Impact Analysis

SWE 795, Fall 2019

Software Engineering Environments

Today

- Part 1 (Lecture)(~40 mins)
    - Impact Analysis
- Part 2 (Project Presentations, Part 1)(~40 mins)
- Break
- Part 3 (Project Presentations, Part 2)(~50 mins)

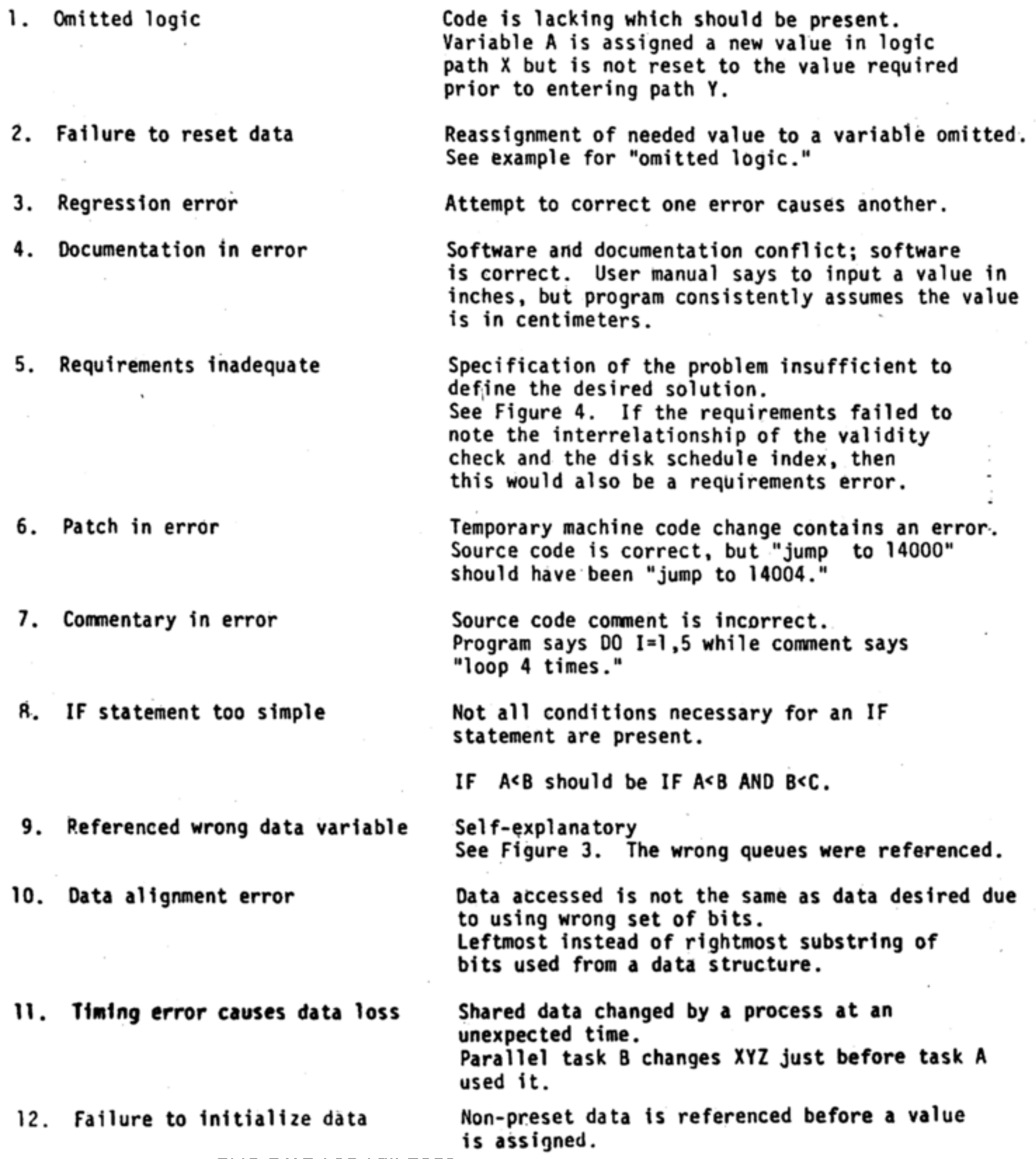
Impact analysis

- "Identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change"
- Using investigation to determine what needs to be done to make change consistently

S. A. Bohner and R. S. Arnold, *Software Change Impact Analysis*. Los Alamitos, CA, USA: IEEE Computer Society Publications Tutorial Series, 1996.

What strategies do you use for impact analysis?

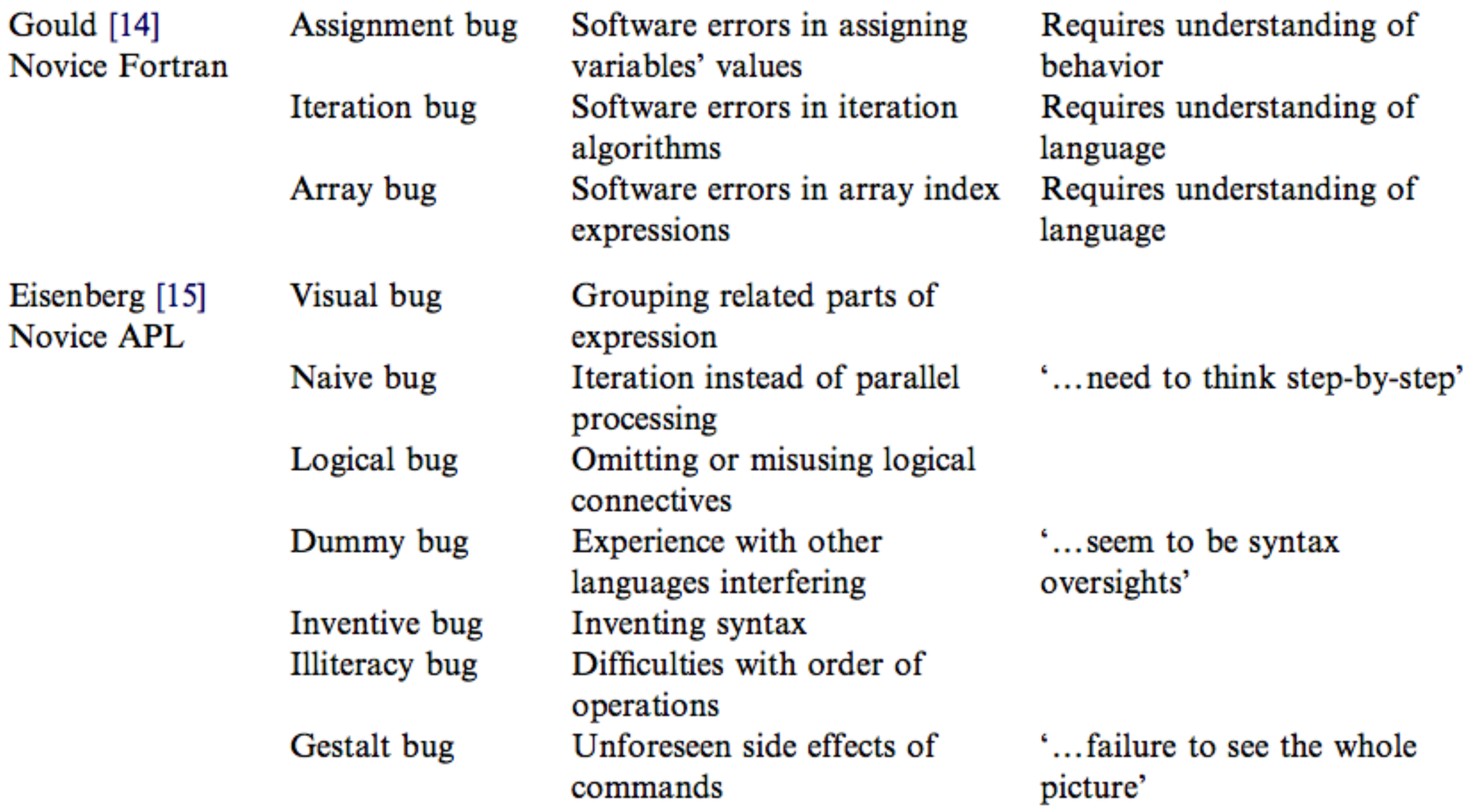
Where do defects come from?



