Detecting Defects

SWE 795, Fall 2019

Software Engineering Environments

GEORGE MASON UNIVERSITY

Today

- Part 1 (Lecture)(~80 mins)

- Break!

- Part 2 (Discussion)(~60 mins)
    - Discussion of readings

Detecting Defects

- Where do defects come from?
- How can defects be prevented?
- How should potential defects be communicated to developers?

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

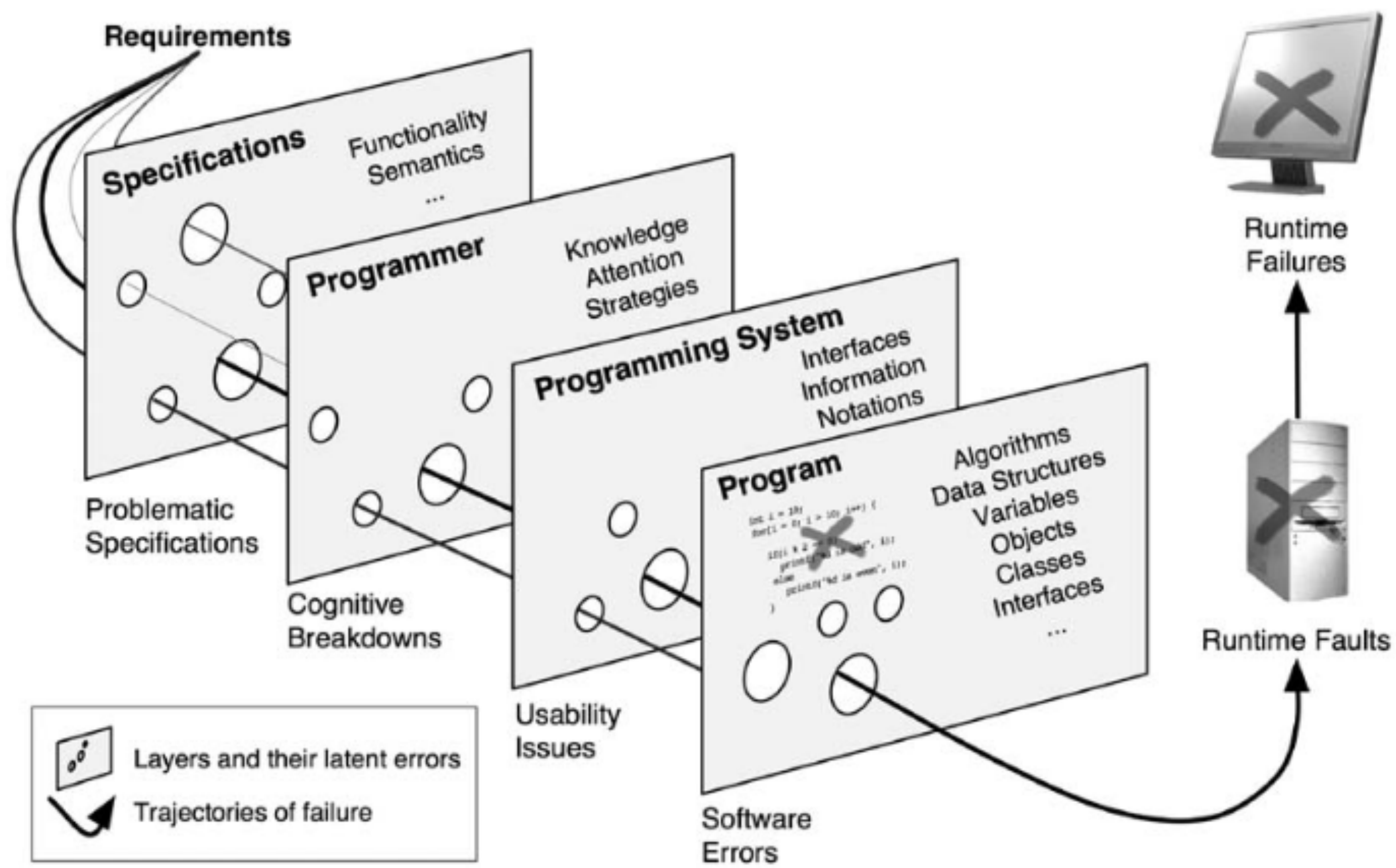
Where do defects come from?

