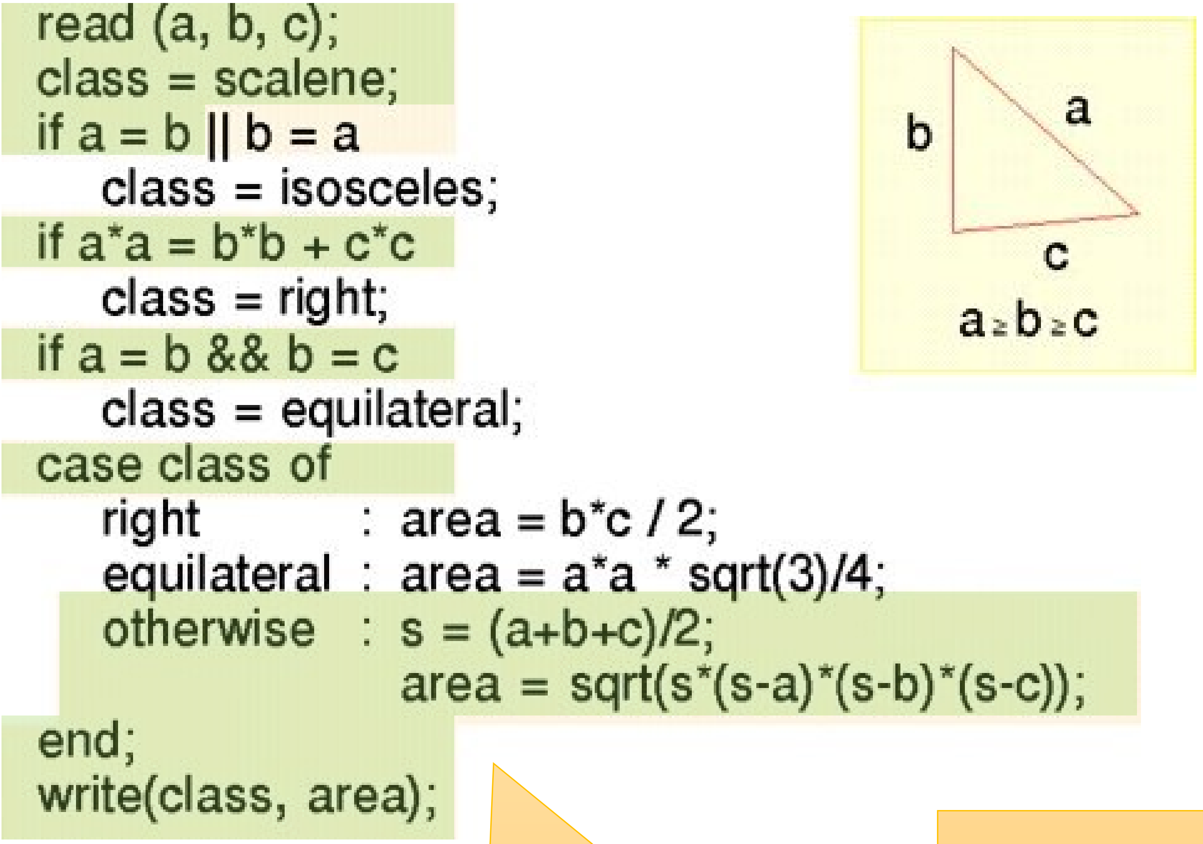
Test case	Input			Output	
	a	b	c	class	area
T_1	2	2	2	equilateral	1.73
T_2	4	4	3	isosceles	5.56
T_3	5	4	3	right	6.00
T_4	6	5	4	scalene	9.92
T_5	3	3	3	equilateral	3.90
T_6	4	3	3	scalene	4.47

Success

Failure!

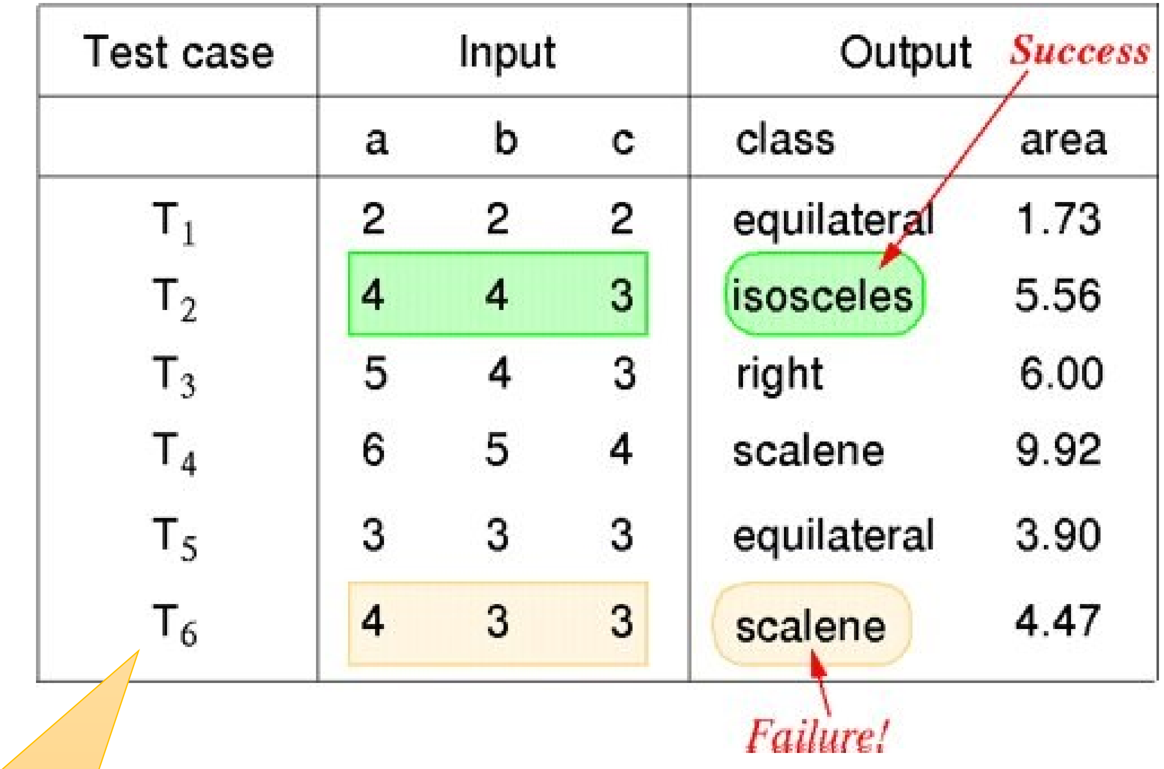
A Passed Test T_2 and a Failed Test T_6

Execution Slice & Dice Example



```
read (a, b, c);
class = scalene;
if a = b || b = a
    class = isosceles;
if a*a = b*b + c*c
    class = right;
if a = b && b = c
    class = equilateral;
case class of
    right       : area = b*c / 2;
    equilateral : area = a*a * sqrt(3)/4;
    otherwise   : s = (a+b+c)/2;
                  area = sqrt(s*(s-a)*(s-b)*(s-c));
end;
write(class, area);
```

Execution slices for Passed Test T_2 and a Failed Test T_6



Test case	Input			Output	
	a	b	c	class	area
T_1	2	2	2	equilateral	1.73
T_2	4	4	3	isosceles	5.56
T_3	5	4	3	right	6.00
T_4	6	5	4	scalene	9.92
T_5	3	3	3	equilateral	3.90
T_6	4	3	3	scalene	4.47

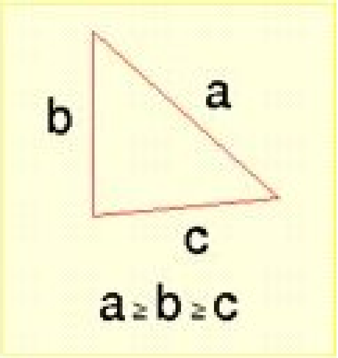
Success
Failure!