1. Omitted logic	Code is lacking which should be present. Variable A is assigned a new value in logic path X but is not reset to the value required prior to entering path Y.
2. Failure to reset data	Reassignment of needed value to a variable omitted. See example for "omitted logic."
3. Regression error	Attempt to correct one error causes another.
4. Documentation in error	Software and documentation conflict; software is correct. User manual says to input a value in inches, but program consistently assumes the value is in centimeters.
5. Requirements inadequate	Specification of the problem insufficient to define the desired solution. See Figure 4. If the requirements failed to note the interrelationship of the validity check and the disk schedule index, then this would also be a requirements error.
6. Patch in error	Temporary machine code change contains an error. Source code is correct, but "jump to 14000" should have been "jump to 14004."
7. Commentary in error	Source code comment is incorrect. Program says DO I=1,5 while comment says "loop 4 times."
8. IF statement too simple	Not all conditions necessary for an IF statement are present. IF A<B should be IF A<B AND B<C.
9. Referenced wrong data variable	Self-explanatory See Figure 3. The wrong queues were referenced.
10. Data alignment error	Data accessed is not the same as data desired due to using wrong set of bits. Leftmost instead of rightmost substring of bits used from a data structure.
11. Timing error causes data loss	Shared data changed by a process at an unexpected time. Parallel task B changes XYZ just before task A used it.
12. Failure to initialize data	Non-preset data is referenced before a value is assigned.

[Glass TSE81]

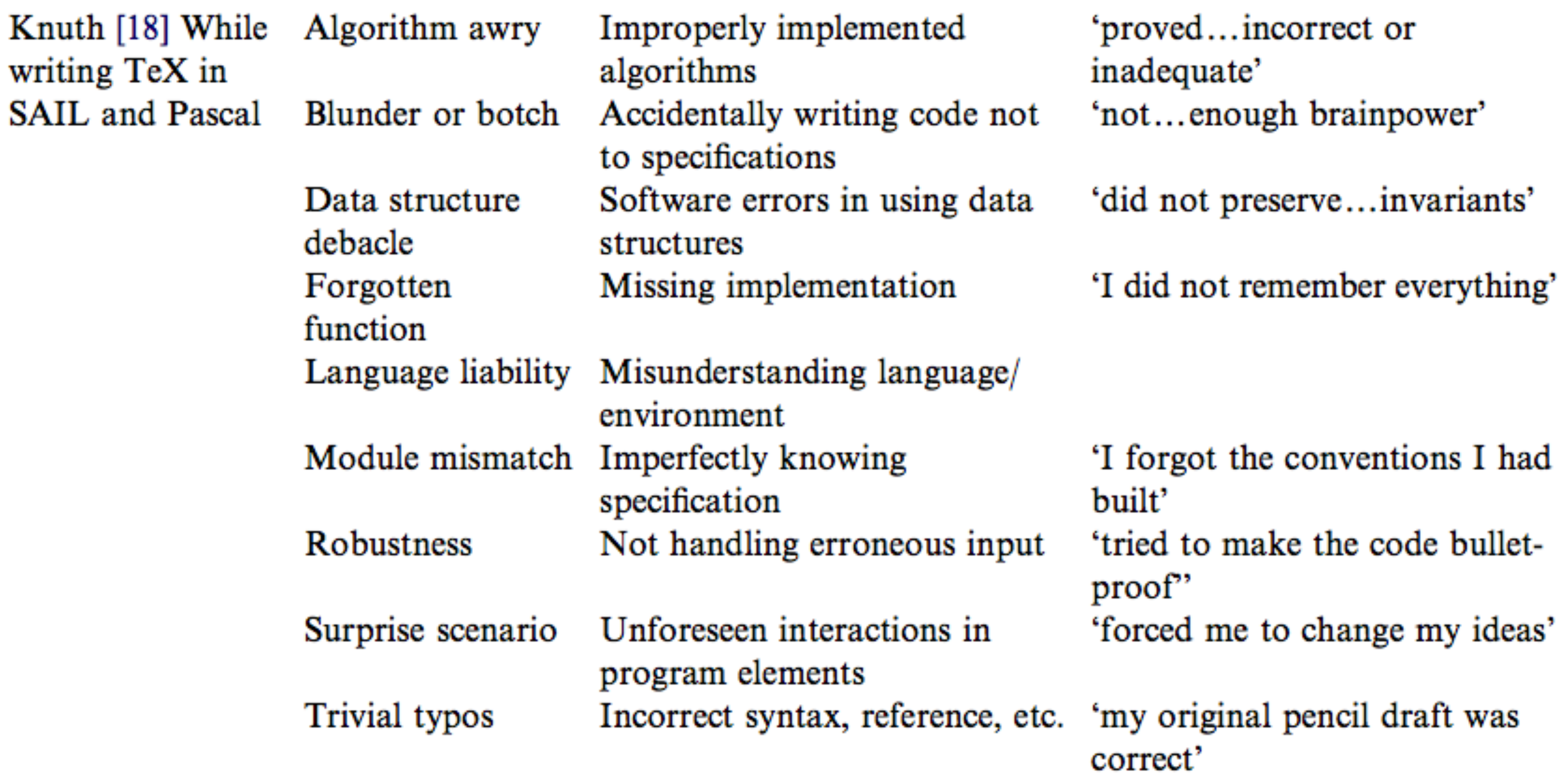
Where do defects come from?



Gould [14] Novice Fortran	Assignment bug	Software errors in assigning variables' values	Requires understanding of behavior
	Iteration bug	Software errors in iteration algorithms	Requires understanding of language
	Array bug	Software errors in array index expressions	Requires understanding of language
Eisenberg [15] Novice APL	Visual bug	Grouping related parts of expression	
	Naive bug	Iteration instead of parallel processing	'…need to think step-by-step'
	Logical bug	Omitting or misusing logical connectives	
	Dummy bug	Experience with other languages interfering	'…seem to be syntax oversights'
	Inventive bug	Inventing syntax	
	Illiteracy bug	Difficulties with order of operations	
	Gestalt bug	Unforeseen side effects of commands	'…failure to see the whole picture'

Adapted from Ko & Myers, JVLC05

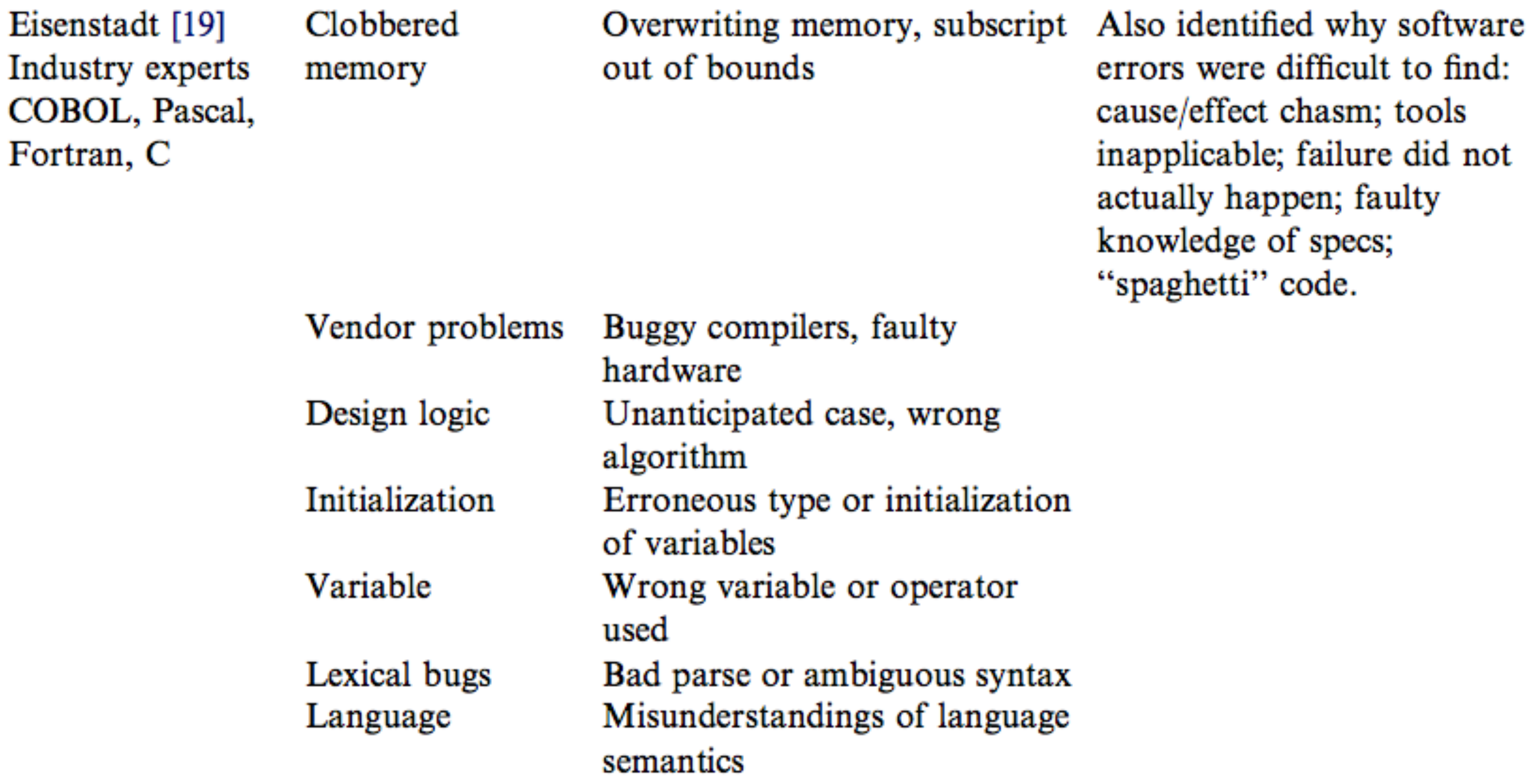
Where do defects come from?