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

Where do defects come from?

- Ko & Myers proposed a model for understanding the *cognitive* causes of defects
- Latent errors becomes active errors when they breach defenses of system



Adapted from Ko & Myers, JVLC05

Skill / Rule / Knowledge

- James Reason proposed a taxonomy of cognitive breakdowns based on differences in type of cognition being used
- Skill-based activity: routine, proceduralized activity
  - e.g., typing a string, opening a source file, compiling a program
- Rule-based activity: use of rules for acting in certain contexts
  - e.g., starting to type a for loop in order to perform an action on each element of a list
- Knowledge-based activity: forming plans & making high-level decisions based on knowledge of program
  - e.g., forming a hypothesis about cause of runtime failure

Adapted from Ko & Myers, JVLC05

Types of skill breakdowns

Inattention	Type	Events resulting in breakdown
Failure to attend to a routine action at a critical time causes forgotten actions, forgotten goals, or inappropriate actions.	Strong habit intrusion	In the middle of a sequence of actions → no attentional check → contextually frequent action is taken instead of intended action
	Interruptions	External event → no attentional check → action skipped or goal forgotten
	Delayed action	Intention to depart from routine activity → no attentional check between intention and action → forgotten goal
	Exceptional stimuli	Unusual or unexpected stimuli → stimuli overlooked → appropriate action not taken
	Interleaving	Concurrent, similar action sequences → no attentional check → actions interleaved

Overattention	Type	Events resulting in breakdown
Attending to routine action causes false assumption about progress of action.	Omission	Attentional check in the middle of routine actions → assumption that actions are already completed → action skipped
	Repetition	Attentional check in the middle of routine actions → assumption that actions are not completed → action repeated

Adapted from Ko & Myers, JVLC05

Types of rule breakdowns

Wrong rule	Type	Events resulting in breakdown
Use of a rule that is successful in most contexts, but not all.	Problematic signs	Ambiguous or hidden signs → conditions evaluated with insufficient info → wrong rule chosen → inappropriate action
	Information overload	Too many signs → important signs missed → wrong rule chosen → inappropriate action
	Favored rules	Previously successful rules are favored → wrong rule chosen → inappropriate action
	Favored signs	Previously useful signs are favored → exceptional signs not given enough weight → wrong rule chosen → inappropriate action
	Rigidity	Familiar, situationally inappropriate rules preferred over unfamiliar, situationally appropriate rules → wrong rule chosen → inappropriate action

