

# CS3213 Project – Week 8

Debugging | 09-03-2022

*Based on slides/lectures by  
Abhik Roychoudhury (National University of Singapore),  
Lars Grunske (Humboldt-Universität zu Berlin), and  
Andreas Zeller (CISPA, Saarland University).*

# Debugging – History

9.9.1945  
15:45

92

9/9


0800 Action started  
1000 " stopped - action ✓

1300 (032) MP - MC { 1.2700 9.037 847 025  
2.130476415 (3) 4.615925059(-2)  
(033) PRO 2 2.130476415  
connect 2.130676415

Relays 6-2 in 033 failed special speed test  
in relay " 11.000 test.

Relays changed

1100 Started Cosine Tape (Sine check)  
1525 Started Multi-Adder Test.

1545  Relay #70 Panel F  
(moth) in relay.

First actual case of bug being found.

1630 Action started.  
1700 closed down.

- ❑ A moth in the Mark II computer causes errors in Relay No. 70, Panel F.
- ❑ Mrs. Grace Murray Hopper removes the error and documents it in the log book: "First actual case of bug being found."
- ❑ "open"-visible error!  
→ Elimination is easy!

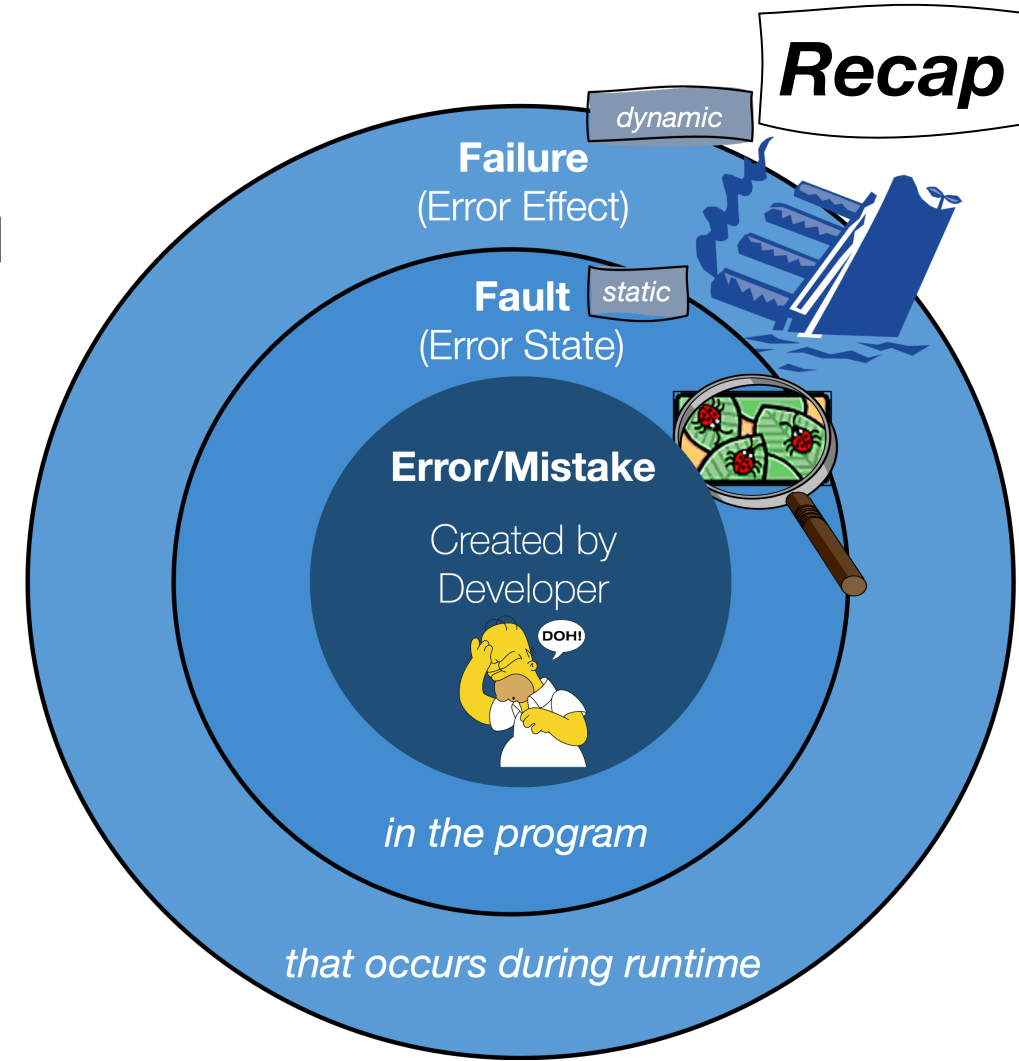
# Debugging – From Error to Failures

The issue of *debugging* is to

- ❑ *relate* an observed failure to a fault/defect and
- ❑ to *remove* the defect such that the failure no longer occurs.

Debugging Steps:

- ❑ Execution of tests!
- ❑ ***Fault Localization!***
- ❑ Identify possible fixes.
- ❑ Choose the best fix.
- ❑ Implement the best fix!



# Debugging: How is it done?

- ❑ Debugging printouts (printf(), print, output)
- ❑ Interactive debugger
  - ❑ Breakpoints
  - ❑ Single stepping
- ❑ Problems
  - ❑ Manual work
  - ❑ Many Data to observe
  - ❑ Single Steps and Executions

# Debugging Statements



Can Debugging statements be harmful?



Debugging statements can harm **maintainability** or even introduce **security vulnerabilities**!

*"A security error in OS X 10.7.3 exposes passwords on systems with support for the pre-Lion FileVault home-directory encryption feature. This security flaw, apparantly created when **Apple left debugging code** in the 10.7.3 update, is only triggered with Lion systems in which legacy support for the original FileVault is retained and when logging in with such an account."*

[https://www.macworld.com/article/217641/security\\_error\\_in\\_os\\_x\\_10\\_7\\_3\\_exposes\\_passwords\\_for\\_legacy\\_filevault\\_users.html](https://www.macworld.com/article/217641/security_error_in_os_x_10_7_3_exposes_passwords_for_legacy_filevault_users.html)

# The Devil's Guide to Debugging <sup>(1/2)</sup>

(from Code Complete by Steve McConnell)

- ❑ Find the error by **guessing**.

- ❑ Scatter print statements randomly throughout the code.