Knuth [18] While writing TeX in SAIL and Pascal	Algorithm awry	Improperly implemented algorithms	'proved…incorrect or inadequate'
	Blunder or botch	Accidentally writing code not to specifications	'not…enough brainpower'
	Data structure debacle	Software errors in using data structures	'did not preserve…invariants'
	Forgotten function	Missing implementation	'I did not remember everything'
	Language liability	Misunderstanding language/ environment	
	Module mismatch	Imperfectly knowing specification	'I forgot the conventions I had built'
	Robustness	Not handling erroneous input	'tried to make the code bullet-proof''
	Surprise scenario	Unforeseen interactions in program elements	'forced me to change my ideas'
	Trivial typos	Incorrect syntax, reference, etc.	'my original pencil draft was correct'

Adapted from Ko & Myers, JVLC05

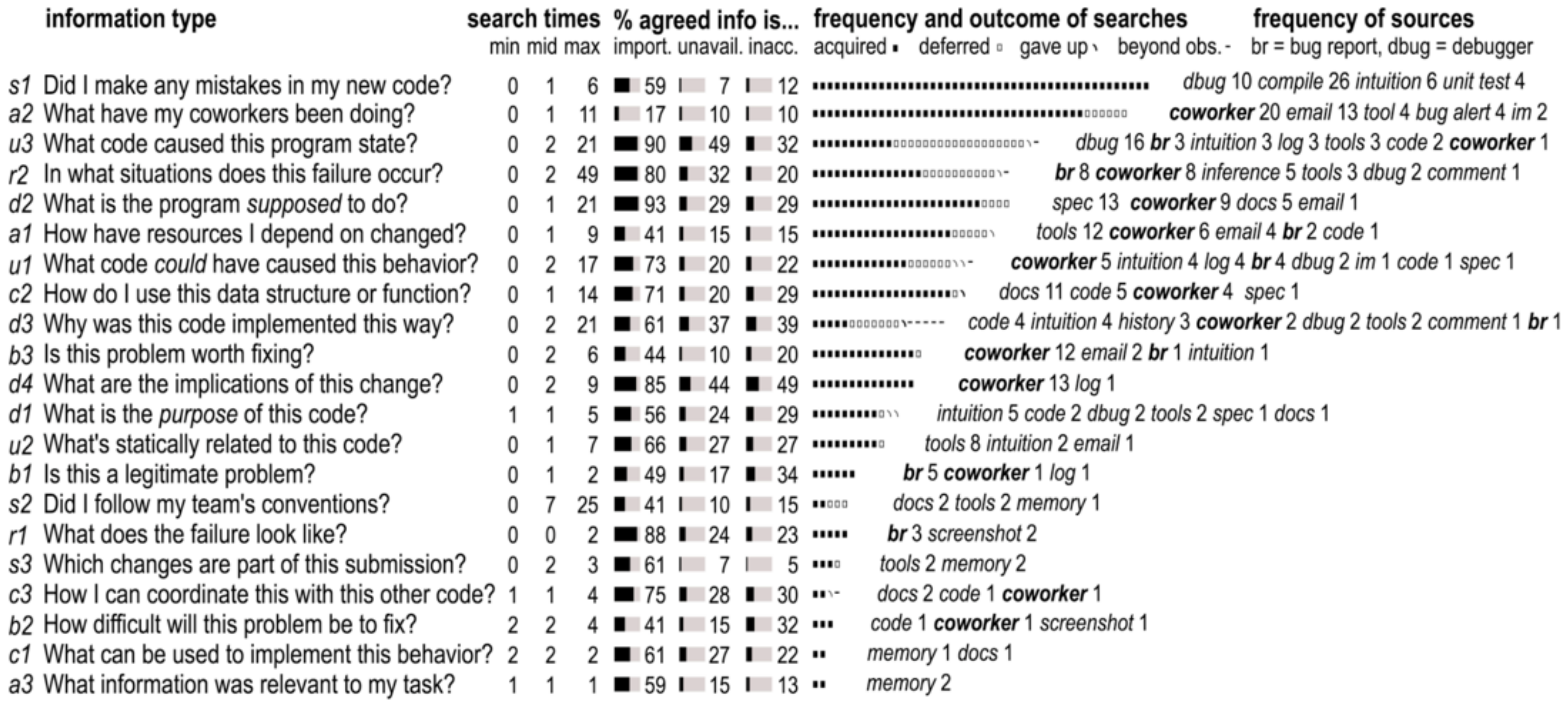
Where do defects come from?



Eisenstadt [19] Industry experts COBOL, Pascal, Fortran, C	Clobbered memory	Overwriting memory, subscript out of bounds	Also identified why software errors were difficult to find: cause/effect chasm; tools inapplicable; failure did not actually happen; faulty knowledge of specs; "spaghetti" code.
	Vendor problems	Buggy compilers, faulty hardware	
	Design logic	Unanticipated case, wrong algorithm	
	Initialization	Erroneous type or initialization of variables	
	Variable	Wrong variable or operator used	
	Lexical bugs	Bad parse or ambiguous syntax	
	Language	Misunderstandings of language semantics	

Adapted from Ko & Myers, JVLC05

Reasoning about correctness



	information type	search times min	mid	max	% agreed info is... import.	unavail.	inacc.	frequency and outcome of searches acquired ▪ deferred ▫ gave up ˎ beyond obs. -	frequency of sources br = bug report, dbug = debugger
s1	Did I make any mistakes in my new code?	0	1	6	59	7	12	▪▪▪	*dbug* 10 *compile* 26 *intuition* 6 *unit test* 4
a2	What have my coworkers been doing?	0	1	11	17	10	10	▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▫▫▫▫▫▫▫	**coworker** 20 *email* 13 *tool* 4 *bug alert* 4 *im* 2
u3	What code caused this program state?	0	2	21	90	49	32	▪▪▪▪▪▪▪▪▪▪▪▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫ˎ-	*dbug* 16 **br** 3 *intuition* 3 *log* 3 *tools* 3 *code* 2 **coworker** 1
r2	In what situations does this failure occur?	0	2	49	80	32	20	▪▪▪▪▪▪▪▪▪▪▪▪▪▪▫▫▫▫▫▫▫▫▫▫ˎ-	**br** 8 **coworker** 8 *inference* 5 *tools* 3 *dbug* 2 *comment* 1
d2	What is the program *supposed* to do?	0	1	21	93	29	29	▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▫▫▫▫	*spec* 13 **coworker** 9 *docs* 5 *email* 1
a1	How have resources I depend on changed?	0	1	9	41	15	15	▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▫▫▫▫▫ˎ	*tools* 12 **coworker** 6 *email* 4 **br** 2 *code* 1
u1	What code *could* have caused this behavior?	0	2	17	73	20	22	▪▪▪▪▪▪▪▪▪▪▪▪▪▫▫▫▫▫▫ˎˎ-	**coworker** 5 *intuition* 4 *log* 4 **br** 4 *dbug* 2 *im* 1 *code* 1 *spec* 1
c2	How do I use this data structure or function?	0	1	14	71	20	29	▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▫ˎ	*docs* 11 *code* 5 **coworker** 4 *spec* 1
d3	Why was this code implemented this way?	0	2	21	61	37	39	▪▪▪▪▪▫▫▫▫▫▫▫ˎ-----	*code* 4 *intuition* 4 *history* 3 **coworker** 2 *dbug* 2 *tools* 2 *comment* 1 **br** 1
b3	Is this problem worth fixing?	0	2	6	44	10	20	▪▪▪▪▪▪▪▪▪▪▪▪▪▫	**coworker** 12 *email* 2 **br** 1 *intuition* 1
d4	What are the implications of this change?	0	2	9	85	44	49	▪▪▪▪▪▪▪▪▪▪▪▪▪	**coworker** 13 *log* 1
d1	What is the *purpose* of this code?	1	1	5	56	24	29	▪▪▪▪▪▪▪▪▪▫ˎˎ	*intuition* 5 *code* 2 *dbug* 2 *tools* 2 *spec* 1 *docs* 1
u2	What's statically related to this code?	0	1	7	66	27	27	▪▪▪▪▪▪▪▪▪▫	*tools* 8 *intuition* 2 *email* 1
b1	Is this a legitimate problem?	0	1	2	49	17	34	▪▪▪▪▪▪	**br** 5 **coworker** 1 *log* 1
s2	Did I follow my team's conventions?	0	7	25	41	10	15	▪▪▫▫▫	*docs* 2 *tools* 2 *memory* 1
r1	What does the failure look like?	0	0	2	88	24	23	▪▪▪▪▪	**br** 3 *screenshot* 2
s3	Which changes are part of this submission?	0	2	3	61	7	5	▪▪▪▫	*tools* 2 *memory* 2
c3	How I can coordinate this with this other code?	1	1	4	75	28	30	▪▪ˎ-	*docs* 2 *code* 1 **coworker** 1
b2	How difficult will this problem be to fix?	2	2	4	41	15	32	▪▪▪	*code* 1 **coworker** 1 *screenshot* 1
c1	What can be used to implement this behavior?	2	2	2	61	27	22	▪▪	*memory* 1 *docs* 1
a3	What information was relevant to my task?	1	1	1	59	15	13	▪▪	*memory* 2