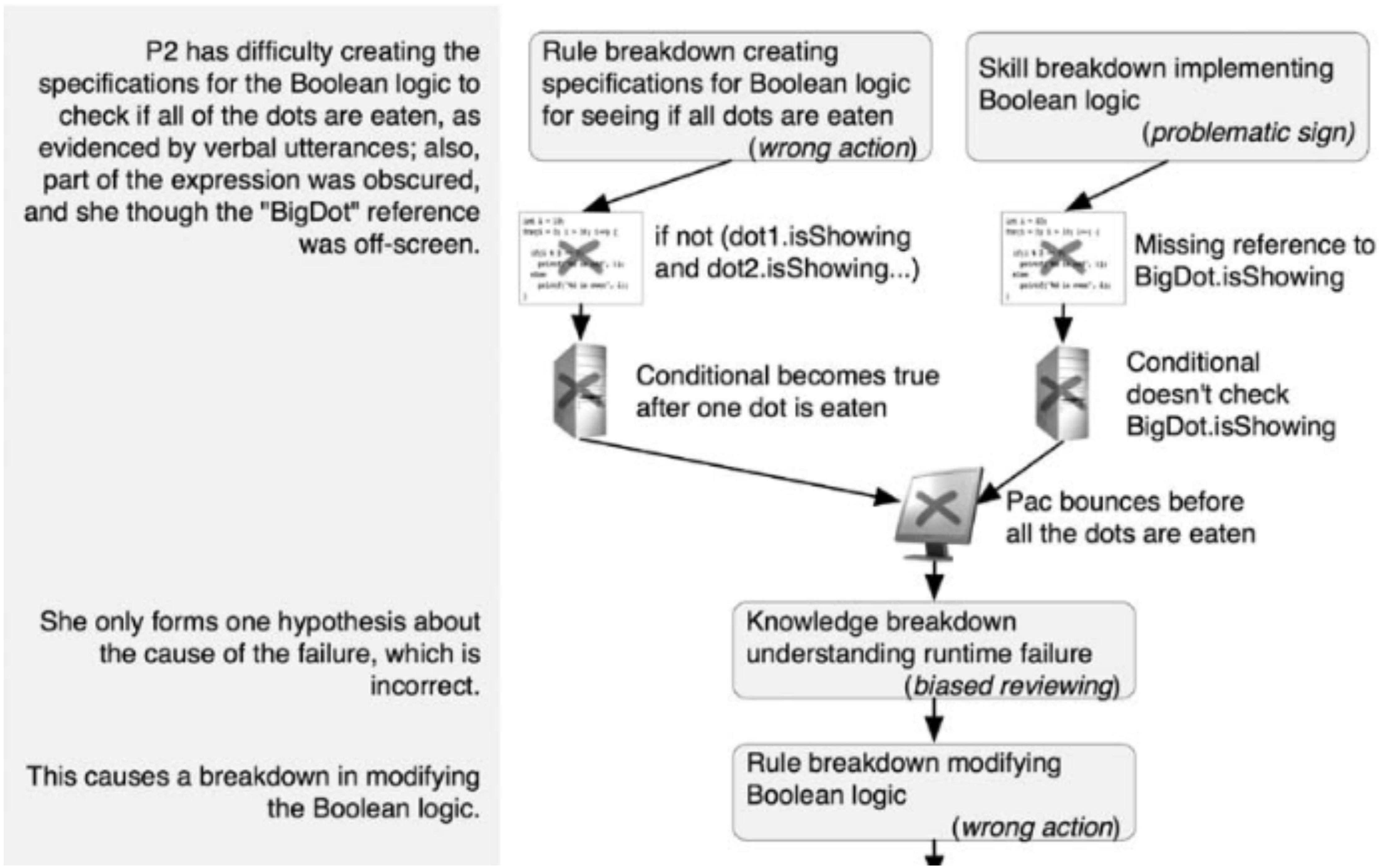
Bad rule	Type	Events resulting in breakdown
Use of a rule with problematic conditions or actions.	Incomplete encoding	Some properties of problem space are not encoded → rule conditions are immature → inappropriate action
	Inaccurate encoding	Properties of problem space encoded inaccurately → rule conditions are inaccurate → inappropriate action
	Exception proves rule	Inexperience → exceptional rule often inappropriate → inappropriate action
	Wrong action	Condition is right but action is wrong → inappropriate action

Types of knowledge breakdowns

Bounded rationality	Type	Events resulting in breakdown
Problem space is too large to explore because working memory is limited and costly.	Selectivity	Psychologically salient, rather than logically important task information is attended to → biased knowledge
	Biased reviewing	Tendency to believe that all possible courses of action have been considered, when in fact very few have been considered → suboptimal strategy
	Availability	Undue weight is given to facts that come readily to mind → facts that are not present are easily ignored → biased knowledge