- ❑ If the print statements do not reveal the error, change code until something appears to work.

- ❑ Do not save the original version of the code and do not track changes.

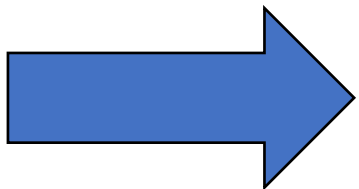
- ❑ Debugging by **superstition**.

- ❑ Why blame yourself when you can blame the computer, the operating system, the compiler, the data, other programmers (especially those ones who write library routines!), and best of all, the stupid users!

# The Devil's Guide to Debugging <sup>(2/2)</sup>

(from Code Complete by Steve McConnell)

- ❑ Don't waste time trying to **understand** the problem.
  - ❑ Why spend an hour analyzing the problem (in your head and on paper) and evaluating possible solutions or methodologies when you can spend days trying to debug your code?
- ❑ Fix the error with the **most obvious fix**.
  - ❑ For example, why try to understand why a particular case is not handled by a supposedly general subroutine when you can make a quick fix?



Obviously, not the best approaches and ideas!



# Debugging – Difficulties

- ❑ Symptom and failure cause can be far apart.
- ❑ Symptoms of one error may be hidden by other errors.
  - ❑ Fault masking: “An occurrence in which one defect prevents the detection of another [IEEE 610]”
- ❑ Symptoms of one error may disappear or change due to correction of another error.

*„Debugging is one of the more frustrating parts of programming. It has elements of brain teasers, coupled with the annoying recognition that you have made a mistake.“*

B. Shneiderman: Software Psychology. Winthrop Publishers, 1980.





# Debugging – Approaches

## ☐ **Brute Force**

Collect all data about the program execution. Try to find the error in it.

## ☐ **Cause Elimination**

The test case is reduced in size until only a small part of the program is executed. The error "must" be located in this part.

## ☐ **Backtracking**

The possible program execution paths are traced back from the occurrence of the symptoms to the program start. The error "must" be on one of the paths.

Tools can be used to collect the needed data (executed program parts, program paths, program state, ...): Coverage analysis, Interactive Debugger, Trace Generator

→ *see Lab sessions in Week 8!*

# TRAFFIC Principle

**Debugging** should follow the TRAFFIC principle:

- ☐ **T**rack the problem
- ☐ **R**eproduce – Requires control over data and environment.
- ☐ **A**utomate – Write a simple test case that exercises the problem.
- ☐ **F**ind Origins – Where does the failure originate? Locate likely fault locations.
- ☐ **F**ocus – Focus your effort on the most likely origin.
- ☐ **I**solate – Isolate the fault (see scientific method of debugging, next slide).
- ☐ **C**orrect – Fix the fault and verify that the failure no longer occurs.  
Check for regression errors.

# Debugging – Scientific Method

1. **Observe** the misbehavior.
2. Create a **hypothesis** about the cause of the misbehavior.
3. Use the hypothesis for predictions.
4. Test the hypothesis with **experiments** or observations and **refine** the hypothesis based on the results.
5. Repeat steps 3 and 4 until the **cause** is found.

# Debugging – Techniques

- ❑ **Reduce the input: *Delta Debugging***

Simplifying and Isolating Failure-Inducing Input

- ❑ **Reduce the program: *Program Slicing***

Isolating the relevant program statements/locations to focus debugging effort.

- ❑ Dynamic Slicing

- ❑ Static Forward and Backward Slicing

- ❑ Relevant Slicing

- ❑ **Identify faulty statements: *Statistical Fault Localization***

Ranking suspicious program statements.

# Reducing the input

IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 28, NO. 2, FEBRUARY 2002

1

## Simplifying and Isolating Failure-Inducing Input

Andreas Zeller, *Member, IEEE Computer Society*, and Ralf Hildebrandt

**Abstract**—Given some test case, a program fails. Which circumstances of the test case are responsible for the particular failure? The *Delta Debugging* algorithm generalizes and simplifies some failing test case to a *minimal test case* that still produces the failure; it also isolates the *difference* between a passing and a failing test case.

In a case study, the Mozilla web browser crashed after 95 user actions. Our prototype implementation automatically simplified the input to 3 relevant user actions. Likewise, it simplified 896 lines of HTML to the single line that caused the failure. The case study required 139 automated test runs, or 35 minutes on a 500 MHz PC.

**Index Terms**—automated debugging, debugging aids, testing tools, combinatorial testing, diagnostics, tracing.

### I. INTRODUCTION

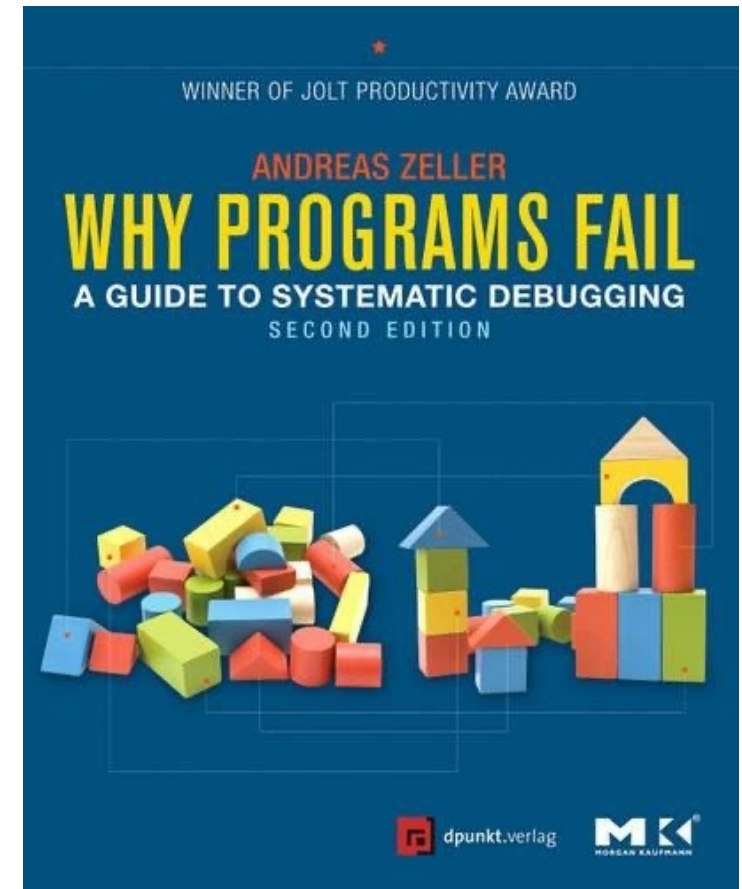
*Often people who encounter a bug spend a lot of time investigating which changes to the input file will make the bug go away and which changes will not affect it.*

— Richard Stallman, *Using and Porting GNU CC*

- A *bug report* should be as *specific* as possible, such that the engineer can recreate the context in which the program failed.
- On the other hand, a *test case* should be as *simple* as possible, because a minimal test case implies a most general context. Thus, a minimal test case not only allows for short problem descriptions and valuable problem insights, but it also subsumes several current and future bug reports.