Andrew J. Ko, Robert DeLine, and Gina Venolia. 2007. Information Needs in Collocated Software Development Teams. In Proceedings of the 29th international conference on Software Engineering (ICSE '07). IEEE Computer Society, Washington, DC, USA, 344-353. DOI: https://doi.org/10.1109/ICSE.2007.45

Which team's component caused this bug? (1)

Intent and Implementation (32)

What is the intent of this code? (12) [15]
What does this do (6) in this case (10)? (16) [24]
How does it implement this behavior? (4) [24]

Refactoring (25)

Is there functionality or code that could be refactored? (4)
Is the existing design a good design? (2)
Is it possible to refactor this? (9)
How can I refactor this (2) without breaking existing users(7)? (9)
Should I refactor this? (1)
Are the benefits of this refactoring worth the time investment? (3)

History (23)

When, how, by whom, and why was this code changed or inserted? (13)[7]
What else changed when this code was changed or inserted? (2)
How has it changed over time? (4)[7]
Has this code always been this way? (2)
What recent changes have been made? (1)[15][7]
Have changes in another branch been integrated into this branch? (1)

Implications (21)

What are the implications of this change for (5) API clients (5), security (3), concurrency (3), performance (2), platforms (1), tests (1), or obfuscation (1)? (21) [15][24]

What parameter values could lead to this case?
What are the possible actual methods called by a
here? (6)
How do calls flow across process boundaries? (
How many recursive calls happen during this op
Is this method or code path called frequently, or
What throws this exception? (1)
What is catching this exception? (1)

Contracts (17)

What assumptions about preconditions does this
What assumptions about pre(3)/post(2)condition
What exceptions or errors can this method gene
What are the constraints on or normal values of
What is the correct order for calling these metho
these objects? (2)
What is responsible for updating this field? (1)

Performance (16)

What is the performance of this code (5) on a la
Which part of this code takes the most time? (4)
Can this method have high stack consumption fr
How big is this in memory? (2)
How many of these objects get created? (1)

Teammates (16)

Who is the owner or expert for this code? (3)[7]
How do I convince my teammates to do this the
Did my teammates do this? (1)

Policies (15)

What is the policy for doing this? (10) [24]
Is this the correct policy for doing this? (2) [15]
How is the allocation lifetime of this object mai

LaToza and Myers. Hard-to-answer questions about code. PLATEAU 2010.

Investigating code to learn facts

- Developers navigated code to answer questions and learn **facts** about code
  - Examples:
  - Whenever the window scrolls, the caret status must be updated.
  - Whenever the cursor moves, the caret status must be updated.
  - Whenever the buffer changes, the caret status should be updated once.
  - EditBus is for low frequency events, not high frequency events like buffer edits
  - When the buffer change EditBus message is sent, the text area has not yet been updated with the new buffer's info.
- Developers sometimes were **unsuccessful** answering their questions.

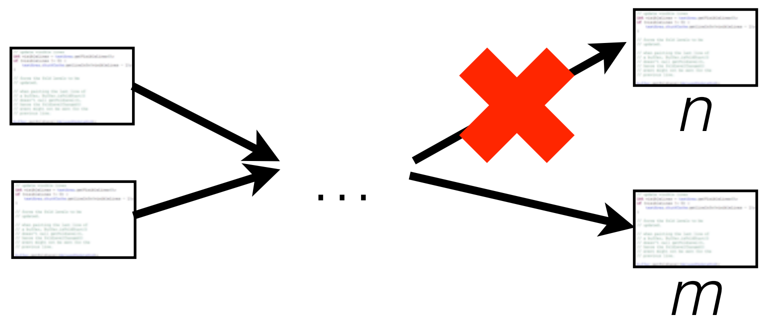
  made optimistic or pessimistic assumptions
- Developers sometimes made **false assumptions**

Thomas D. LaToza and Brad A. Myers. 2010. Developers ask reachability questions. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10), Vol. 1. ACM, New York, NY, USA, 185-194. DOI=http://dx.doi.org/10.1145/1806799.1806829

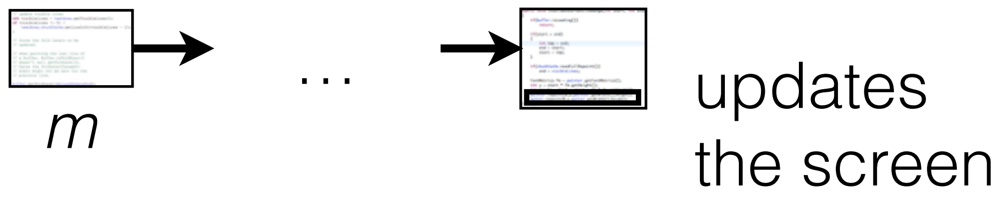
Examples of false beliefs and questions answered incorrectly

False assumption	Correct fact about control flow
Method *m* need not invoke method *n*, as it is only called in a situation in which *n* has already been called.	*m* is called in several additional situations in which *n* has not been called.

…



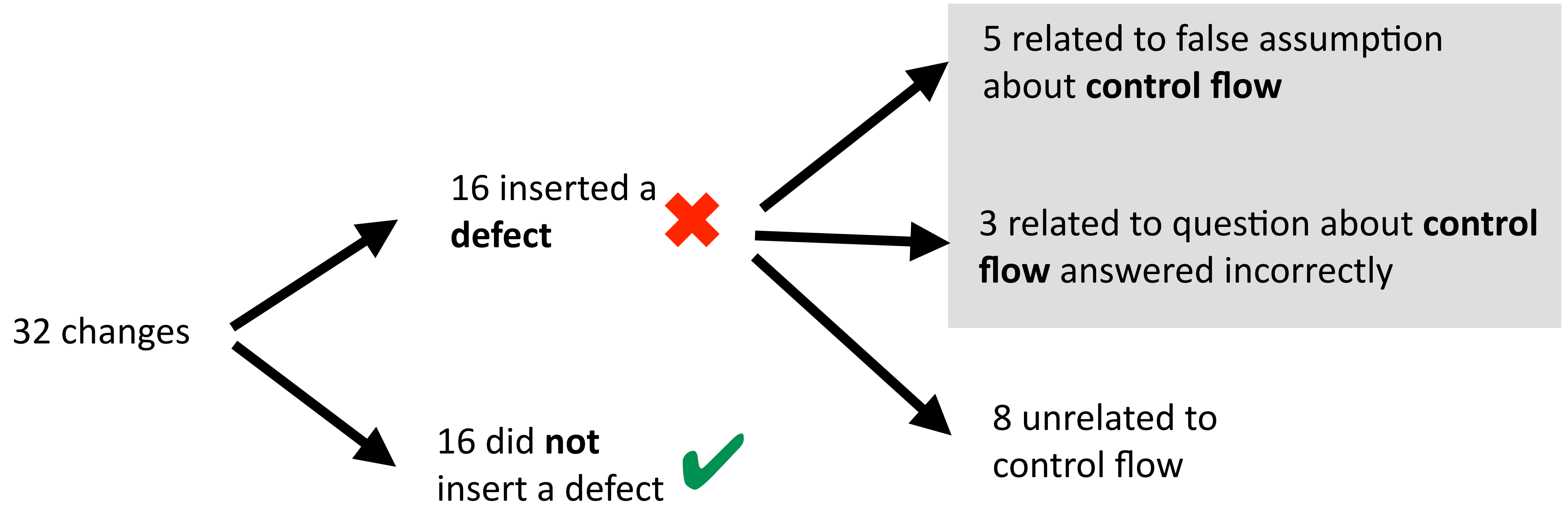
Question answered incorrectly	Correct fact about control flow
Why is calling *m* necessary?	*m* indirectly calls a function that updates the screen.