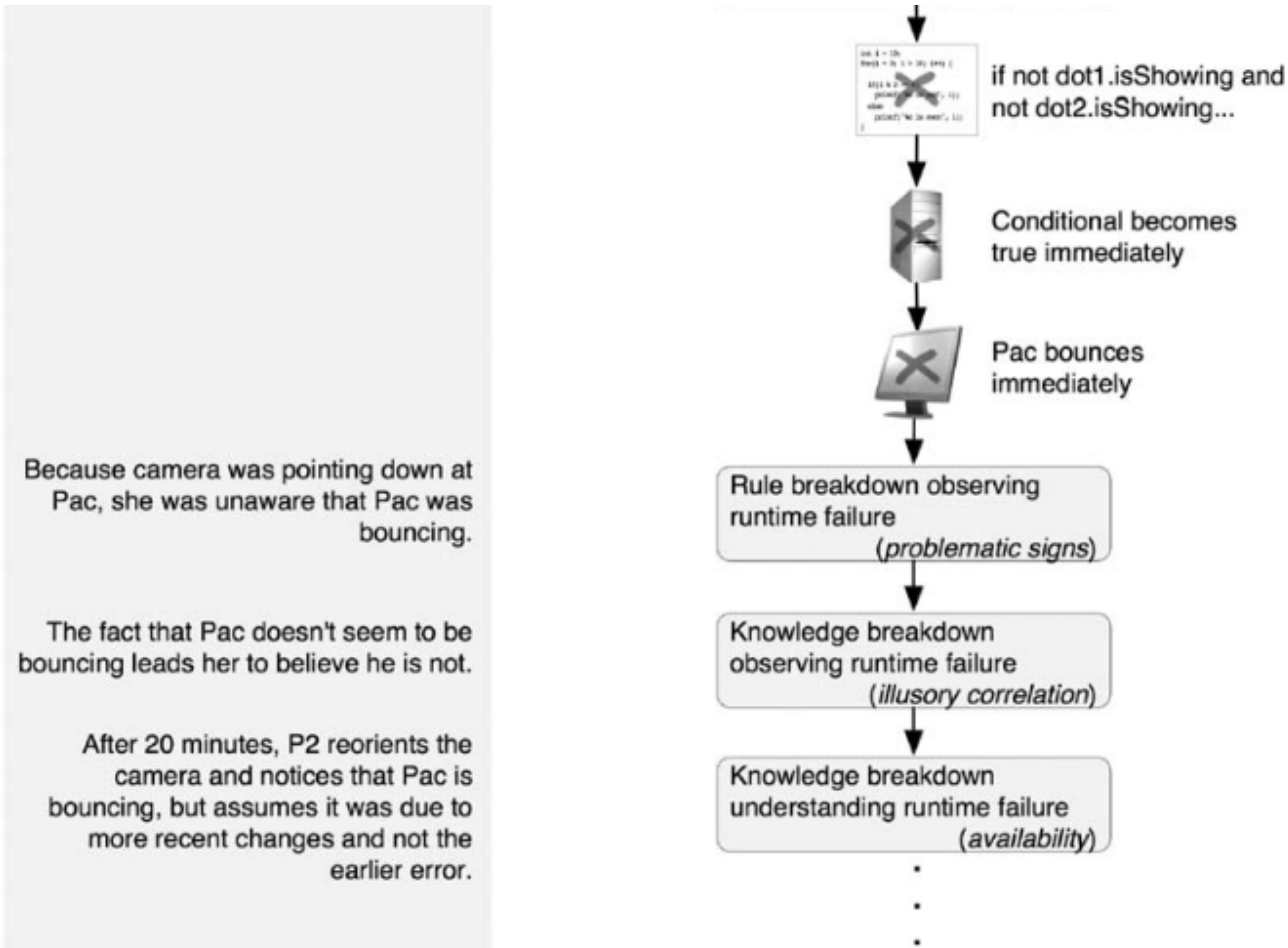
Faulty models of problem space	Type	Events resulting in breakdown
Formation and evaluation of knowledge leads to incomplete or inaccurate models of problem space.	Simplified causality	Judged by perceived similarity between cause and effect → knowledge of outcome increases perceived likelihood → invalid knowledge of causation
	Illusory correlation	Tendency to assume events are correlated and develop rationalizations to support the belief → invalid model of causality
	Overconfidence	False belief in correctness and completeness of knowledge, especially after completion of elaborate, difficult tasks → invalid, inadequate knowledge
	Confirmation bias	Preliminary hypotheses based on impoverished data interfere with later interpretation of more abundant data → invalid, inadequate hypotheses

Breakdown chain example (Part 1)



Adapted from Ko & Myers, JVLC05

Breakdown chain example (Part 1)



Adapted from Ko & Myers, JVLC05

Causes of defects: API misuse

- Components expose APIs which have rules about how they should be used
- What types of rules do components impose?