The striking thing about test case simplification is that no one so far has thought to *automate* this task. Several textbooks and guides about debugging are available that tell how to use binary search in order to isolate the problem—based on the assumption that tests are carried out manually, too. With an automated test, however, we can automate this *simplification of test cases*, and we can automatically *isolate the difference that causes the failure*.

*Simplification of test cases.* Our *minimizing delta debugging algorithm dadmin* is fed with a failing test case, which it simplifies by *successive testing*. It stops when a *minimal test case*



Andreas Zeller and Ralf Hildebrandt, "Simplifying and isolating failure-inducing input," in IEEE Transactions on Software Engineering, vol. 28, 2002.  
Andreas Zeller. "Why programs fail: a guide to systematic debugging". Elsevier, 2009.

# Delta Debugging

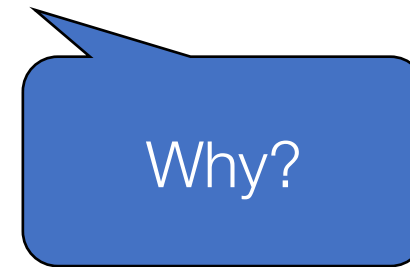
- ❑ Goal: Figure out a minimal cause that ‘explains’ an error!
- ❑ Use a variation on binary search: narrow the difference between passing and failing inputs



*“Rather than only **minimizing** the failing input, delta debugging (dd) also maximizes the passing input until a minimal **failure-inducing difference** is obtained.”*

# Delta Debugging

- ❑ Check: <https://www.debuggingbook.org>
- ❑ Applications: Delta Debugging can isolate failure causes
  - ❑ in the (general) input
  - ❑ in the version history
  - ❑ in thread schedules
- ❑ Every such identified cause implies a fix – but not necessarily a correction.



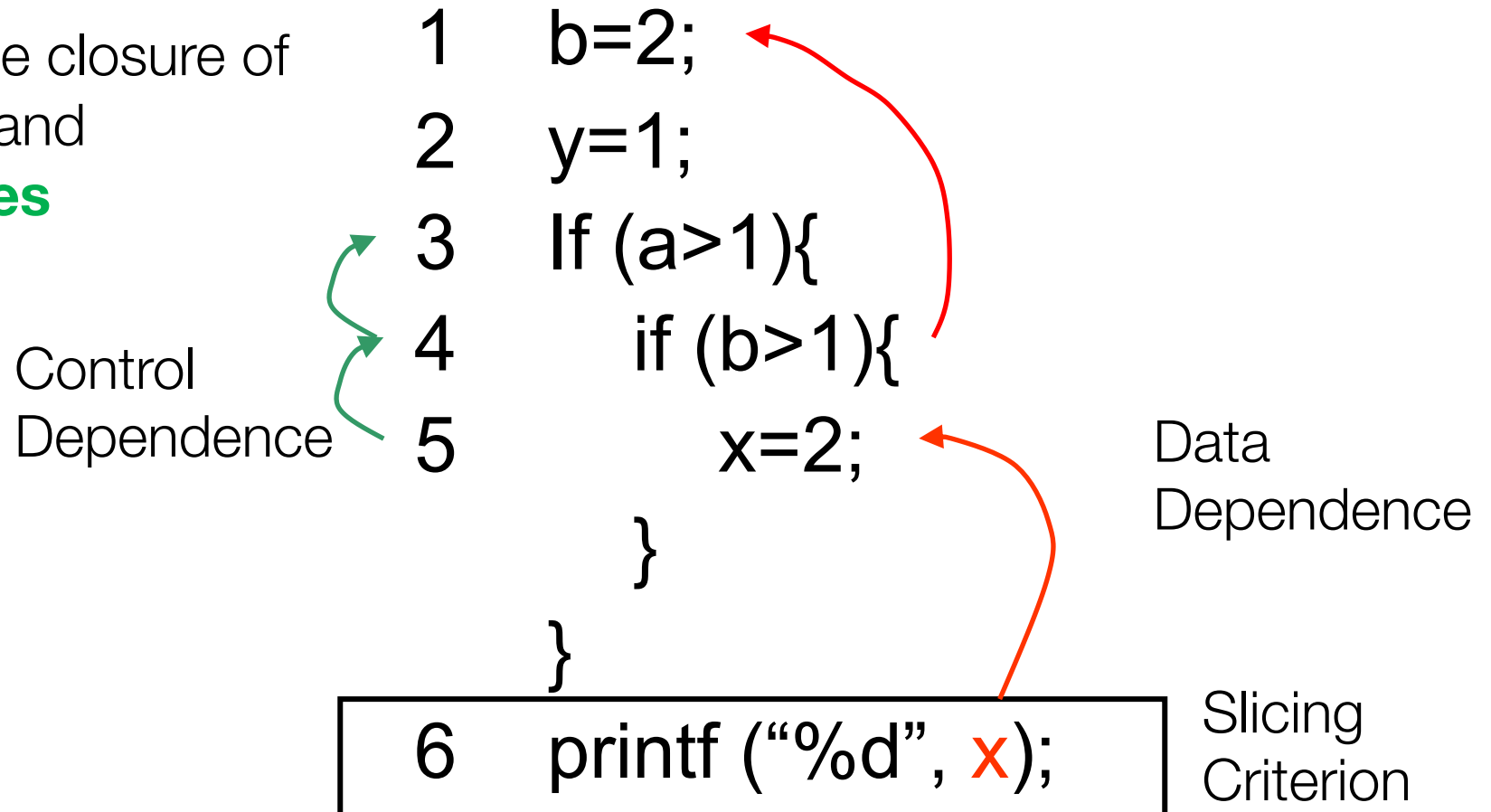


# Reducing the program

- ❑ **Program slicing** is the computation of the ***subset of program statements*** (→ the program **slice**).
- ❑ The **program slice** includes statements that may affect the values at some point of interest (~ the **slicing criterion**)
- ❑ Concept: Select a line to be considered and hide all irrelevant lines.
- ❑ Dynamic Slicing: slice for a particular program execution  
→ dynamic dependencies
- ❑ Static Slicing → static dependencies
  - ❑ What is **affected** by this slicing criterion? (forward)
  - ❑ What is **influenced** the value of this variable? (backward)