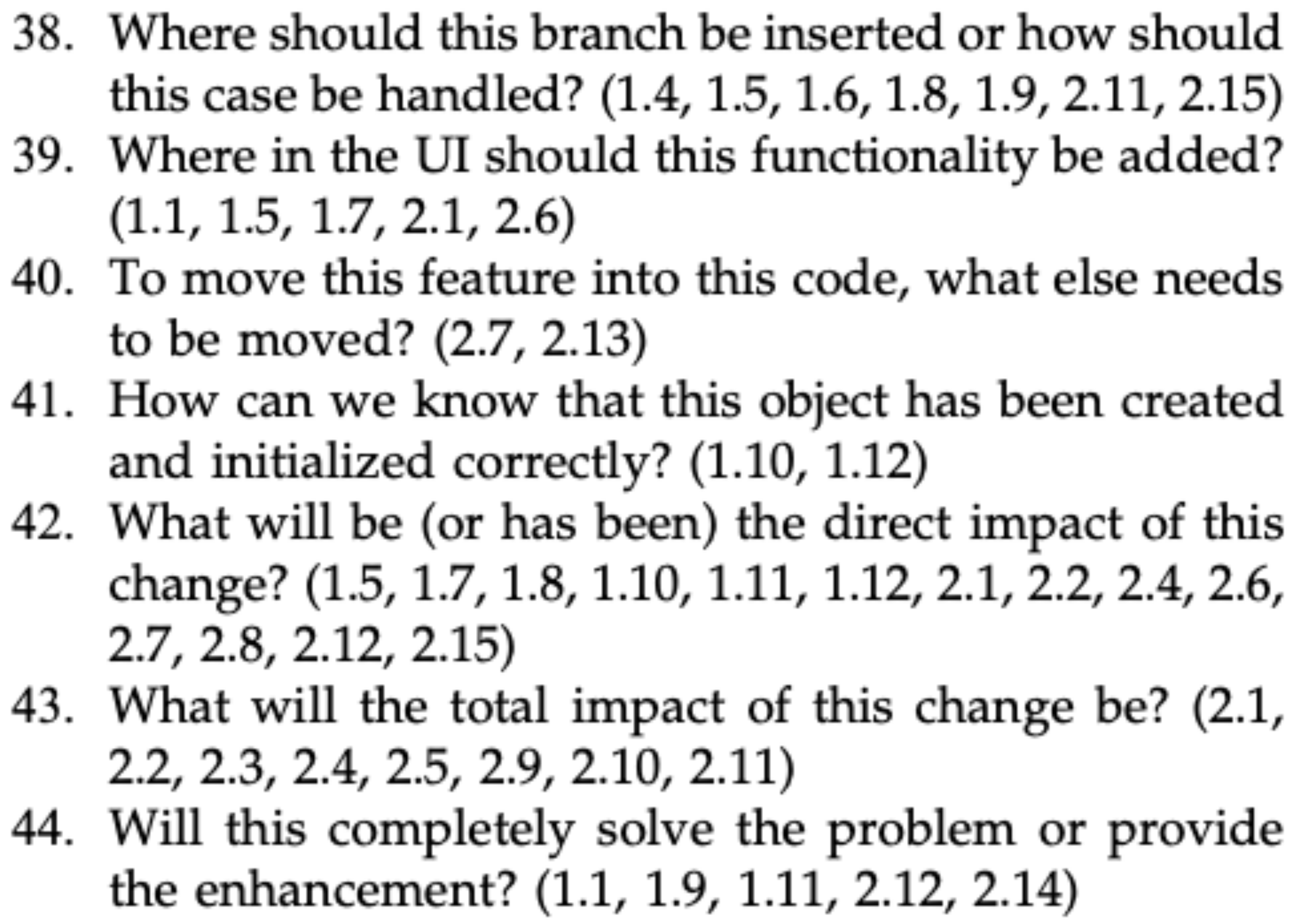
Thomas D. LaToza and Brad A. Myers. 2010. Developers ask reachability questions. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10), Vol. 1. ACM, New York, NY, USA, 185-194. DOI=http://dx.doi.org/10.1145/1806799.1806829

False facts lead to defects

- Developers seek task-relevant information by asking **questions** and **navigating** code to learn facts about code
- Developers built **mental models** (sometimes externalized in sketches and notes) of control flow
- Developers sometimes hold **false beliefs** about code
  because they answered questions incorrectly
  or made false assumptions
- False beliefs about **control flow** led developers to introduce defects

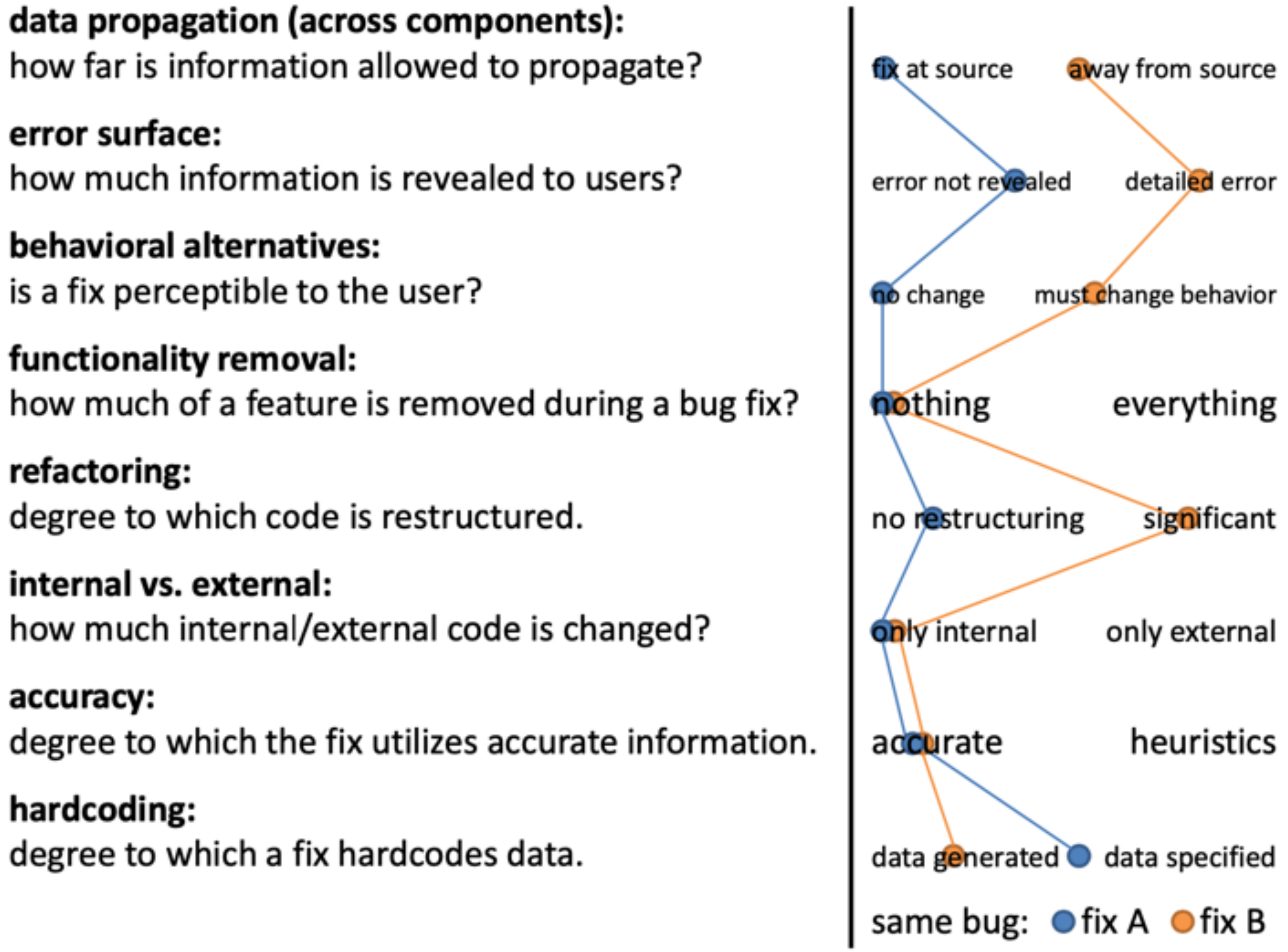

32 changes
16 inserted a **defect** ✗
16 did **not** insert a defect ✓
5 related to false assumption about **control flow**
3 related to question about **control flow** answered incorrectly
8 unrelated to control flow