Causes of defects: API misuse

- Based on survey of APIs, categorized directives APIs impose on clients
- Restrictions on when to call
  - Do not call from UI thread, for debugging use only
- Protocols specifying ordering constraints
  - Method must only be called once, method must be called prior to other method
- Locking describing thread synchronization
- Restrictions on possible parameter values
  - String.replaceAll() should not include $ or \ characters in replacement string

Uri Dekel and James D. Herbsleb. 2009. Improving API documentation usability with knowledge pushing. In *Proceedings of the 31st International Conference on Software Engineering* (ICSE ’09), 320-330.

Causes of defects: Object protocol misuse

- Examined Java code for presence of protocols, found 7.2% of types defined protocols & 13% of classes used protocols
- Most frequent causes:
  - Initialization (28.1%): calls to an instance method *m* without first calling initializing method *i*
  - Deactivation (25.8%): calls to an instance method *m* after calling a deactivation method *d*
  - Type Qualifier (16.4%): object enters a state during which method *m* will always fail

Nels E. Beckman, Duri Kim, and Jonathan Aldrich. 2011. An empirical study of object protocols in the wild. In *Proceedings of the 25th European conference on Object-oriented programming* (ECOOP'11), Mira Mezini (Ed.). Springer-Verlag, Berlin, Heidelberg, 2-26.

Causes of defects in JavaScript

- Examined 502 bug reports from 19 repos, categorizing the cause of each error
- Most common types of errors:
  - Erroneous input validation (16%): inputs passed into JS code are not validated or sanitized
  - Error in writing a string literal (13%): incorrect CSS selectors, regular expressions, forgetting prefixes, etc.
  - Forgetting null / undefined check (10%)
  - Neglecting differences in browser behavior (9%): differences in behavior of browser API across browsers
  - Errors in syntax (7%)

Ocariza et al, A Study of Causes and Consequences of Client-Side JavaScript Bugs, TSE 2016

40% OBJECT-INTERACTION IDIOMS

21% **VALID REFERENCES** DETERMINING DEFINED STANDARD OR FRAMEWORK IDENTIFIERS AT COMPILE TIME OR RUNTIME

OB1 *Errorful Invalid Reference:*
statement generating error & error message → explanation of error message

OB2 *Errorless Invalid Reference:*
error message → statement generating error and error message

20% **COLLECTIONS AND FORMATS** CREATING OR MANIPULATING A COLLECTION, OR FORMATTING DATA FOR USE IN A FRAMEWORK

OB4 *Iteration Construct:*
collection object → corresponding iteration construct

OB5 *Concurrent Modifications:*
collection object & loop fragment → concurrent modifications of collection

OB6 Format Conversion:
object in format A → object in format B

16% **BACK-END REQUESTS** SENDING STRUCTURED DATA TO A SERVER, OR HANDLING SERVER RESPONSES

OB10 *Back-end request configuration:*
back-end request & desired behavior → modified request matching behavior

OB11 *Transmission mutations:*
back-end request as sent → back-end request as received

OB12 *Receiving Data:*
back-end request → code fragment for listening for response(s)

8% **METHOD CHAINS** DETERMINING THE EFFECTS OF A METHOD INVOCATION WITHIN A SEQUENCE OF CONSECUTIVE CALL EXPRESSIONS

OB7 *Incomplete Sequence:*
o.m1(...).m2(...)....mn(....) → o.m1(...).m2(...)....mk(...)....mn(...)

OB8 *Incorrect Sequence:*
o.m1(...).m2(...)....mn(....) → o.mk(...)....m1(...).mn(...)