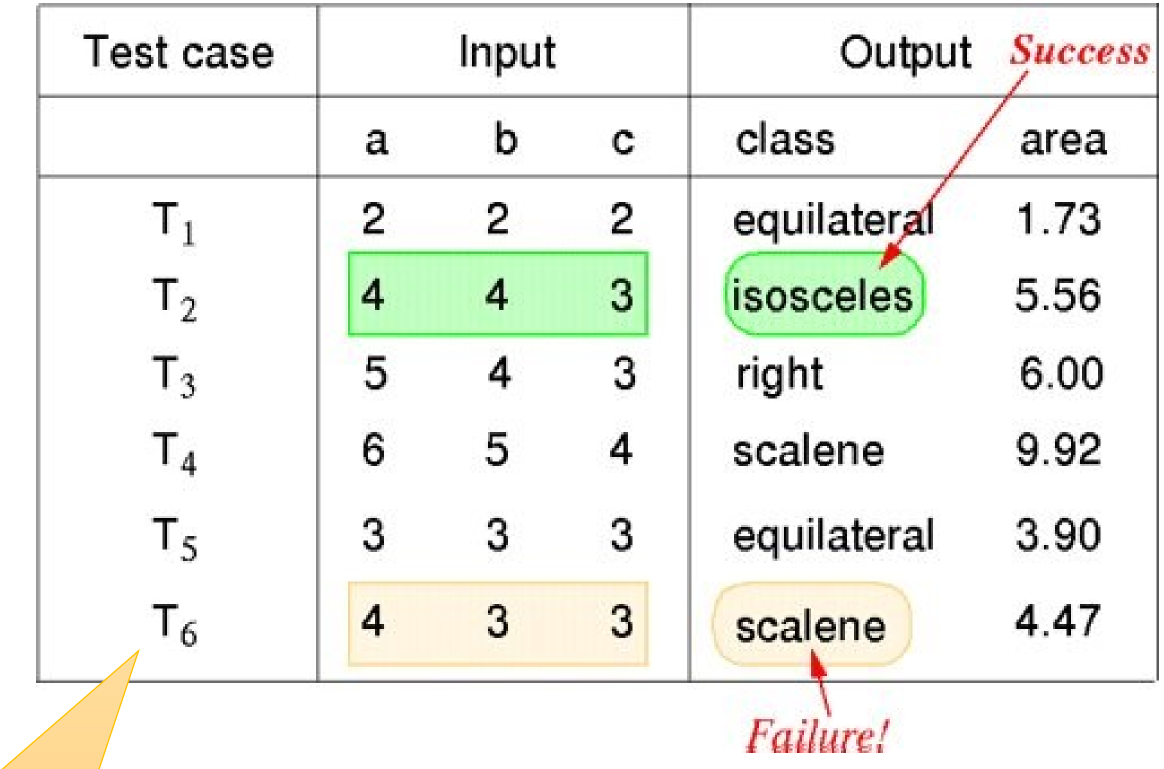
A Passed Test T_2 and a Failed Test T_6

Execution Slice & Dice Example

```
read (a, b, c);
class = scalene;
if a = b || b = a
    class = isosceles;
if a*a = b*b + c*c
    class = right;
if a = b && b = c
    class = equilateral;
case class of
    right       : area = b*c / 2;
    equilateral : area = a*a * sqrt(3)/4;
    otherwise   : s = (a+b+c)/2;
                  area = sqrt(s*(s-a)*(s-b)*(s-c));
end;
write(class, area);
```

Execution dice (suspicious program element(s))
Bug:
Should be **b = c**





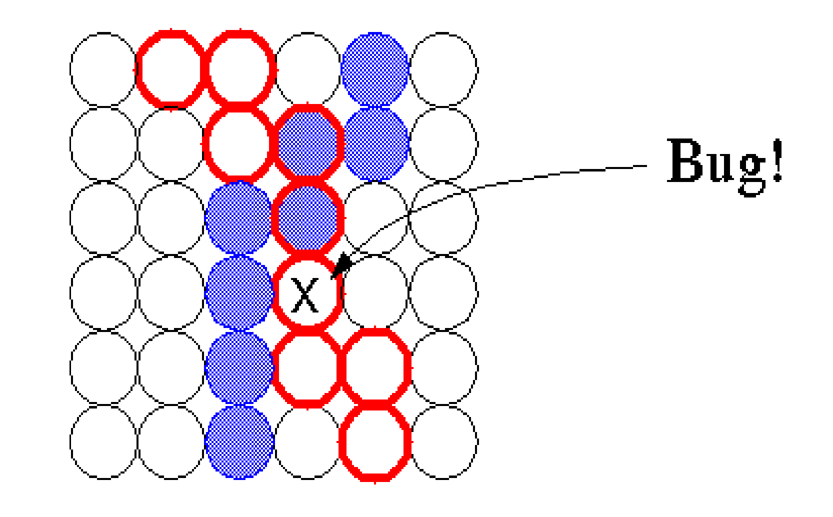
Test case	Input			Output	
	a	b	c	class	area
T_1	2	2	2	equilateral	1.73
T_2	4	4	3	isosceles	5.56
T_3	5	4	3	right	6.00
T_4	6	5	4	scalene	9.92
T_5	3	3	3	equilateral	3.90
T_6	4	3	3	scalene	4.47

Success
Failure!

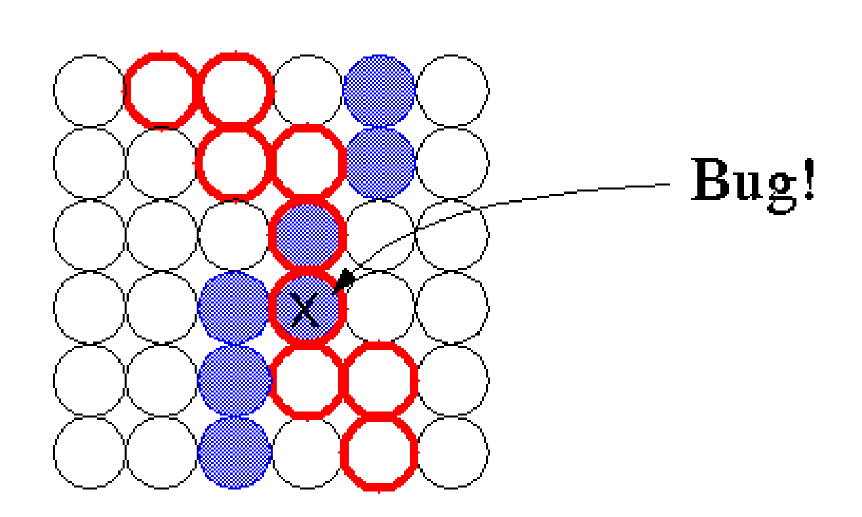
A Passed Test T_2 and a Failed Test T_6

Execution Slice & Dice Limitations

- Buggy code is in the **execution dice** (top priority)
- A bug is in the failed execution slice (the red path) but not in the successful execution slice (the blue path)

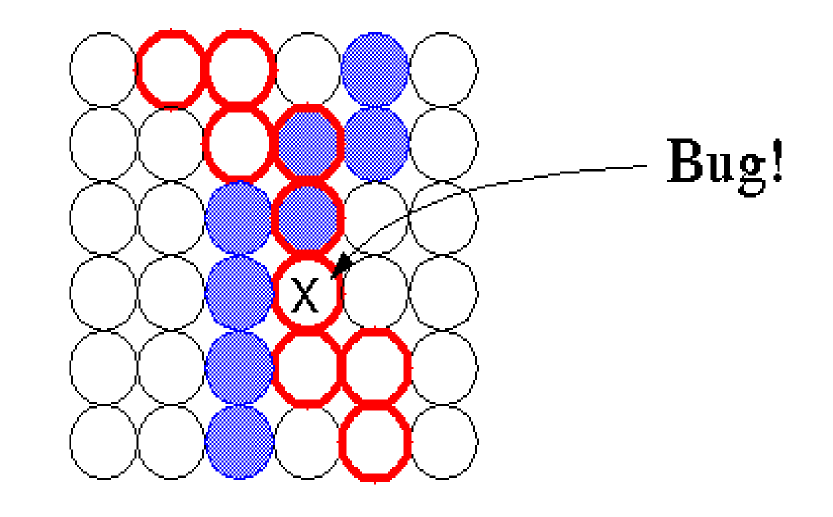


- Buggy code is in the **failed execution slice but not in the dice**
- A bug is in the failed execution slice (the red path) and in the successful execution slice (the blue path)

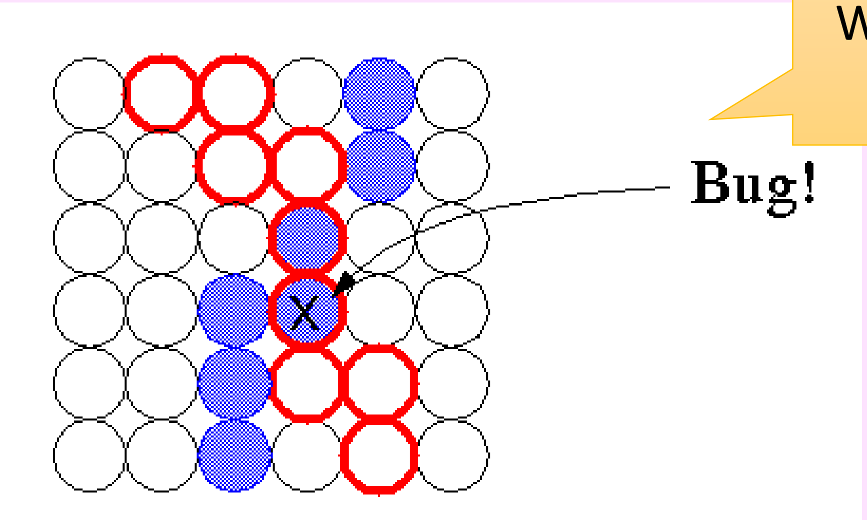


Execution Slice & Dice Limitations

- Buggy code is in the **execution dice** (top priority)
- A bug is in the failed execution slice (the red path) but not in the successful execution slice (the blue path)