# Dynamic Slicing

- ❑ Slice backward from the erroneous output of the program
- ❑ Dynamic slice includes the closure of
  - ❑ **Data dependencies** and
  - ❑ **Control dependencies**



# Dynamic Slicing (applied)

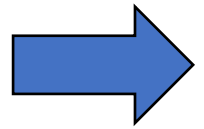
runningVersion=false

```
1. void setRunningVersion(boolean runningVersion)
2.   if( runningVersion ) {
3.     savedValue = value;
4.   }
5.   else{
6.     savedValue = "";
7.   }
8.   this.runningVersion = runningVersion;
9.   System.out.println(savedValue);
10. }
```

Slicing Criterion

# Dynamic Slicing – Problems

- ❑ Huge overheads: Backwards slicing requires trace storage.
- ❑ Dynamic Slice is still too large ...  
... for human comprehension



Hierarchical Slicing: operates on higher abstractions, e.g., components, packages, classes, ...

Developer/User can *zoom in* and investigate!

---

Tao Wang and Abhik Roychoudhury. "Hierarchical dynamic slicing". In Proceedings of the 2007 international symposium on Software testing and analysis (ISSTA), 2007.

J. Gao, Y. Liu and B. Li, "Hierarchical Slicing Object-Oriented Programs for Debugging," 2009 International Conference on Information Engineering and Computer Science, 2009.

# Static Slicing (1/2)

If dynamic slice computation and traversal becomes **manageable**

- ❑ We can look **beyond** dynamic slices.
- ❑ We can look at errors which **are not captured in dynamic slices**.

*What is affected by this assignment?*

```
read(max);  
int a = 1;  
int b = 1;  
int i = 1;  
int fak = 1;  
do {  
    if (i > 2) {  
        int tmp = a;  
        b = b + a;  
        a = tmp;  
    }  
    fak = fak * i;  
    i++;  
} while (i <= max);  
  
print(n);  
print(b);  
print(fak);
```

*What is influenced the value of this variable?*

→ static dependencies, does not depend on specific test case

# Static Slicing (2/2)

## Forward Slice

*What is affected by this assignment?*

```
read(max);  
int a = 1;  
int b = 1;  
int i = 1;  
int fak = 1;  
do {  
    if (i > 2) {  
        int tmp = a;  
        b = b + a;  
        a = tmp;  
    }  
    fak = fak * i;  
    i++;  
} while (i <= max);  
  
print(n);  
print(b);  
print(fak);
```

```
read(max);  
int a = 1;  
int b = 1;  
int i = 1;  
int fak = 1;  
do {  
    if (i > 2) {  
        int tmp = a;  
        b = b + a;  
        a = tmp;  
    }  
    fak = fak * i;  
    i++;  
} while (i <= max);  
  
print(n);  
print(b);  
print(fak);
```

## Backward Slice

*What is influenced the value of this variable?*

```
read(max);  
int a = 1;  
int b = 1;  
int i = 1;  
int fak = 1;  
do {  
    if (i > 2) {  
        int tmp = a;  
        b = b + a;  
        a = tmp;  
    }  
    fak = fak * i;  
    i++;  
} while (i <= max);  
  
print(n);  
print(b);  
print(fak);
```

# Static vs Dynamic Slicing (1/4)

## Static Slicing

- ❑ source code
- ❑ statement
- ❑ static dependence

## Dynamic Slicing

- ❑ a particular execution
- ❑ statement instance
- ❑ dynamic dependence

```
1  b=1;  
2  If (a>1)  
3      x=1;  
4  else  
5      x=2;
```

```
6  printf ("%d", x);
```

Slicing Criterion



# Limitations of Dynamic Slicing



input: a=2

```
1 b=1;
2 x=1;
3 if (a>1){
4     if (b>1){
5         x=2;
6     }
7 }
8 printf ("%d", x);
```

What is the content of the dynamic slice?