OB9 *Overridden Effect:*
o.m1(...).m2(...)....mn(....) → methods mk and ml both mutate object p

8% **SCOPE CONTEXTS** DETERMINING THE CONTEXT GIVEN TO THE KEYWORD `this` WITHIN A CODE BLOCK OR A VARIABLE'S VISIBILITY

OB3 *"this" Scope:*
`this` statement → scope defining `this`

42% GRAPHICAL IDIOMS

37% **GRAPHICAL SETTERS** CHANGING GRAPHICAL PROPERTIES OF THE DOM VIA API METHODS OR CSS PROPERTIES

GB5 *Unidentified Setter:*
visual property change → code fragment to mutate property

GB6 *Unobservable Setter:*
setterA and visual property change → setterB to mutate property

GB7 *Indirect Setter:*
setterA → elements which inherit properties from setterA or occlude elements mutated by setter

GB8 *Conflicted Setter:*
setterA → setterB which overwrites setterA and code fragment with alternative setter or sequencing

21% **GRAPHICAL QUERIES** RETRIEVING DOM ELEMENTS OR SIMILAR REPRESENTATIONS VIA QUERY METHODS OR CSS SELECTORS

GB1 *Incomplete Queries:*
queryA and elements to be matched → queryB matching only elements

GB2 *Live Query Results:*
queryA → changes to query result set over time and alternative fragment

GB3 *Overwritten Query:*
queryA → queryB intersecting mutations made by queryA and queryB' which does not

8% **GRAPHICAL GETTERS** OBTAINING GRAPHICAL PROPERTIES OF THE DOM VIA API METHODS

GB4 *Unidentified Getter:*
visual property → getter code fragment to retrieve

51% CALLBACK IDIOMS

29% **BINDING TARGETS** IDENTIFYING OR CHOOSING AN EVENT, LIFECYCLE HOOK, OR TRIGGER TO REGISTER A CALLBACK

CB1 *Unidentified Target:*
desired target → target name & code fragment

CB2 *Constrained Target:*
binding target code fragment → framework rules making fragment invalid

CB3 *Confused Target:*
binding target & desired target → explanation of difference, new target name, code fragment

25% **CALL BACK CONTEXTS** IDENTIFYING WHEN THE CALLBACK IS DISPATCHED, USING ITS ARGUMENTS OR OTHER RELATED OBJECTS

CB6 *Improper Scheduling:*
call back code fragments and desired state → ordering of call backs and state accessibility

CB7 *Unidentified State:*
desired state → code fragment to obtain state

CB8 *Missed Call Backs:*
call back code fragment → framework state required for call back to occur

23% **BINDER CONFIGURATIONS** SETTING PROPERTIES OF A CALLBACK TRIGGER, OR MODIFYING PARAMETERS OF ITS BINDING MECHANISM

CB4 *Incorrect Parameters:*
call back configuration code fragments and desired behavior → updated call back configuration code

CB5 *Misconfigured Framework:*
framework configuration code fragments and desired behavior →updated framework configuration code

42%

GRAPHICAL IDIOMS

37%

GRAPHICAL SETTERS CHANGING GRAPHICAL PROPERTIES OF THE DOM VIA API METHODS OR CSS PROPERTIES

GB5 *Unidentified Setter:*

visual property change → code fragment to mutate property

GB6 *Unobservable Setter:*

setterA and visual property change → setterB to mutate property

GB7 *Indirect Setter:*

setterA → elements which inherit properties from setterA or occlude elements mutated by setter

GB8 *Conflicted Setter:*

setterA → setterB which overwrites setterA and code fragment with alternative setter or sequencing

21%

GRAPHICAL QUERIES RETRIEVING DOM ELEMENTS OR SIMILAR REPRESENTATIONS VIA QUERY METHODS OR CSS SELECTORS

GB1 *Incomplete Queries:*

queryA and elements to be matched → queryB matching only elements

GB2 *Live Query Results:*

queryA → changes to query result set over time and alternative fragment

GB3 *Overwritten Query:*

queryA → queryB intersecting mutations made by queryA and queryB' which does not

8%

GRAPHICAL GETTERS OBTAINING GRAPHICAL PROPERTIES OF THE DOM VIA API METHODS

GB4 *Unidentified Getter:*

visual property → getter code fragment to retrieve

51%

CALLBACK IDIOMS

29%

BINDING TARGETS IDENTIFYING OR CHOOSING AN EVENT, LIFECYCLE HOOK, OR TRIGGER TO REGISTER A CALLBACK

Some techniques for helping developers better work with defects

- Help developers engage in better information seeking to prevent defects from ever occurring