- Buggy code is in the **failed execution slice but not in the dice**
- A bug is in the failed execution slice (the red path) and in the successful execution slice (the blue path)

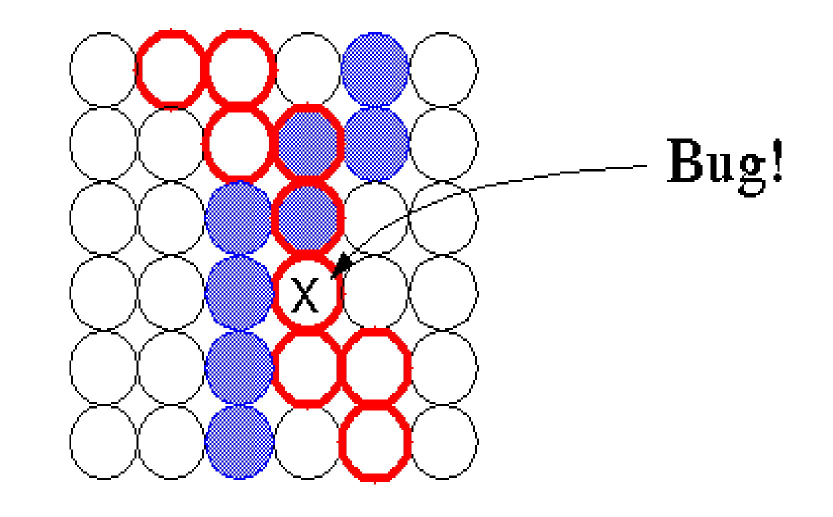


Wont work in such cases!

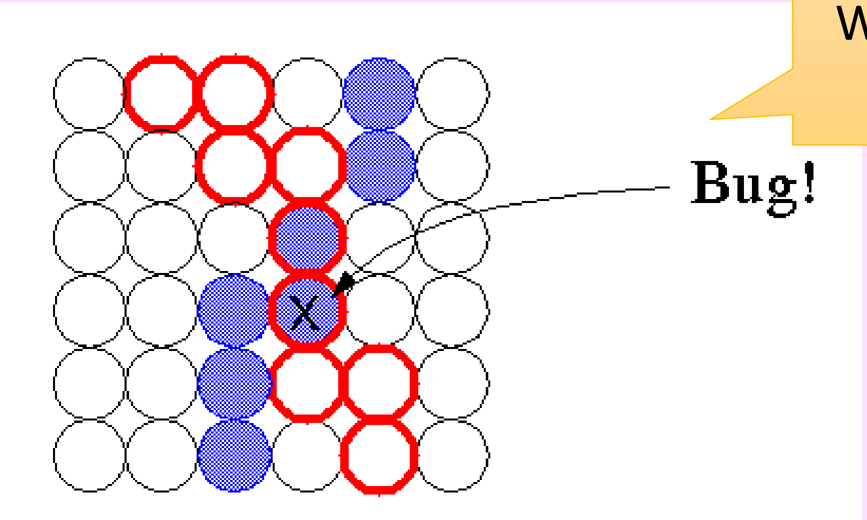
H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault localization using execution slices and dataflow tests," IEEE International Symposium on Software Reliability Engineering, pp. 143-151, Toulouse, France, October 1995.

Execution Slice & Dice Limitations

- Buggy code is in the **execution dice** (top priority)
- A bug is in the failed execution slice (the red path) but not in the successful execution slice (the blue path)



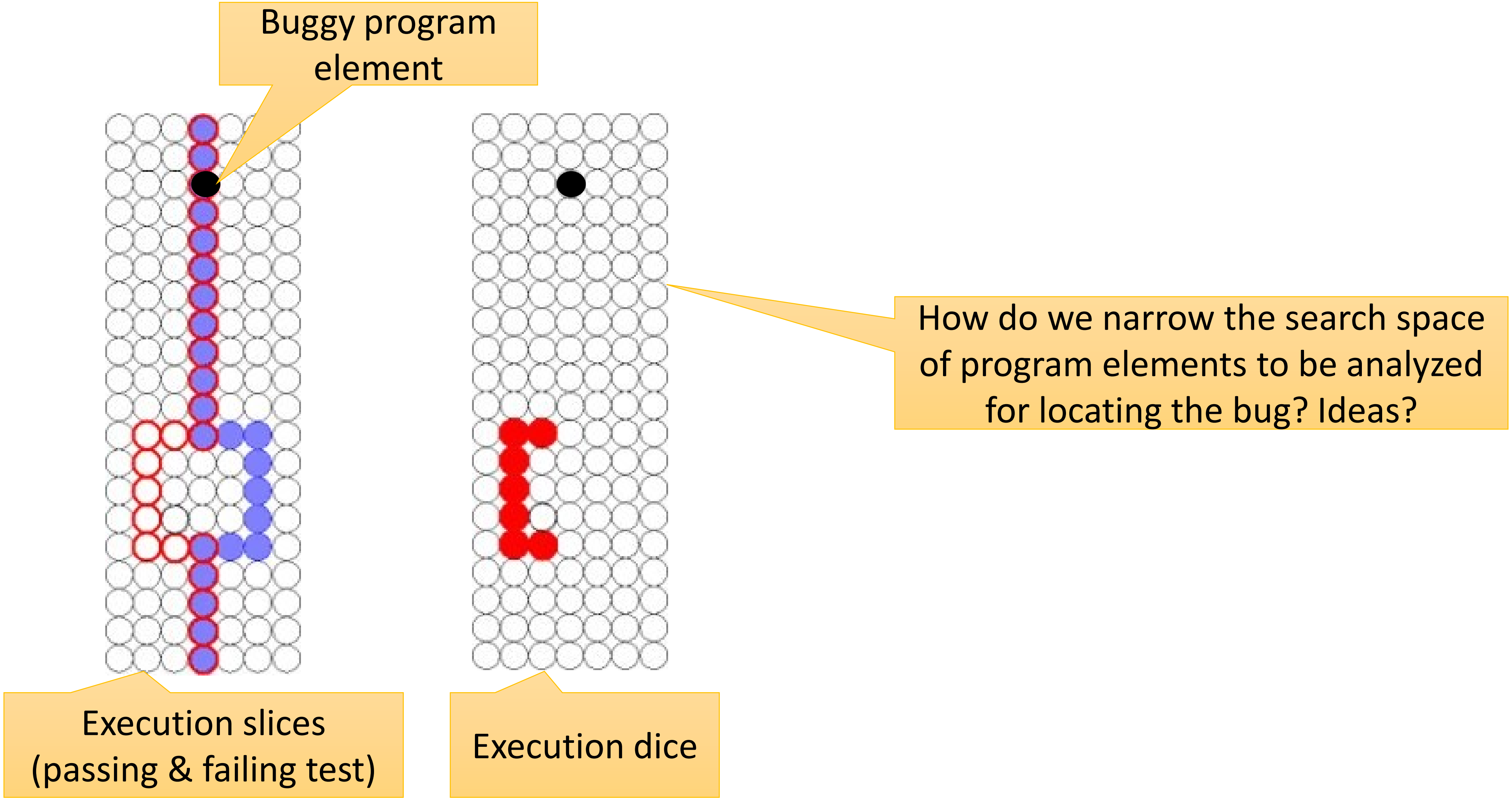
- Buggy code is in the **failed execution slice but not in the dice**
- A bug is in the failed execution slice (the red path) and in the successful execution slice (the blue path)



Wont work in such cases!

H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, “Fault localization using execution slices and dataflow tests,” IEEE International Symposium on Software Reliability Engineering, pp. 143-151, Toulouse, France, October 1995.