Impact analysis



38. Where should this branch be inserted or how should this case be handled? (1.4, 1.5, 1.6, 1.8, 1.9, 2.11, 2.15)
39. Where in the UI should this functionality be added? (1.1, 1.5, 1.7, 2.1, 2.6)
40. To move this feature into this code, what else needs to be moved? (2.7, 2.13)
41. How can we know that this object has been created and initialized correctly? (1.10, 1.12)
42. What will be (or has been) the direct impact of this change? (1.5, 1.7, 1.8, 1.10, 1.11, 1.12, 2.1, 2.2, 2.4, 2.6, 2.7, 2.8, 2.12, 2.15)
43. What will the total impact of this change be? (2.1, 2.2, 2.3, 2.4, 2.5, 2.9, 2.10, 2.11)
44. Will this completely solve the problem or provide the enhancement? (1.1, 1.9, 1.11, 2.12, 2.14)

Design space of bug fixes



data propagation (across components):
how far is information allowed to propagate?

error surface:
how much information is revealed to users?

behavioral alternatives:
is a fix perceptible to the user?

functionality removal:
how much of a feature is removed during a bug fix?

refactoring:
degree to which code is restructured.

internal vs. external:
how much internal/external code is changed?

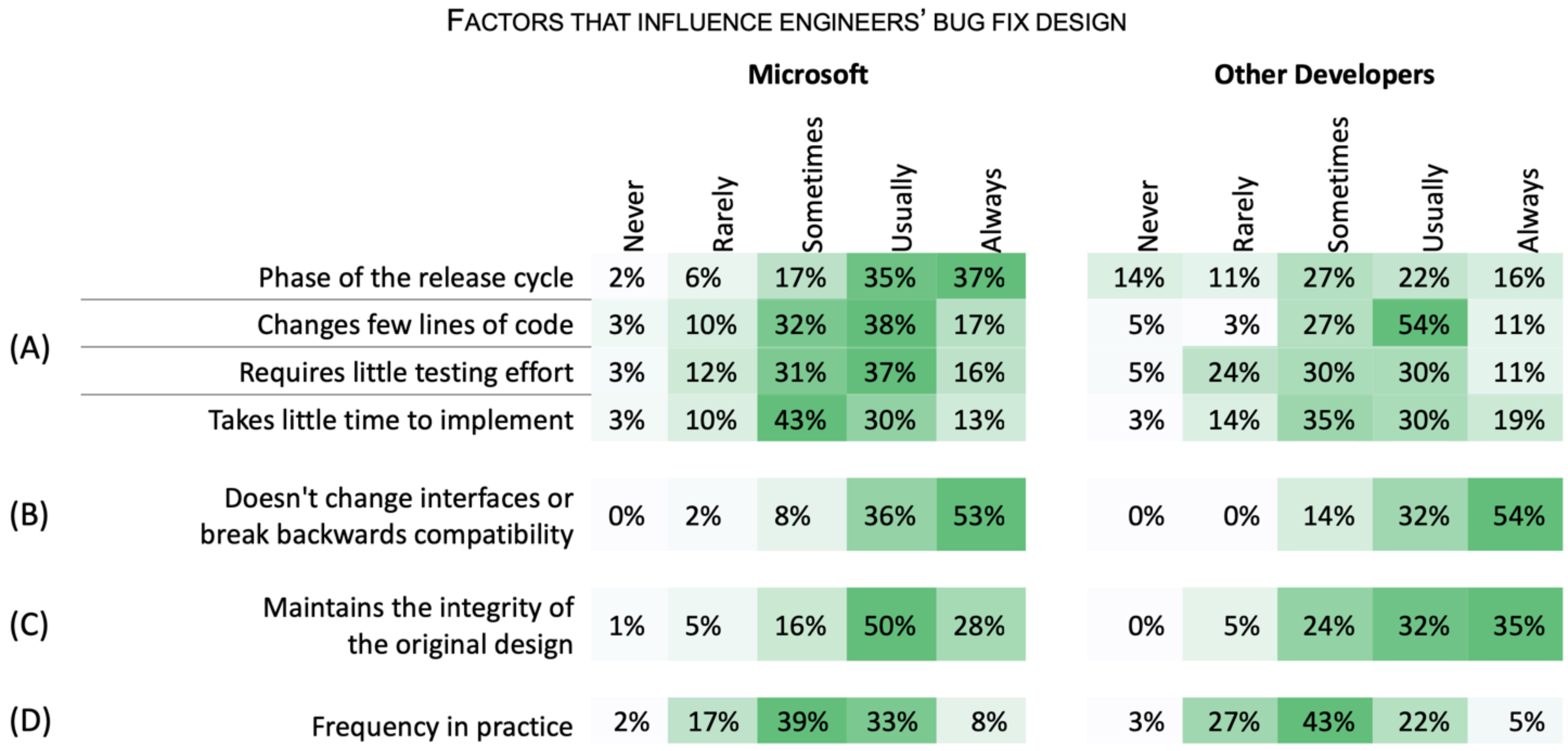
accuracy:
degree to which the fix utilizes accurate information.

hardcoding:
degree to which a fix hardcodes data.

Fig. 3. Two fixes for the same hypothetical bug plotted in our design space.

E. Murphy-Hill, T. Zimmermann, C. Bird and N. Nagappan, "The Design Space of Bug Fixes and How Developers Navigate It," in *IEEE Transactions on Software Engineering*, vol. 41, no. 1, pp. 65-81, 1 Jan. 2015. doi: 10.1109/TSE.2014.2357438

Design space of bug fixes



FACTORS THAT INFLUENCE ENGINEERS' BUG FIX DESIGN

		Microsoft					Other Developers				
		Never	Rarely	Sometimes	Usually	Always	Never	Rarely	Sometimes	Usually	Always
(A)	Phase of the release cycle	2%	6%	17%	35%	37%	14%	11%	27%	22%	16%
	Changes few lines of code	3%	10%	32%	38%	17%	5%	3%	27%	54%	11%
	Requires little testing effort	3%	12%	31%	37%	16%	5%	24%	30%	30%	11%
	Takes little time to implement	3%	10%	43%	30%	13%	3%	14%	35%	30%	19%
(B)	Doesn't change interfaces or break backwards compatibility	0%	2%	8%	36%	53%	0%	0%	14%	32%	54%
(C)	Maintains the integrity of the original design	1%	5%	16%	50%	28%	0%	5%	24%	32%	35%
(D)	Frequency in practice	2%	17%	39%	33%	8%	3%	27%	43%	22%	5%

E. Murphy-Hill, T. Zimmermann, C. Bird and N. Nagappan, "The Design Space of Bug Fixes and How Developers Navigate It," in *IEEE Transactions on Software Engineering*, vol. 41, no. 1, pp. 65-81, 1 Jan. 2015. doi: 10.1109/TSE.2014.2357438

Do developers do impact analysis?

- Do developers believe that they do it?
  - “I always try to understand how I can influence the code. If Im uncertain about my changes I can make a list of influenced part and give it to our QA[Quality Assurance] engineers. They are checking all cases."
  - "I have to make sure that my change will not cause bugs or other problems for other parts of the project or systems components ... "
  - "After my changes I have to find direct and indirect calls of this method and make sure that system will be ok after my changes."