- Use tool to find defect, report error message to developer

- Use tests to find defect, report test failures to developers

Preventing defects by supporting better information seeking

1. Help programmers recover from interruptions or delays by **reminding** them of their previous actions
2. Highlight **exceptional** circumstances to help programmers adapt their routine strategies
3. Help programmers manage **multiple tasks** and detect interleaved actions
4. Design task-relevant information to be visible and unambiguous
5. Avoid **inundating** programmers with information
6. Help programmers consider all relevant **hypotheses**, to avoid the formation of invalid hypotheses
7. Help programmers identify and understand **causal relationships**, to avoid invalid knowledge
8. Help programmers identify **correlation** and recognize illusory correlation
9. Highlight **logically** important information to combat availability and selectivity heuristics
10. **Prevent** programmer's **overconfidence** in their knowledge by testing their assumptions

Adapted from Ko & Myers, JVLC05

Tools for preventing defects

- Early work in program analysis and formal methods made possible analyzing code to find inconsistencies with a specification
- But...
  - Often required extensive work to write a specification of behavior

Early 2000s

- Static analysis tools becoming sufficiently scalable to be used on real-world programs
- More emphasis on finding real-world defects rather than simply focusing on improvements in underlying analysis technology
- Several tools adopted in industry, often to address specific and important problems

Slam

Rules governing lock

```
state {
   enum { Unlocked, Locked} s = Unlocked;   // FSM states
}
AcquireSpinLock.entry {          // Transition on lock acquire
   if (s == Locked) error;
   else s = Locked;
}
ReleaseSpinLock.entry {          // Transition on lock release
   if (s == Unlocked) error;
   else s = Unlocked;
}
```

Iteratively refines boolean abstraction of program to determine if there exists path that violates rules

```
void example() {
do {
A: AcquireSpinLock();
    nPacketsOld = nPackets;
    req = devExt->WLHV;
    if (req && req->status) {
          devExt = req->Next;
B:        ReleaseSpinLock();
          irp = req->irp;
          if (req->status > 0)
                irp->IoS.Status = S;
          else
                irp->IoS.Status = F;
          nPackets++;
          }
    } while(nPackets!=nPacketsOld);
C: ReleaseSpinLock();
}
```

(a)

```
void example() {
do {
A:  AcquireSpinLock();
    skip;
    skip;
    if (*) {
        skip;
B:      ReleaseSpinLock();
        skip;
        if (*)
          skip;
        else
          skip;
        skip;
        }
    } while (*);
C: ReleaseSpinLock();
}
```

(b)

```
void example() {
do {
A:  AcquireSpinLock();
    b := true;
    skip;
    if (*) {
        skip;
B:      ReleaseSpinLock();
        skip;
        if (*)
          skip;
        else
          skip;
        b := b ? false : *;
        }
    } while (!b);
C: ReleaseSpinLock();
}
```

(c)

T. Ball and S.K. Rajamani, "The Slam Project: Debugging System Software via Static Analysis," *Proc. 29th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages* (POPL 2002), ACM Press, 2002, pp. 1–3.

docs.microsoft.com/en-us/windows-hardware/drivers/devtest/static-driver-verifier

Microsoft | Hardware Dev Center Explore Docs Downloads Events Samples Support Dashboard

Docs / Windows / Windows Drivers / Driver Technologies / Tools for Testing Drivers   Bookmark Feedback Edit Shar

Static Driver Verifier

06/13/2019 • 2 minutes to read •

Static Driver Verifier (also known as "StaticDV" or "SDV") is a static verification tool that systematically analyzes the source code of Windows kernel-mode drivers. SDV is a compile time tool that is capable of discovering defects and design issues in a driver. Based on a set of interface rules and a model of the operating system, SDV determines whether the driver correctly interacts with the Windows operating system kernel.