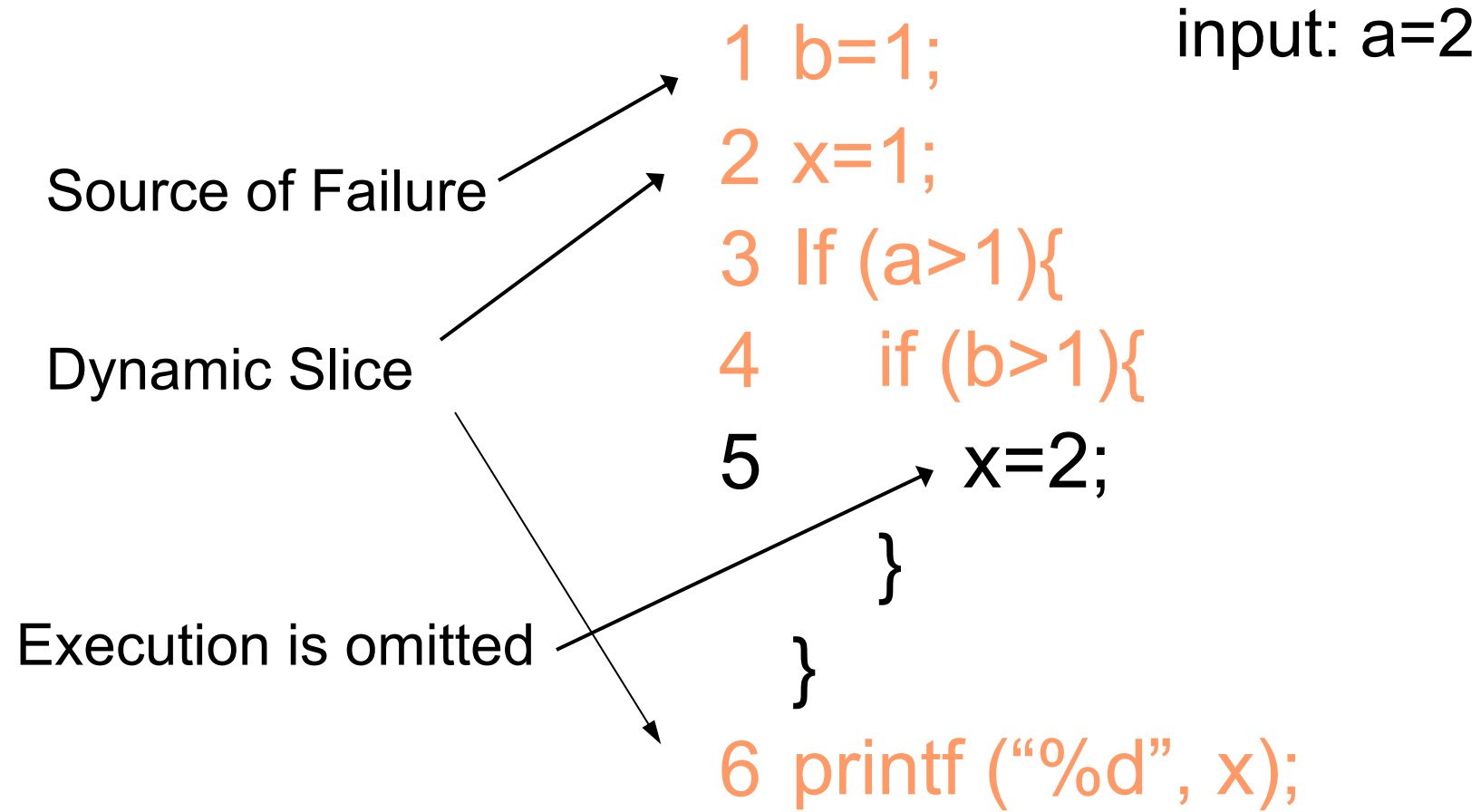
Let's assume that the error is that line 5 is not executed. What can be the cause and why is it problematic wrt dynamic slice?

# Static vs Dynamic Slicing <sup>(2/4)</sup>

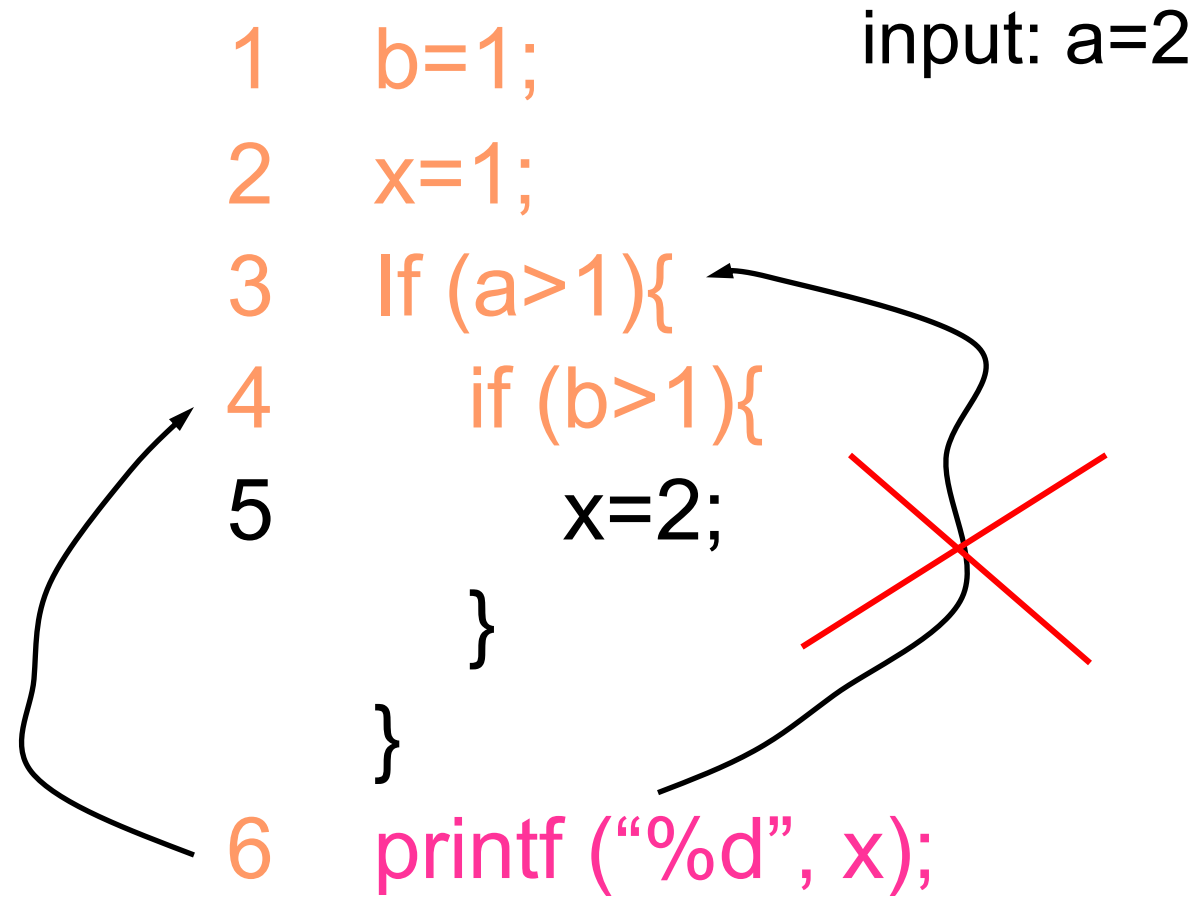
```
1 b=1;
2 x=1;
3 if (a>1){
4     if (b>1){
5         x=2;
6     }
7 }
8 printf ("%d", x);
```

input: a=2

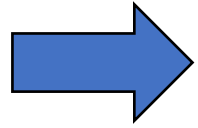
# Static vs Dynamic Slicing (3/4)