Siyuan Jiang, Collin Mcmillan, and Raul Santelices. 2017. Do Programmers do Change Impact Analysis in Debugging?. Empirical Softw. Engg. 22, 2 (April 2017), 631-669. DOI: https://doi.org/10.1007/s10664-016-9441-9

Techniques for impact analysis

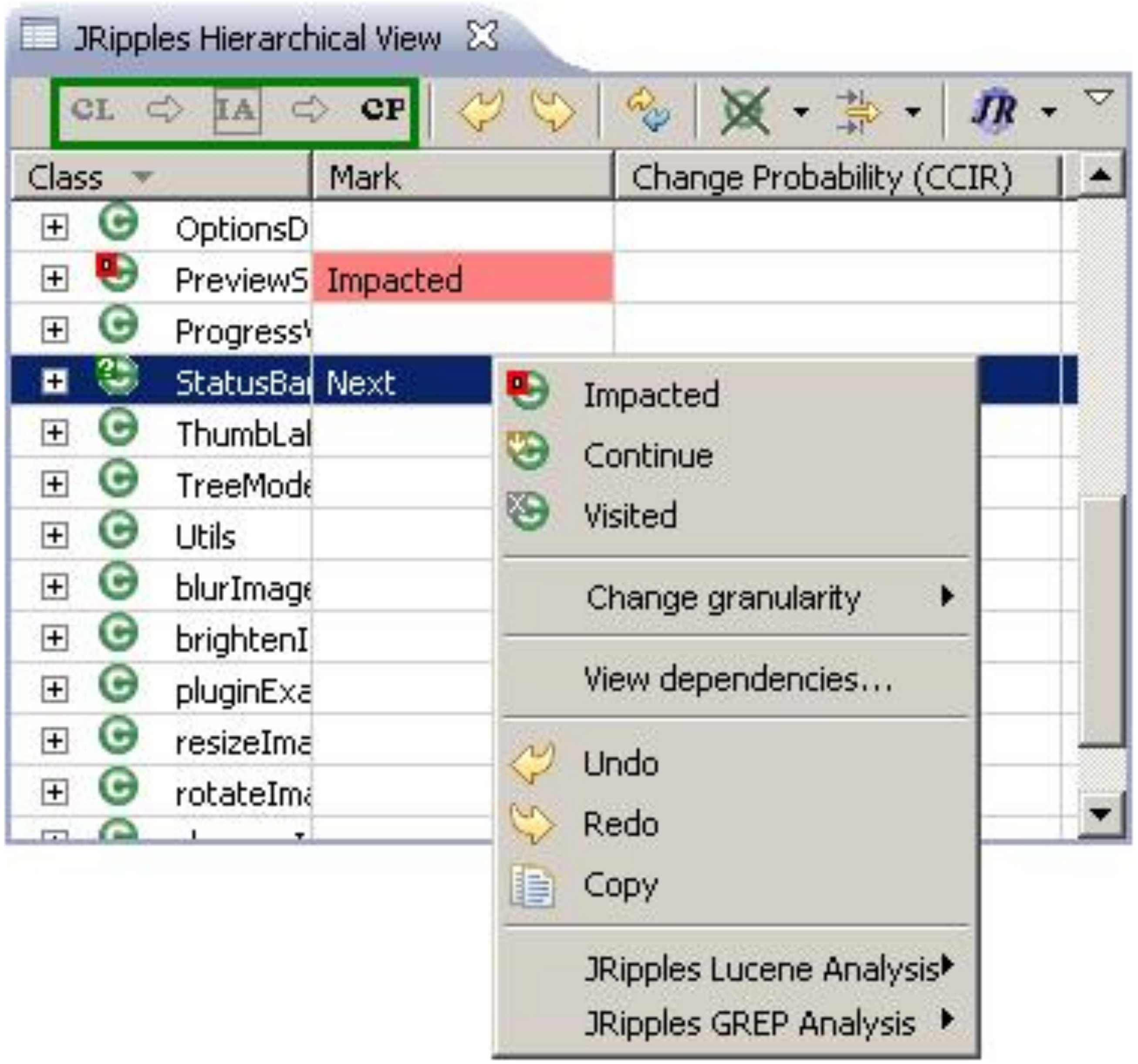
- Find element that changes
- Find related elements where change might "ripple" to impact
- Show to user related elements to inspect

- Elements
  - method
  - statement in slice
  - class in UML diagram

- Many approaches
  - Lehnert identified 150

Lehnert, Steffen. “A review of software change impact analysis.” (2011).

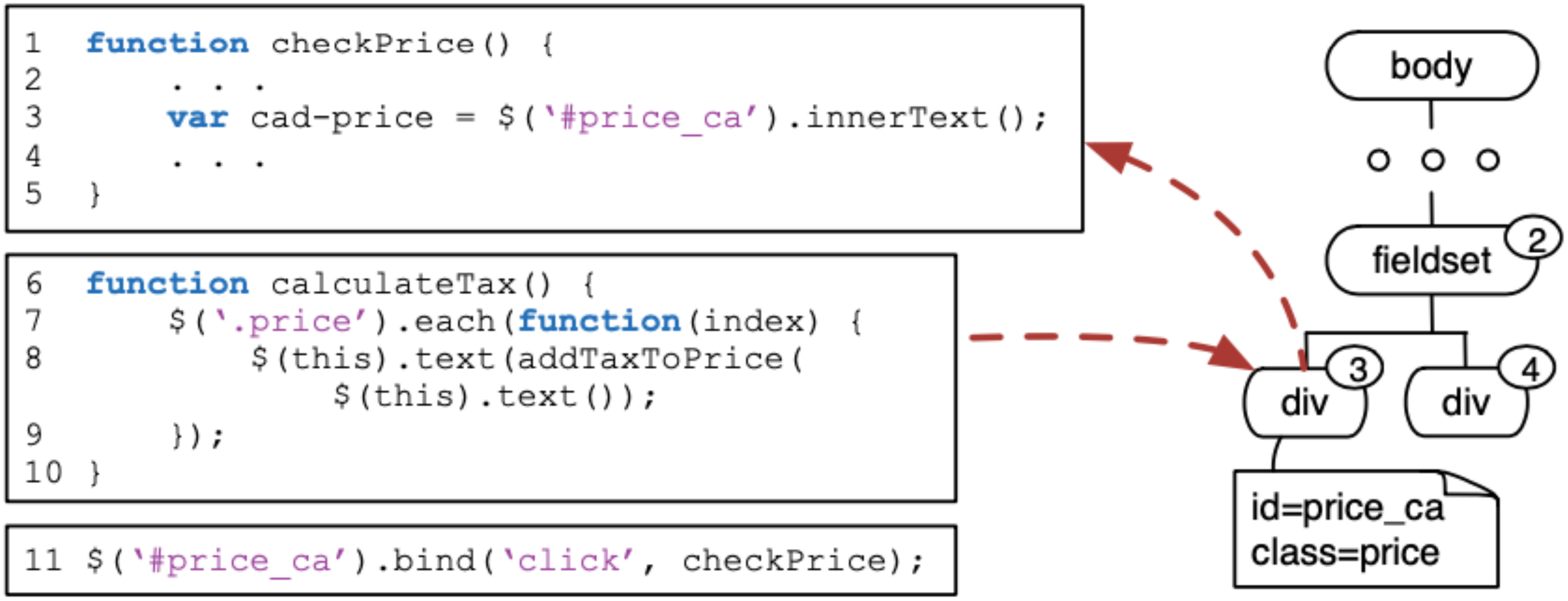
JRipples



Class	Mark	Change Probability (CCIR)
OptionsD		
PreviewS	Impacted	
Progress\		
StatusBa	Next	
ThumbLal		
TreeMode		
Utils		
blurImage		
brightenI		
pluginExa		
resizeIma		
rotateIma		

http://jripples.sourceforge.net/

Impact in JavaScript



```
function checkPrice() {
    . . .
    var cad-price = $('#price_ca').innerText();
    . . .
}
```

```
function calculateTax() {
    $('.price').each(function(index) {
        $(this).text(addTaxToPrice(
            $(this).text());
    });
}
```

```
$('#price_ca').bind('click', checkPrice);
```

Figure 3 Impact transfer through DOM elements.

Alimadadi, Saba, Ali Mesbah and Karthik Pattabiraman. “Hybrid DOM-Sensitive Change Impact Analysis for JavaScript.” *ECOOP* (2015).