Execution Slice & Dice Limitations



Buggy program element

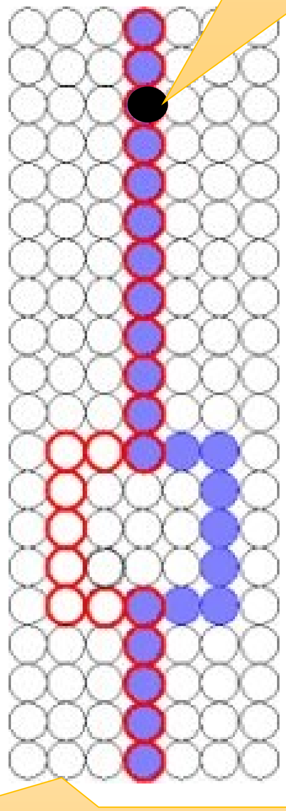
How do we narrow the search space of program elements to be analyzed for locating the bug? Ideas?

Execution slices (passing & failing test)

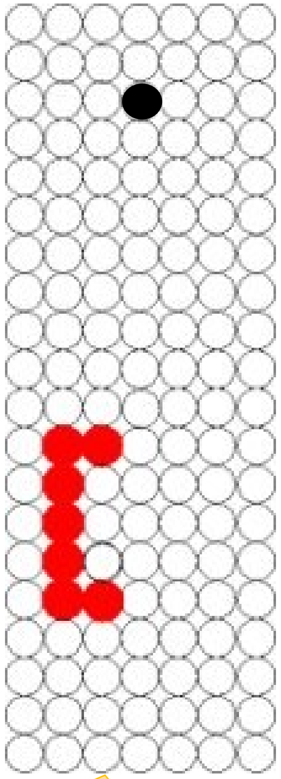
Execution dice

Execution Slice & Dice Extension

Buggy program element

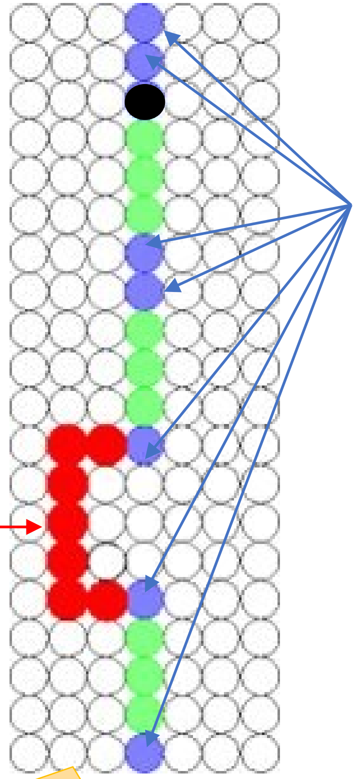


Execution slices (passing & failing test)



Execution dice

Code in red is in the execution dice

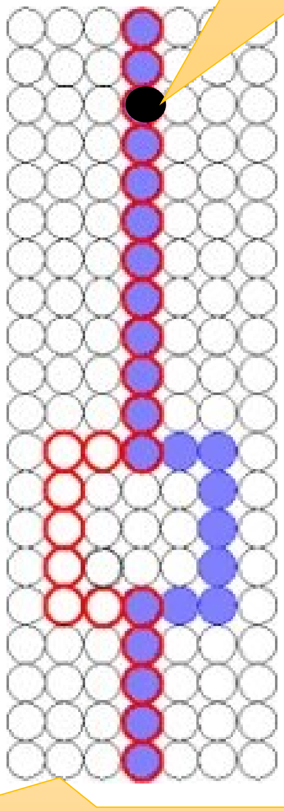


Code in blue has some data dependency with code in red
➔ *Higher Priority*

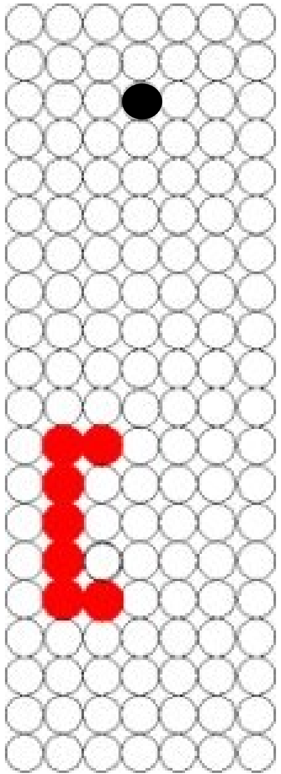
Use program analysis to augment dice (e.g., use data dependency)

Execution Slice & Dice Extension

Buggy program element

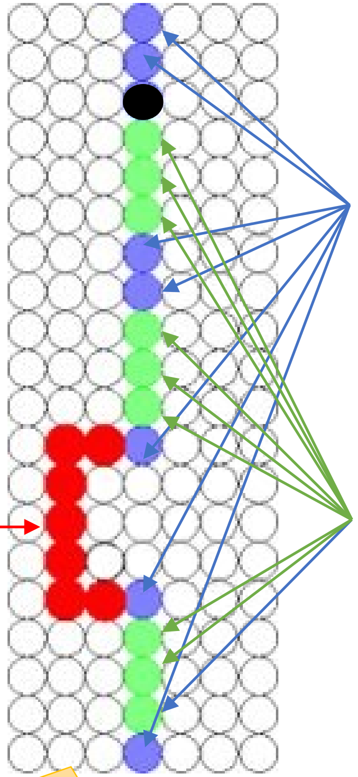


Execution slices (passing & failing test)



Execution dice

Code in red is in the execution dice



Code in blue has some data dependency with code in red ➔ *Higher Priority*

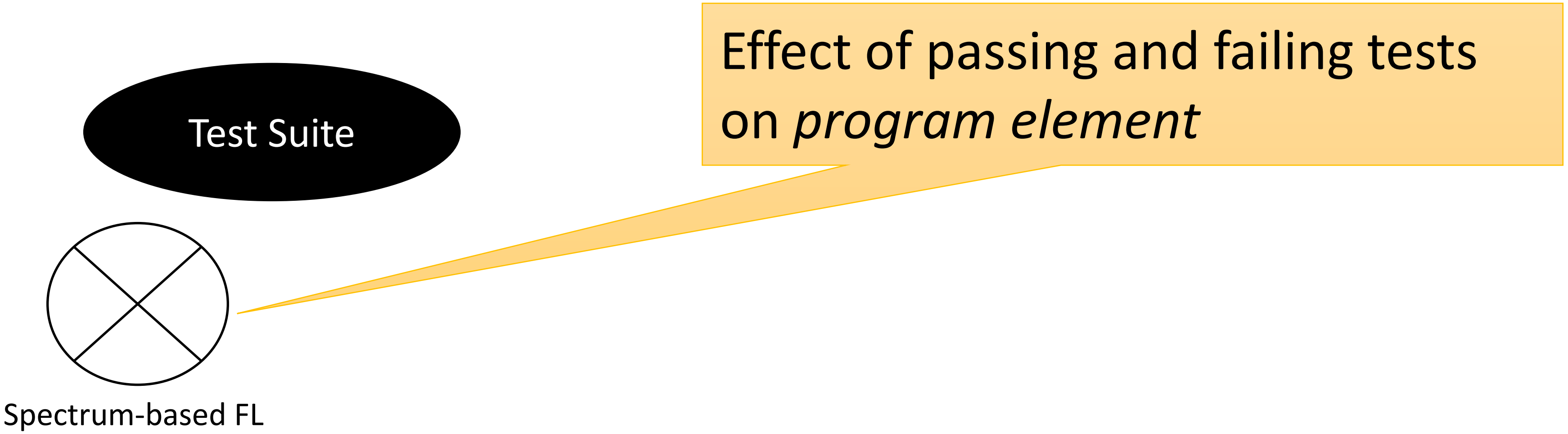
Code in green has no data dependency with code in red ➔ *Lower Priority*

Use program analysis to augment dice (e.g., use data dependency)

Suspicious Ranking Based FL

- Compute the **suspiciousness** (likelihood of containing bug) of each program element
- Rank all the executable program elements in **descending** order of their suspiciousness
- Examine the elements one-by-one from the top of the ranking until the first faulty element is located
- Elements with higher suspiciousness should be examined before elements with lower suspiciousness as the former are more likely to contain bugs than the latter