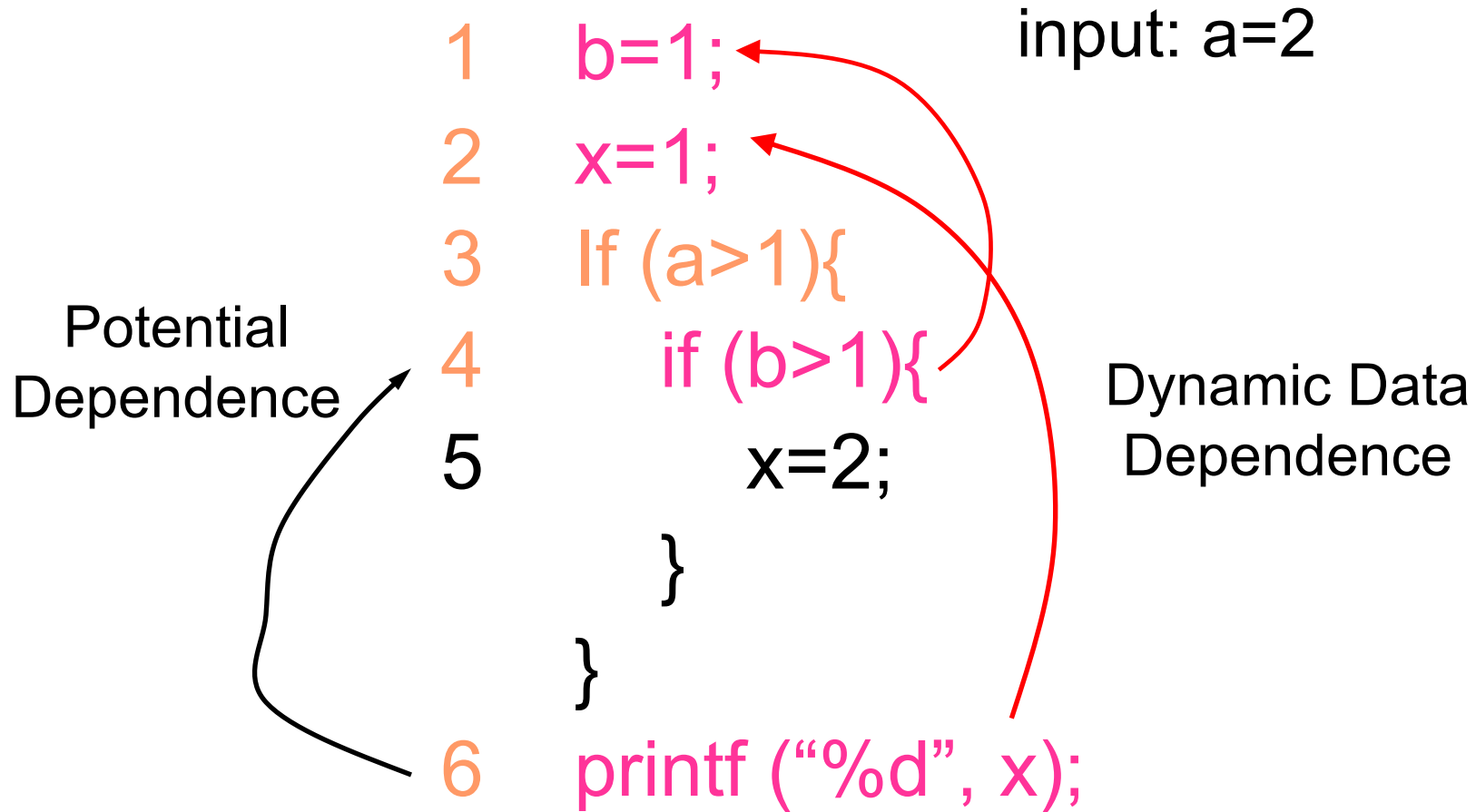
# Potential Dependence (1/2)



# Potential Dependence (2/2)



*Relevant Slice*



# Slicing Example Result

Static	Dynamic	Relevant	
1		1	1    b=1;    input: a=2
2	2	2	2    x=1;
3			3    If (a>1){
4		4	4        if (b>1){
5			5            x=2;
			}
			}
6	6	6	6    printf ("%d", x);

# Statistical Fault Localization – Tarantula

$$suspiciousness(s) = \frac{\frac{failed(s)}{total\ failed}}{\frac{passed(s)}{total\ passed} + \frac{failed(s)}{total\ failed}}$$

- ❑ „fuzzy slices“
  - ❑ Probability that the line of code will lead to the misbehavior
  - ❑ Lines of code that are often executed on failed tests are suspicious.

---

James A. Jones, Mary Jean Harrold, John T. Stasko: Visualization of test information to assist fault localization. ICSE 2002: 467-477  
James A. Jones, Mary Jean Harrold: Empirical evaluation of the tarantula automatic fault-localization technique. ASE 2005: 273-282



# Dynamic Slicing - Example

*Recap*

Test Cases

mid(){ int x,y,z,m;	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3
read("Enter 3 numbers:", x,y,z);	•	•	•	•	•	•
m = z;	•	•	•	•	•	•
if(y < z)	•	•	•	•	•	•
if(x < y)	•	•			•	•
m = y;		•				
else if (x < z)	•				•	•
m = y; // bug	•					•
else			•	•		
if (x > y)			•	•		
m = y;			•			
else if ( x > z)				•		
m = x;						
print("Middle number is:", m);	•	•	•	•	•	•
}						
Pass Status	P	P	P	P	P	F

# Statistical Fault Localization

$$suspiciousness(s) = \frac{\frac{failed(s)}{total\ failed}}{\frac{passed(s)}{total\ passed} + \frac{failed(s)}{total\ failed}}$$

mid(){ int x,y,z,m;	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3	suspiciousness
read("Enter 3 numbers:", x,y,z);	•	•	•	•	•	•	0.5
m = z;	•	•	•	•	•	•	0.5
if(y < z)	•	•	•	•	•	•	0.5
if(x < y)	•	•			•	•	0.625
m = y;		•					0.0
else if (x < z)	•				•	•	0.714
m = y; // bug	•					•	0.833
else			•	•			0.0
if (x > y)			•	•			0.0
m = y;			•				0.0
else if ( x > z)				•			0.0
m = x;							---
print("Middle number is:", m);	•	•	•	•	•	•	0.5
}							
Pass Status	P	P	P	P	P	F	

# Suspiciousness Scores

Can use several other available metrics for ranking statements, e.g. Ochiai metric

$$\text{Score}(s) = \frac{\text{fail}(s)}{\sqrt{\text{allfail} * (\text{fail}(s) + \text{pass}(s))}}$$

A model for spectra-based software diagnosis, Naish et. al., TOSEM 20(3), 2011.