Tracking Impact in JavaScript

```
function checkPrice() {
  var itemName = extractName($('#item231'));
  var cadPrice = $('#price_ca').innerText;
  $.ajax({
     url : "prices/latest.php",
     type : "POST",
     data : itemName,
     success : eval(getAction() + "Item")
  });
  confirmPrice();
}
function updateItem(xhr) {
  var updatedInfo = getUpdatedPrice(xhr.responseText);
  suggestItem.apply(this, updatedInfo);
}
function suggestItem() {
  if (arguments.length > 2) {
    displaySuggestion(arguments1);
  }
}
function calculateTax() {
  $(".price").each(function(index) {
    $(this).text(addTaxToPrice($(this).text()));
  });
}
$("#price-ca").bind("click", checkPrice);
$("prices").bind("click", calculateTax);
```

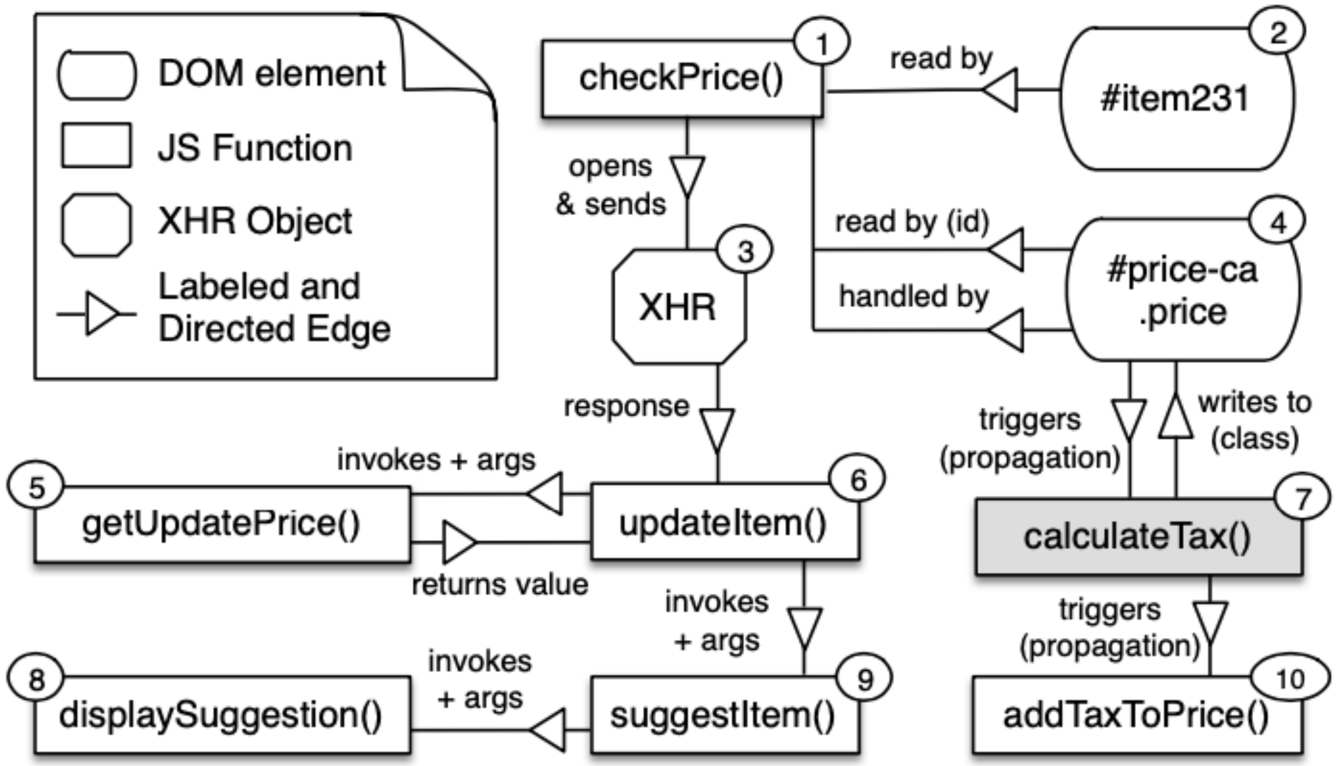
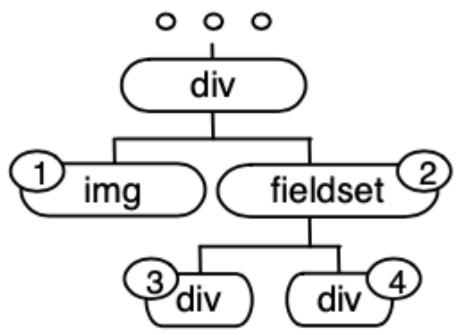


Figure 1 Motivating example: JavaScript code

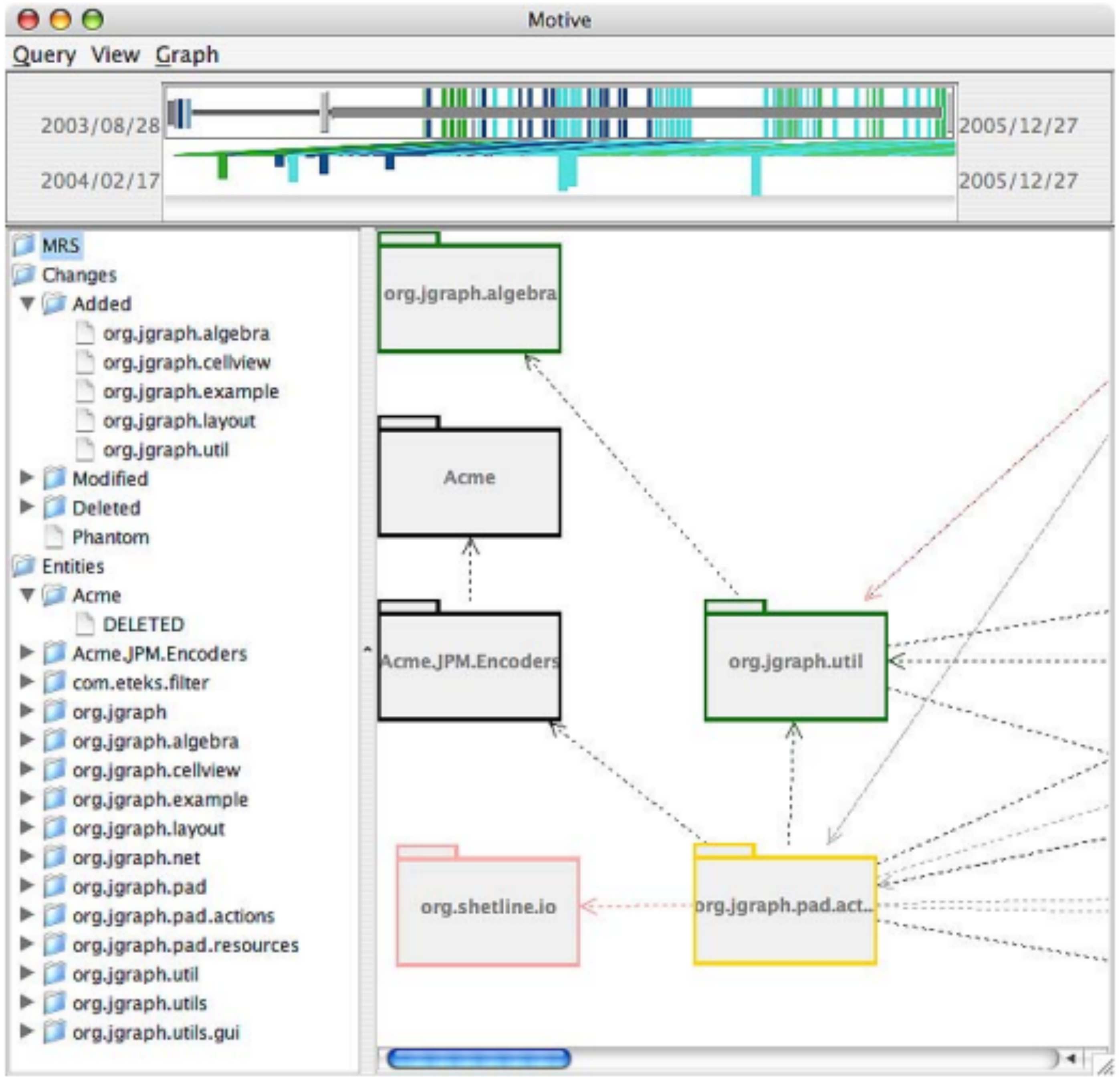
```
<img id='item231' src='img/items/231.png'
          itemName='dress'/>
<fieldset name='prices'>
    <div class='price' id='price-ca'>120</div>
    <div class='price' id='price-us'>110</div>
</fieldset>
```



Alimadadi, Saba, Ali Mesbah and Karthik Pattabiraman. “Hybrid DOM-Sensitive Change Impact Analysis for JavaScript.” *ECOOP* (2015).

Visualizing architectural changes

green: added
yellow: modified
black: deleted
pink: phantom
grey: unchanged



Andrew McNair, Daniel M. German, and Jens Weber-Jahnke. 2007. Visualizing Software Architecture Evolution Using Change-Sets. In Proceedings of the 14th Working Conference on Reverse Engineering (WCRE '07). IEEE Computer Society, Washington, DC, USA, 130-139. DOI=http://dx.doi.org/10.1109/WCRE.2007.52