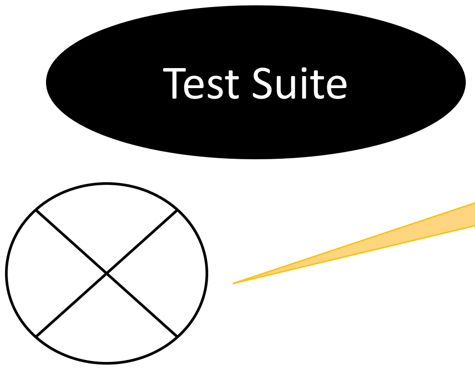
Techniques for computing suspiciousness



Effect of passing and failing tests on *program element*

Techniques for computing suspiciousness

Effect of passing and failing tests on *program element*

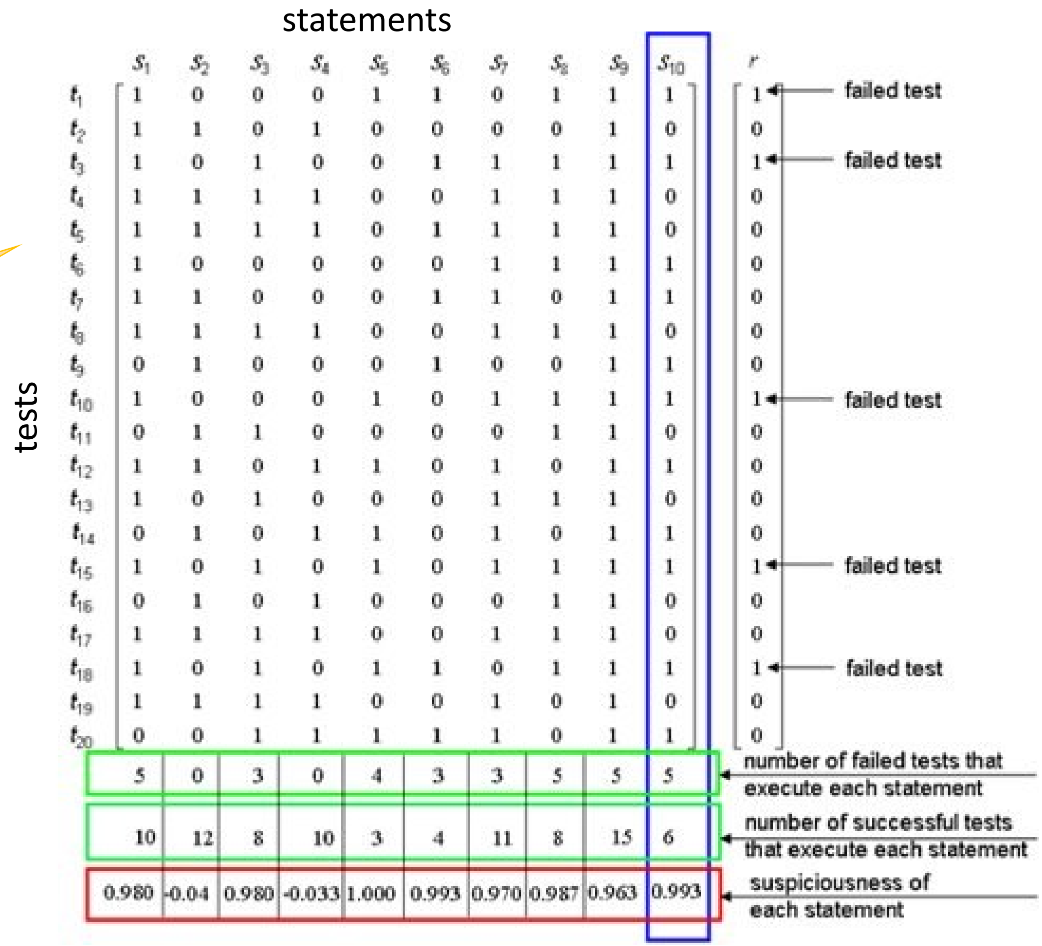


Spectrum-based FL

- The number of failing tests that **do** (e_f) and **do not** (n_f) execute the element
- The number of passing tests that do (e_p) and do not (n_p) execute the element.
- ***Ochiai***

$$score = \frac{e_f}{\sqrt{(e_f+n_f)(e_f+e_p)}}$$

Example of ***program spectra*** (coverage matrix)



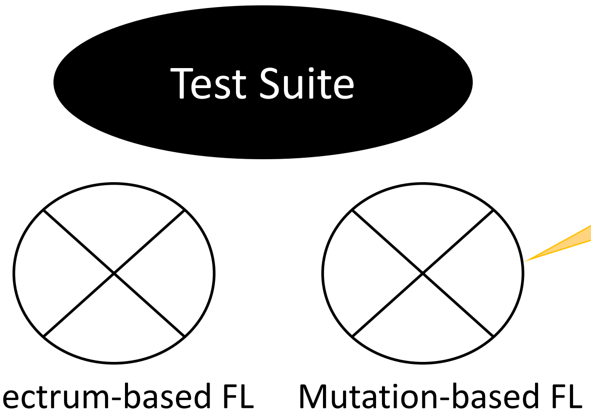
statements

tests

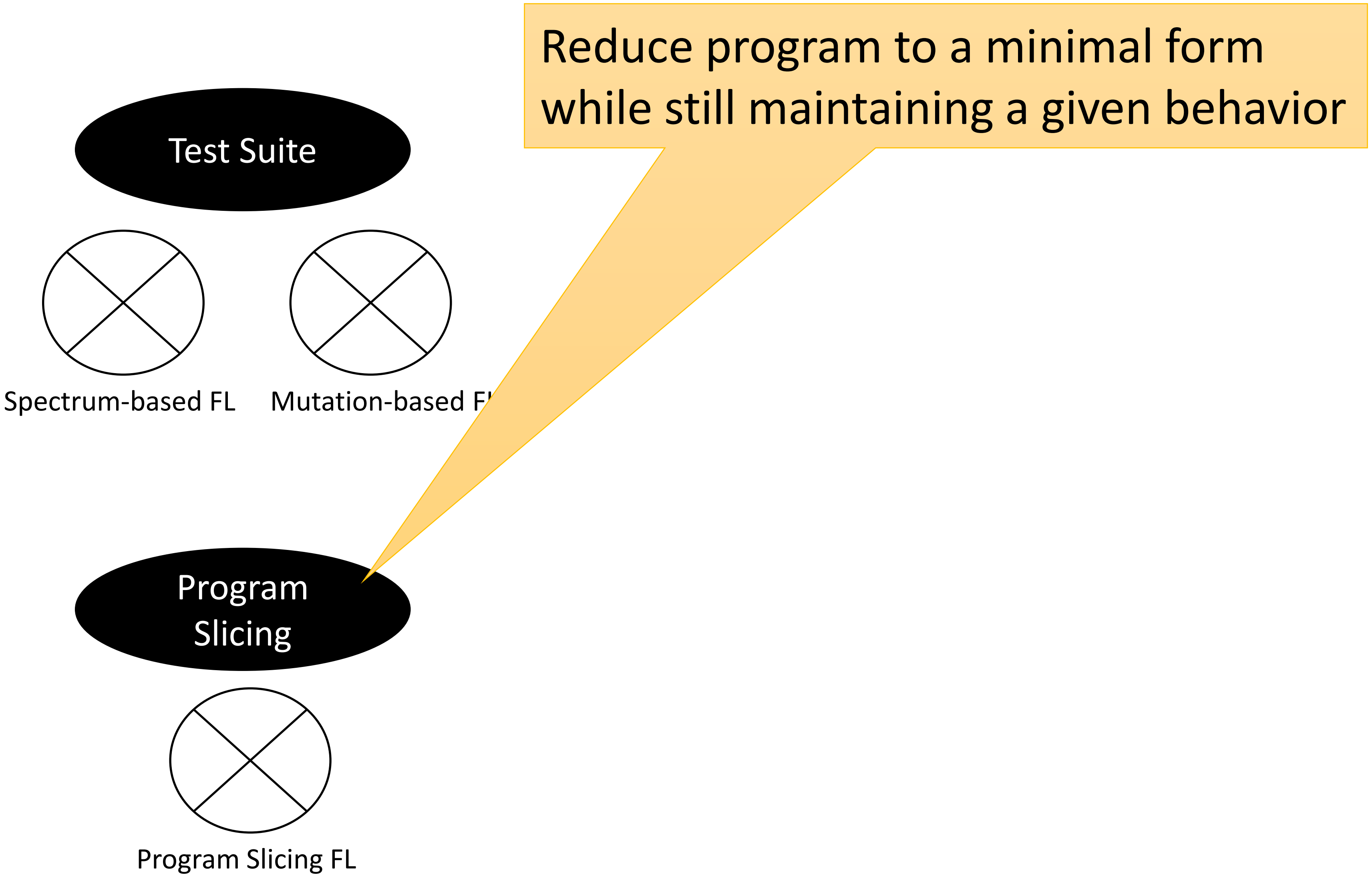
	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	r	
t_1	1	0	0	0	1	1	0	1	1	1	1	failed test
t_2	1	1	0	1	0	0	0	0	1	0	0	
t_3	1	0	1	0	0	1	1	1	1	1	1	failed test
t_4	1	1	1	1	0	0	1	1	1	0	0	
t_5	1	1	1	1	0	1	1	1	1	0	0	
t_6	1	0	0	0	0	0	1	1	1	1	0	
t_7	1	1	0	0	0	1	1	0	1	1	0	
t_8	1	1	1	1	0	0	1	1	1	0	0	
t_9	0	1	0	0	0	1	0	0	1	1	0	
t_{10}	1	0	0	0	1	0	1	1	1	1	1	failed test
t_{11}	0	1	1	0	0	0	0	1	1	0	0	
t_{12}	1	1	0	1	1	0	1	0	1	1	0	
t_{13}	1	0	1	0	0	0	1	1	1	0	0	
t_{14}	0	1	0	1	1	0	1	0	1	1	0	
t_{15}	1	0	1	0	1	0	1	1	1	1	1	failed test
t_{16}	0	1	0	1	0	0	0	1	1	0	0	
t_{17}	1	1	1	1	0	0	1	1	1	0	0	
t_{18}	1	0	1	0	1	1	0	1	1	1	1	failed test
t_{19}	1	1	1	1	0	0	1	0	1	0	0	
t_{20}	0	0	1	1	1	1	1	0	1	1	0	
	5	0	3	0	4	3	3	5	5	5		number of failed tests that execute each statement
	10	12	8	10	3	4	11	8	15	6		number of successful tests that execute each statement
	0.980	-0.04	0.980	-0.033	1.000	0.993	0.970	0.987	0.963	0.993		suspiciousness of each statement