Name	Formula	Name	Formula
Jaccard	$\frac{a_{ef}}{a_{ef} + a_{nf} + a_{ep}}$	Anderberg	$\frac{a_{ef}}{a_{ef} + 2(a_{nf} + a_{ep})}$
Sørensen-Dice	$\frac{2a_{ef}}{2a_{ef} + a_{nf} + a_{ep}}$	Dice	$\frac{2a_{ef}}{a_{ef} + a_{nf} + a_{ep}}$
Kulczynski1	$\frac{a_{ef}}{a_{nf} + a_{ep}}$	Kulczynski2	$\frac{1}{2} \left( \frac{a_{ef}}{a_{ef} + a_{nf}} + \frac{a_{ef}}{a_{ef} + a_{ep}} \right)$
Russell and Rao	$\frac{a_{ef}}{a_{ef} + a_{nf} + a_{ep} + a_{np}}$	Hamann	$\frac{a_{ef} + a_{np} - a_{nf} - a_{ep}}{a_{ef} + a_{nf} + a_{ep} + a_{np}}$
Simple Matching	$\frac{a_{ef} + a_{np}}{a_{ef} + a_{nf} + a_{ep} + a_{np}}$	Sokal	$\frac{2(a_{ef} + a_{np})}{2(a_{ef} + a_{np}) + a_{nf} + a_{ep}}$
M1	$\frac{a_{ef} + a_{np}}{a_{nf} + a_{ep}}$	M2	$\frac{a_{ef}}{a_{ef} + a_{np} + 2(a_{nf} + a_{ep})}$
Rogers-Tanimoto	$\frac{a_{ef} + a_{np}}{a_{ef} + a_{ep} + 2(a_{nf} + a_{ep})}$	Goodman	$\frac{2a_{ef} - a_{nf} - a_{ep}}{2a_{ef} + a_{nf} + a_{ep}}$
Hamming etc.	$a_{ef} + a_{np}$	Euclid	$\sqrt{a_{ef} + a_{np}}$
Ochiai	$\frac{a_{ef}}{\sqrt{(a_{ef} + a_{np})(a_{ef} + a_{ep})}}$	Overlap	$\frac{a_{ef}}{\min(a_{ef}, a_{nf}, a_{ep})}$
Tarantula	$\frac{\frac{a_{ef}}{a_{ef} + a_{nf}}}{\frac{a_{ef}}{a_{ef} + a_{nf}} + \frac{a_{ep}}{a_{ep} + 2a_{np}}}$	Zoltar	$\frac{a_{ef}}{a_{ef} + a_{nf} + a_{ep} + \frac{10000a_{nf}a_{ep}}{a_{ef}}}$
Ample	$\left  \frac{a_{ef}}{a_{ef} + a_{nf}} - \frac{a_{ep}}{a_{ep} + a_{np}} \right $	Wong1	$a_{ef}$
Wong2	$a_{ef} - a_{ep}$		
Wong3	$a_{ef} - h$ , where $h = \begin{cases} a_{ep} & \text{if } a_{ep} \leq 2 \\ 2 + 0.1(a_{ep} - 2) & \text{if } 2 < a_{ep} \leq 10 \\ 2.8 + 0.001(a_{ep} - 10) & \text{if } a_{ep} > 10 \end{cases}$		
Ochiai2	$\frac{a_{ef}a_{np}}{\sqrt{(a_{ef} + a_{ep})(a_{np} + a_{nf})(a_{ef} + a_{nf})(a_{ep} + a_{np})}}$		
Geometric Mean	$\frac{a_{ef}a_{np} - a_{nf}a_{ep}}{\sqrt{(a_{ef} + a_{ep})(a_{np} + a_{nf})(a_{ef} + a_{nf})(a_{ep} + a_{np})}}$		
Harmonic Mean	$\frac{(a_{ef}a_{np} - a_{nf}a_{ep})((a_{ef} + a_{ep})(a_{np} + a_{nf}) + (a_{ef} + a_{nf})(a_{ep} + a_{np}))}{(a_{ef} + a_{ep})(a_{np} + a_{nf})(a_{ef} + a_{nf})(a_{ep} + a_{np})}$		
Arithmetic Mean	$\frac{2a_{ef}a_{np} - 2a_{nf}a_{ep}}{(a_{ef} + a_{ep})(a_{np} + a_{nf}) + (a_{ef} + a_{nf})(a_{ep} + a_{np})}$		
Cohen	$\frac{2a_{ef}a_{np} - 2a_{nf}a_{ep}}{(a_{ef} + a_{ep})(a_{np} + a_{ep}) + (a_{ef} + a_{nf})(a_{nf} + a_{np})}$		
Scott	$\frac{4a_{ef}a_{np} - 4a_{nf}a_{ep} - (a_{nf} - a_{ep})^2}{(2a_{ef} + a_{nf} + a_{ep})(2a_{np} + a_{nf} + a_{ep})}$		
Fleiss	$\frac{4a_{ef}a_{np} - 4a_{nf}a_{ep} - (a_{nf} - a_{ep})^2}{(2a_{ef} + a_{nf} + a_{ep}) + (2a_{np} + a_{nf} + a_{ep})}$		
Rogot1	$\frac{1}{2} \left( \frac{a_{ef}}{2a_{ef} + a_{nf} + a_{ep}} + \frac{a_{np}}{2a_{np} + a_{nf} + a_{ep}} \right)$		
Rogot2	$\frac{1}{4} \left( \frac{a_{ef}}{a_{ef} + a_{ep}} + \frac{a_{ef}}{a_{ef} + a_{nf}} + \frac{a_{np}}{a_{np} + a_{ep}} + \frac{a_{np}}{a_{np} + a_{nf}} \right)$		

# Visualizing Fault Localization <sup>(1/3)</sup>

For a program line  $s$ :