Techniques for computing suspiciousness

Effect of *mutating program element* on passing and failing tests



Techniques for computing suspiciousness



Reduce program to a minimal form while still maintaining a given behavior

Test Suite

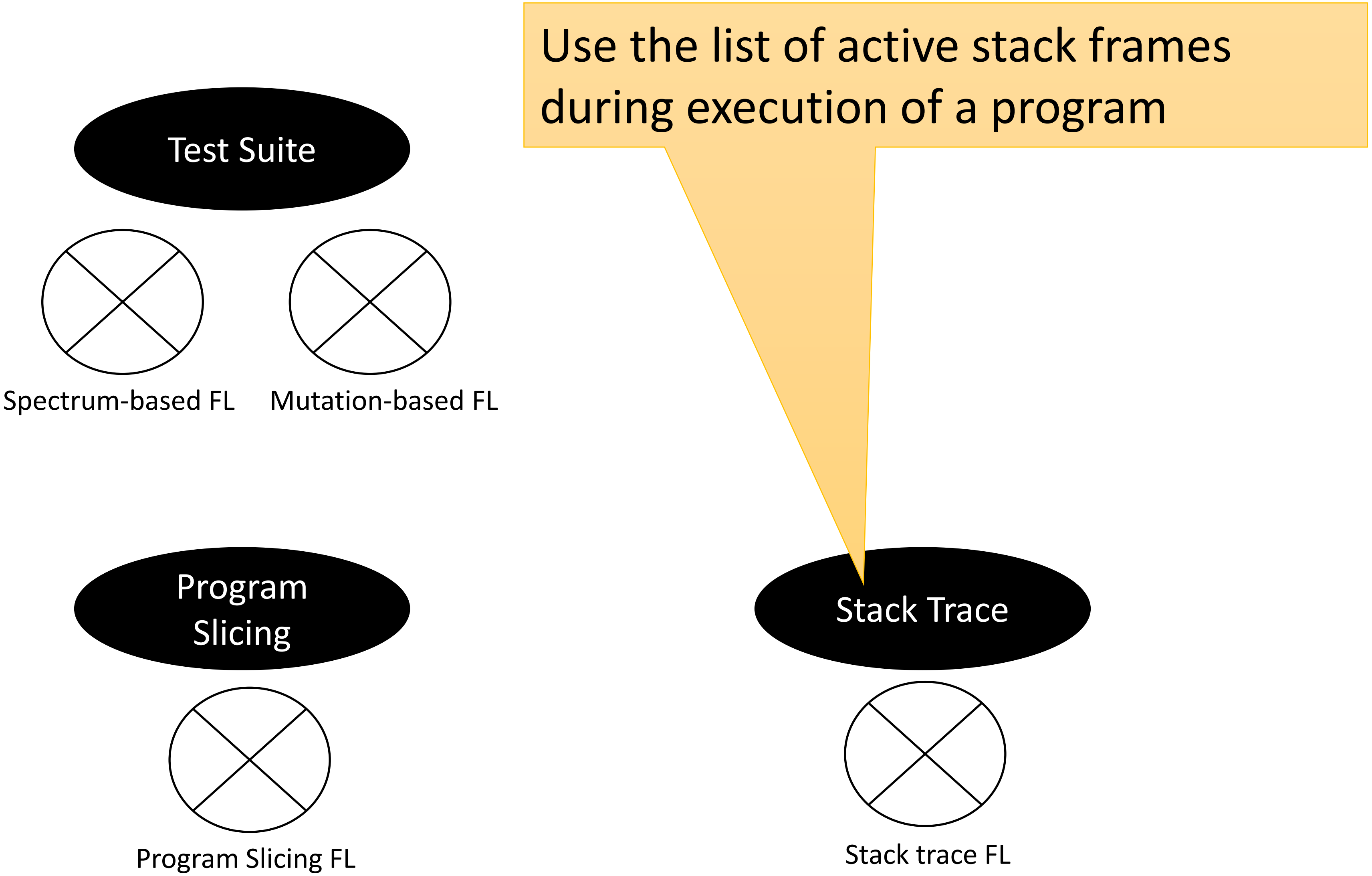
Spectrum-based FL

Mutation-based F

Program Slicing

Program Slicing FL

Techniques for computing suspiciousness



Use the list of active stack frames during execution of a program

Test Suite

Spectrum-based FL

Mutation-based FL

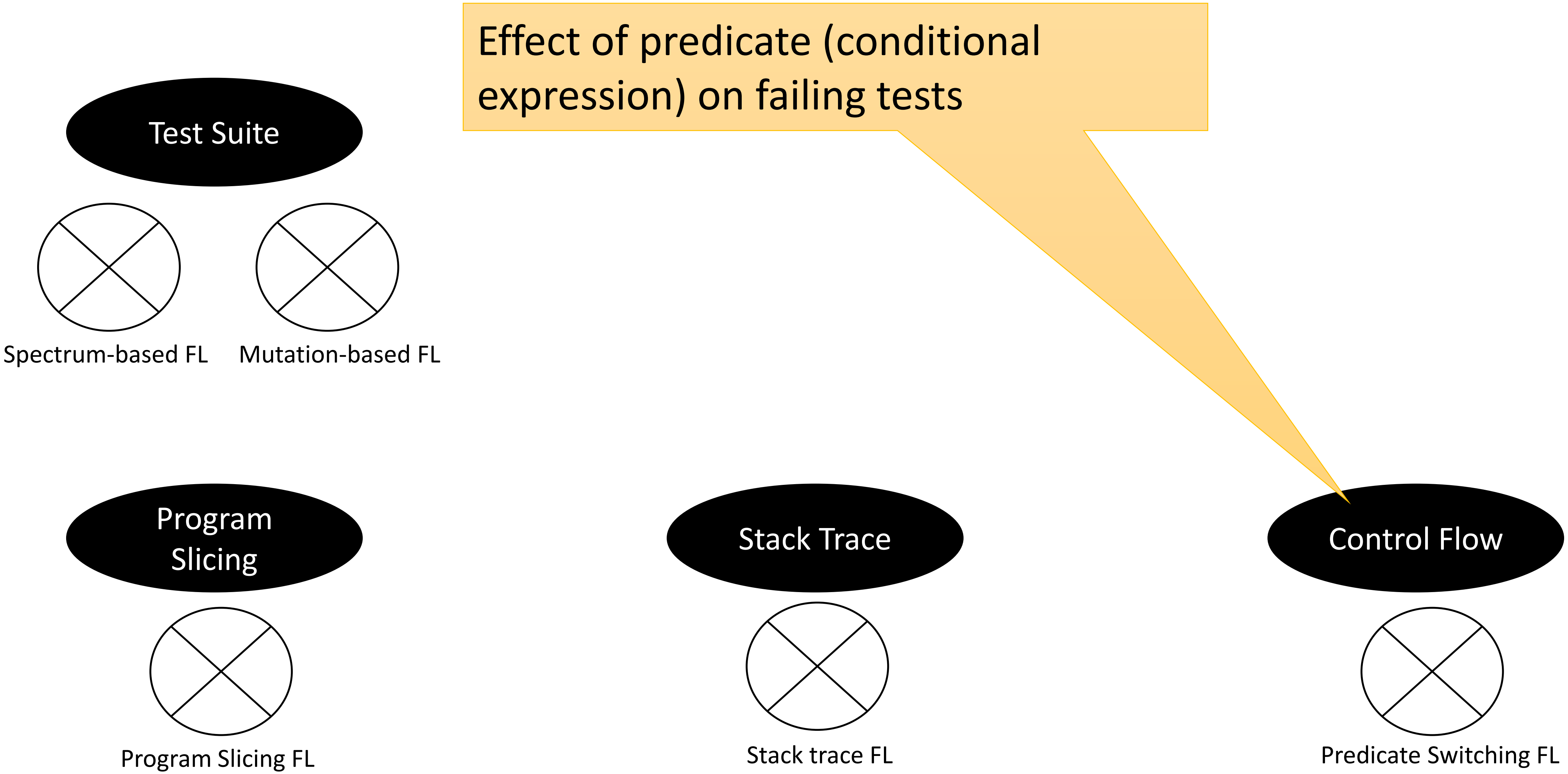
Program Slicing

Program Slicing FL

Stack Trace

Stack trace FL

Techniques for computing suspiciousness



Effect of predicate (conditional expression) on failing tests

Test Suite

Spectrum-based FL

Mutation-based FL

Program Slicing

Program Slicing FL

Stack Trace

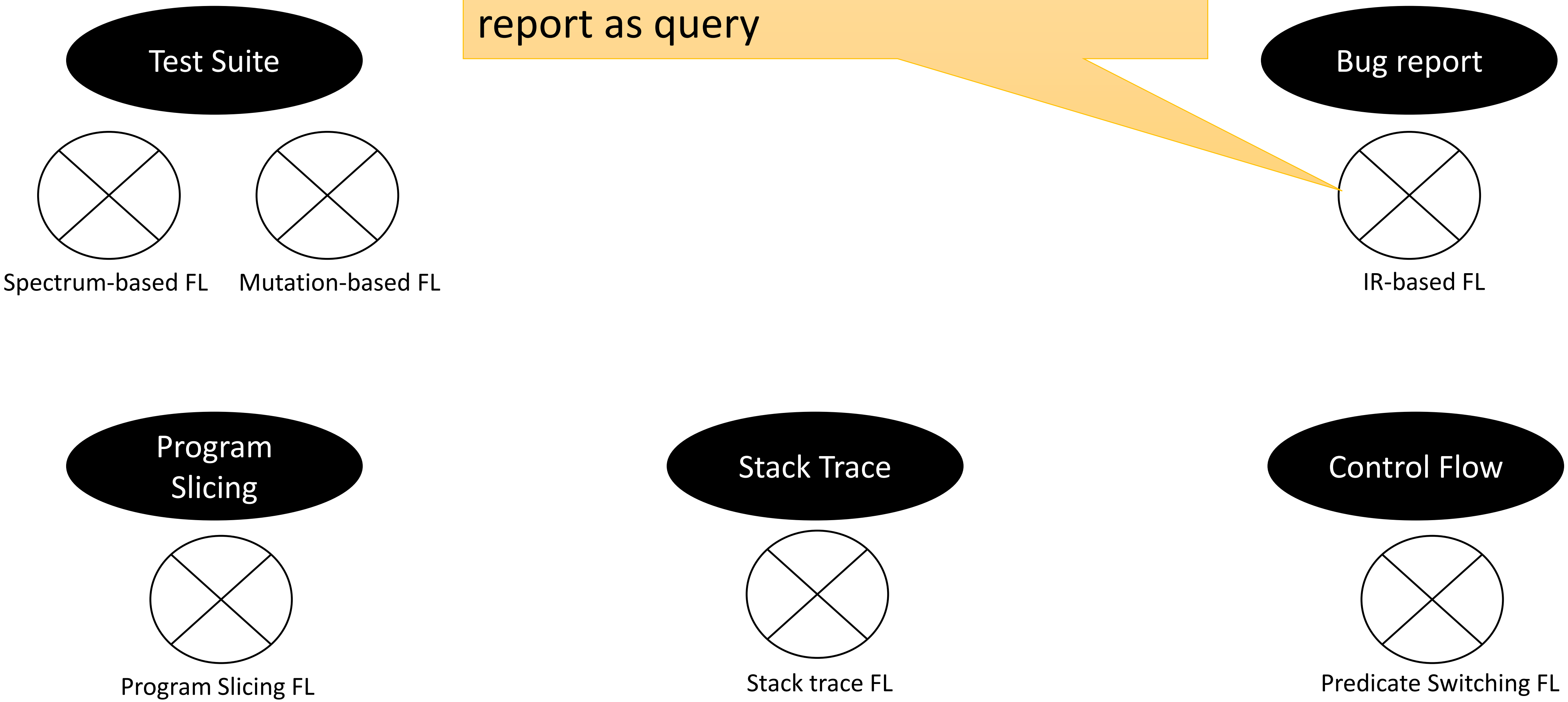
Stack trace FL

Control Flow

Predicate Switching FL

Techniques for computing suspiciousness

Rank program elements using bug report as query



Test Suite

Spectrum-based FL

Mutation-based FL

Bug report

IR-based FL

Program Slicing

Program Slicing FL

Stack Trace

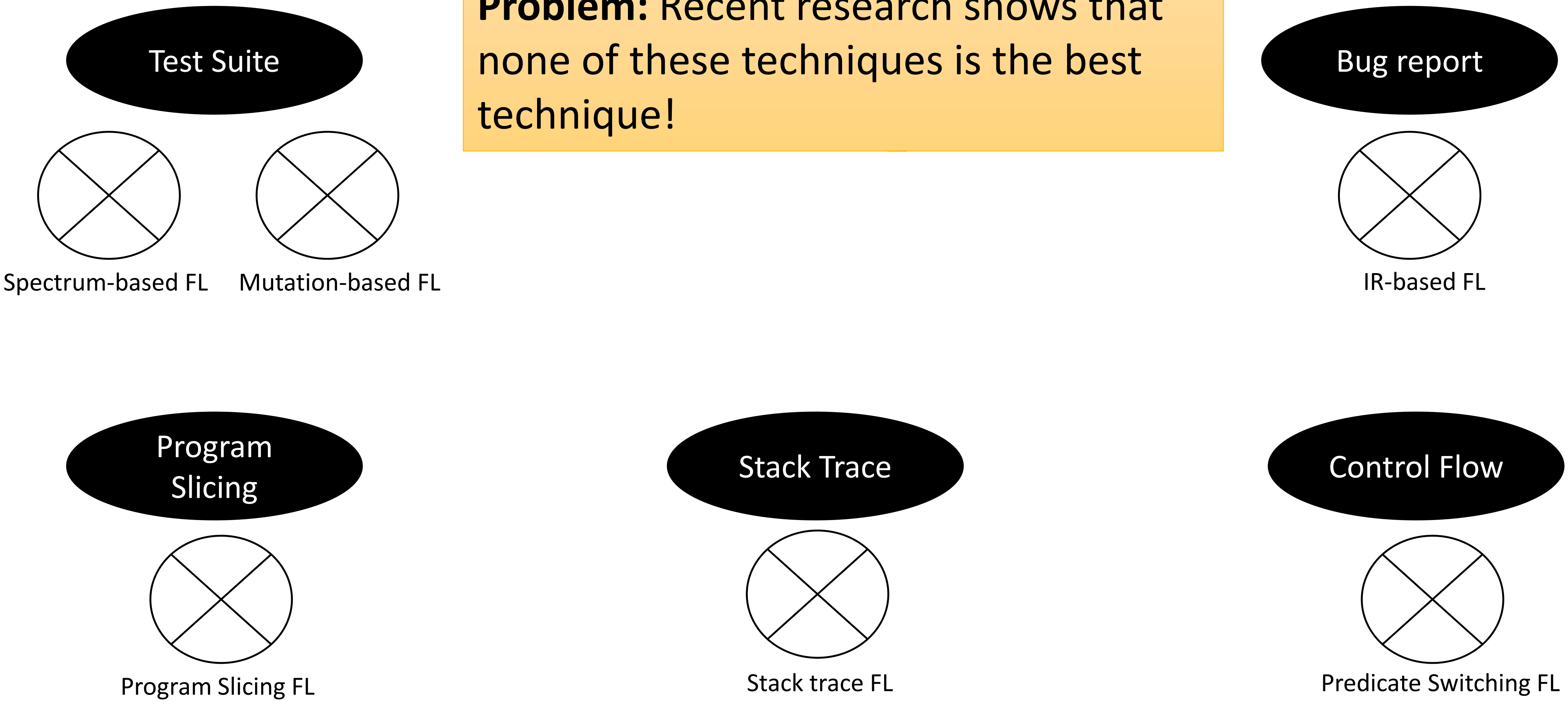
Stack trace FL

Control Flow

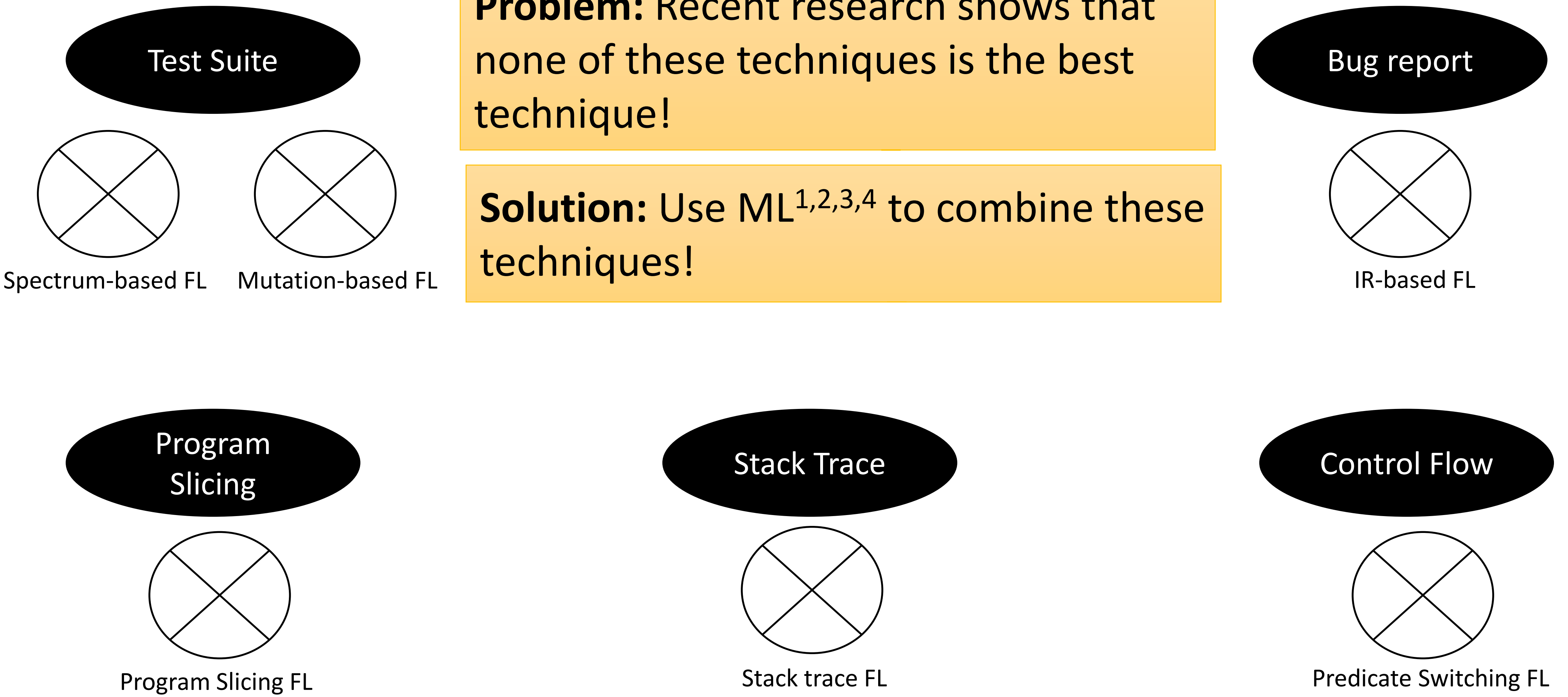
Predicate Switching FL