**Color** describes the **pass/fail** of test cases that executed the program line  $s$



**Brightness** represents the **confidence** of the color assignment to the program line  $s$



# Visualizing Fault Localization <sup>(2/3)</sup>

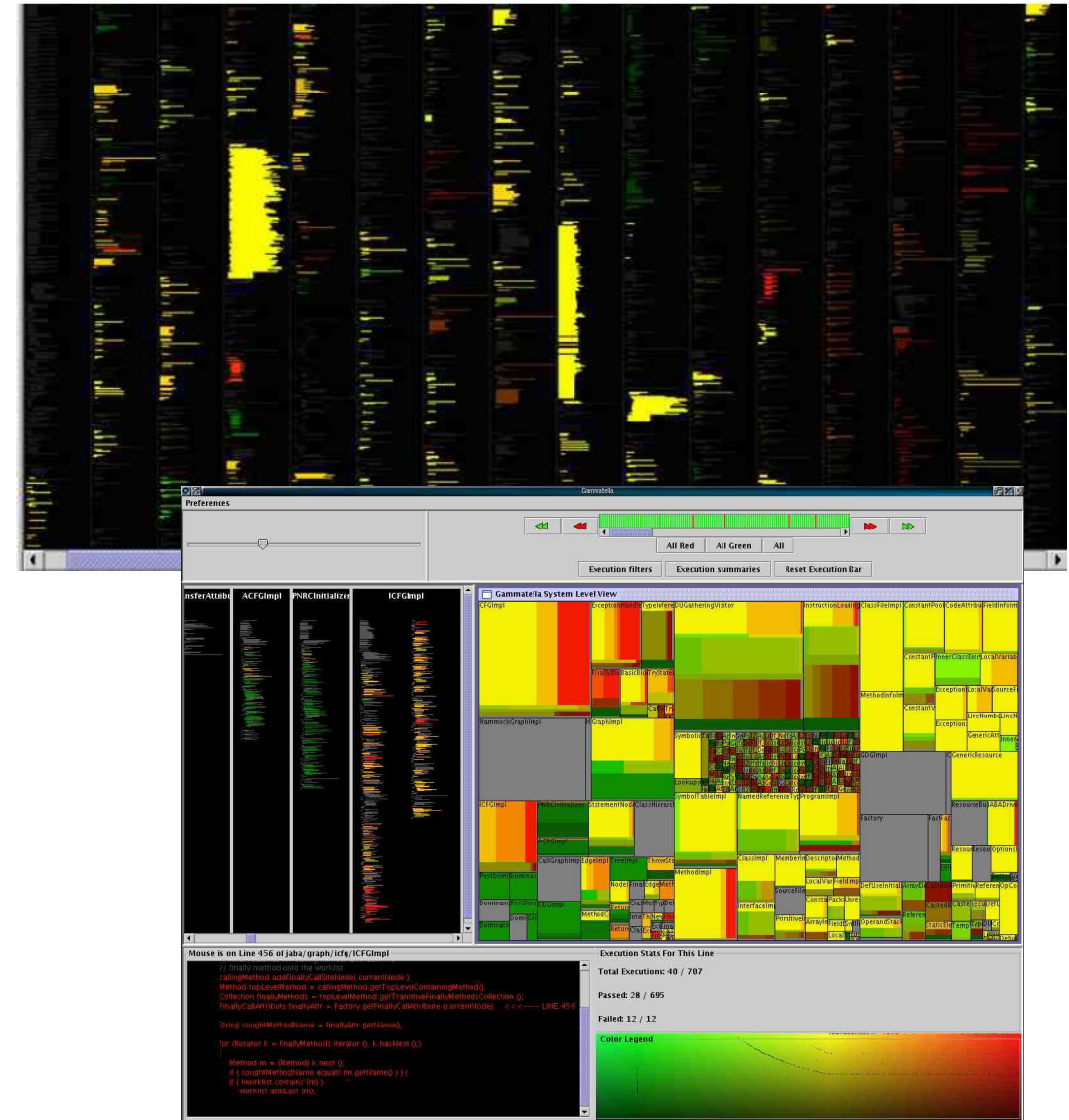
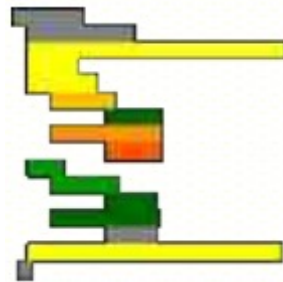
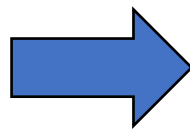
Test Cases

mid(){ int x,y,z,m;	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3
read("Enter 3 numbers:", x,y,z);	•	•	•	•	•	•
m = z;	•	•	•	•	•	•
if(y < z)	•	•	•	•	•	•
if(x < y)	•	•			•	•
m = y;		•				
else if (x < z)	•				•	•
m = y; // bug	•					•
else			•	•		
if (x > y)			•	•		
m = y;			•			
else if ( x > z)				•		
m = x;						
print("Middle number is:", m);	•	•	•	•	•	•
}						
Pass Status	P	P	P	P	P	F

# Visualizing Fault Localization (3/3)

- ❑ SeeSoft view!
- ❑ Each pixel represents a char in the program file

```
mid(){  
  int x,y,z,m;  
  read("Enter 3 numbers:", x,y,z);  
  m = z;  
  if(y < z)  
    if(x < y)  
      m = y;  
    else if (x < z)  
      m = y; // bug  
  else  
    if (x > y)  
      m = y;  
    else if ( x > z)  
      m = x;  
  print("Middle number is:", m);  
}
```



# Debugging – Closing Remarks

- ❑ **Before** fixing, be sure to **understand the problem** (cause-effect chain).
- ❑ **Understand the program**, not just the problem (effect of your change).
- ❑ Before you make a change to the code, be confident that it will work!
- ❑ Never start changing code before saving the original. Always use **version control**!
- ❑ Wishful thinking doesn't fix bugs! (think hard about the solution)
- ❑ Use the **most general** fix available.
- ❑ Do not attempt to fix **multiple defects** at the same time
- ❑ After fixing,
  - ❑ make sure the fix **solves the problem**,
  - ❑ make sure **no new defects** are introduced,
  - ❑ you might want to **learn** from it, and
  - ❑ look for **similar defects**!



# Conclusion

- ❑ **Debugging** is time-consuming, but can be aided by **tools** and **techniques**.
- ❑ Note: quality **cannot** be introduced by testing...  
→ analytical vs constructive methods!

## Next Week (Project-Part) – Week 9: **Static Analysis: Techniques & Tools**

- Finishing Debugging (if remaining items) and Recap.
- Static Analysis: IDE, Checkstyle, FindBugs/Spotbugs, Error Prone, Infer
- Assignment 8: Final Code Submission + Presentations