Techniques for computing suspiciousness

Problem: Recent research shows that none of these techniques is the best technique!



Techniques for computing suspiciousness



Problem: Recent research shows that none of these techniques is the best technique!

Solution: Use ML[1,2,3,4] to combine these techniques!

[1] DeepRL4FL (Li et al., ICSE 2021); [2] CombineFL (Zou et al., TSE, 2019); [3] NeuralBugLocator (Gupta et al., NeurIPS , 2019); [4] DeepFL (Li et al., ISSTA, 2019)

Evaluation Metrics for FL techniques

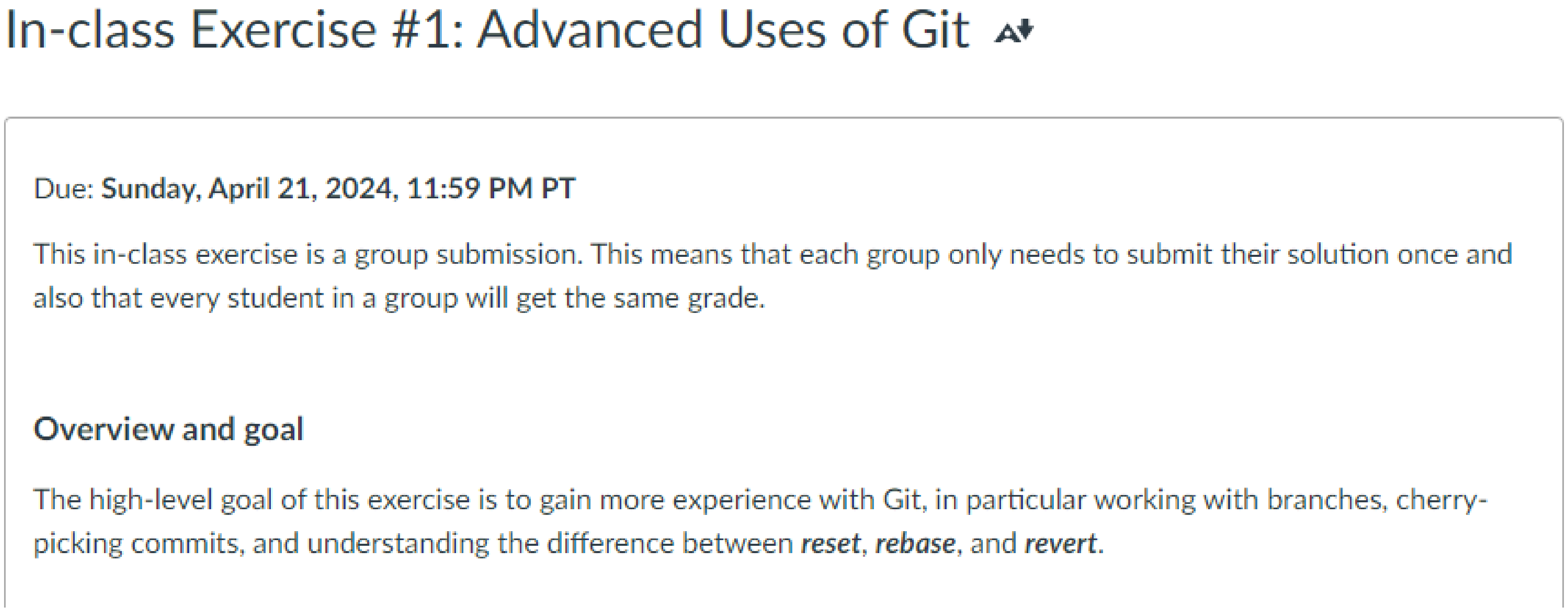
- **Precision**: proportion of reported faulty statements that are actually faulty
- **Recall (or Sensitivity)**: proportion of actually faulty statements that are identified
- **F-measure:** harmonic mean of precision and recall ((2 x P x R)/(P + R)) to evaluate overall effectiveness. Higher F-measure means better balance between precision and recall
- **Area Under the Curve (AUC):** area under the precision recall (or ROC) curve
- **Top-k accuracy:** proportion of actually faulty statements that are ranked within the top-k positions
- **Reciprocal Rank:** 1/rank of the first faulty statement found in the ranked list
- **Mean Average Precision (MAP):** Average **Precision** across all faults in a dataset
- **Mean Reciprocal Rank (MRR):** Average **Reciprocal Rank** across all faults in a dataset
- **EXAM:** fraction of ranked statements one needs to inspect before finding a faulty statement
- **Time Complexity:** How efficiently can a technique be applied to large codebases
- **Space Complexity:** Memory required to execute the technique (think about the size of coverage matrix for MLOC applications!)

Announcements

- **Paper selection due tonight!**
- Project idea proposal due next Monday
  - Try to prepare a rough sketch of the project idea to brainstorm and get feedback during the next in-class exercise on Wednesday

In-class exercise: Advanced uses of Git

- Form a team of 2, go to Canvas Assignments and work on:
- Due at the end of this week on **Sunday, April 21, 2024, at 11:59 PM**



In-class Exercise #1: Advanced Uses of Git

Due: **Sunday, April 21, 2024, 11:59 PM PT**

This in-class exercise is a group submission. This means that each group only needs to submit their solution once and also that every student in a group will get the same grade.

Overview and goal

The high-level goal of this exercise is to gain more experience with Git, in particular working with branches, cherry-picking commits, and understanding the difference between ***reset***, ***rebase***, and ***revert***.