Installing Static Driver Verifier

Static Driver Verifier is available as part of the Windows Driver Kit (WDK) in both the full WDK experience and in the standalone Enterprise WDK. In addition, the Visual C++ Redistributable Packages for Visual Studio are required for SDV to run. See the following:

- Visual Studio 2019 Redistribution
- Visual C++ Redistributable Packages for Visual Studio 2017
- Visual C++ Redistributable Packages for Visual Studio 2013

For versions of SDV available in the WDK for Windows 10, Version 1809 or earlier, the Visual C++ Redistributable Packages for Visual Studio 2012 should be installed instead of the 2017 packages.

Visual Studio Integration

Static Driver Verifier is integrated into Visual Studio. You can run static analysis on your Visual Studio driver project. You can launch, configure, and control Static Driver Verifier from the **Driver** menu in Visual Studio.

Static Driver Verifier Documentation

- Static Driver Verifier Known Issues: Lists latest known issues for Static Driver Verifier
- Using Static Driver Verifier to Find Defects in Drivers: Tells you what you need to get started analyzing your driver code in the Visual Studio environment.
- Static Driver Verifier commands (MSBuild): Lists the MSBuild commands to use to run SDV in a Visual Studio Command Prompt window.
- Introducing Static Driver Verifier: Provides an overview of the static analysis tool.
- Using Static Driver Verifier: Provides the details about using and configuring the static analysis tool.
- Static Driver Verifier Report: Describes the viewer that displays the detailed trace of the static code analysis.
- Static Driver Verifier Rules: The rules define the requirements for proper interaction between a driver model and the kernel interface of the operating system.
- Static Driver Verifier Reference: Provides reference information about the function role types, SDV configuration files, error, and warning messages.

Rules for Audio Drivers

05/20/2018 • 2 minutes to read •

The DDI compliance rules for audio (PortCls) miniport drivers verify the DDI interface between PortCls.sys and its miniport drivers.

In this section

Topic	Description
PcAddAdapterDevice	The PcAddAdapterDevice rule specifies that a PortCls miniport driver correctly uses the **PcAddAdapterDevice** function, specifically that the *DeviceExtensionSize* should be either zero (0) or no less than PORT_CLASS_DEVICE_EXTENSION_SIZE.
PcAllocateAndMapPages	The PcAllocateAndMapPages rule specifies that a PortCls miniport driver calls the following interfaces, using the correct parameters: • IPortWaveRTStream::AllocatePagesForMdl • IPortWaveRTStream::AllocateContiguousPagesForMdl • IPortWaveRTStream::MapAllocatedPages
PcAllocatedPages	The PcAllocatedPages rule specifies that a PortCls miniport driver frees previous allocated pages by calling AllocatePagesForMdl or AllocateContiguousPagesForMdl methods.
PcIrqlDDIs	The PcIrqlDDIs rule specifies that a PortCls miniport driver must call PortCls DDIs at the correct IRQL level.
PcIrqlIport	The PcIrqlIport rule specifies that a PortCls miniport driver must call PortCls IPort interfaces at the correct IRQL level.

FindBugs

Null pointer deref

```
// Eclipse 3.0,
// org.eclipse.jdt.internal.ui.compare,
// JavaStructureDiffViewer.java, line 131

Control c= getControl();
if (c == null && c.isDisposed())
    return;
```

Unconditional wait

```
// JBoss 4.0.0RC1
// org.jboss.deployment.scanner
// AbstractDeploymentScanner.java, line 185

// If we are not enabled, then wait
if (!enabled) {
  try {
    synchronized (lock) {
      lock.wait();
...
```

David Hovemeyer and William Pugh. 2004. Finding bugs is easy. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (OOPSLA '04). ACM, New York, NY, USA, 132-136.

Some initial Findbugs bug patterns

Code	Description
CN	Cloneable Not Implemented Correctly
DC	Double Checked Locking
DE	Dropped Exception
EC	Suspicious Equals Comparison
Eq	Bad Covariant Definition of Equals
HE	Equal Objects Must Have Equal Hashcodes
IS2	Inconsistent Synchronization
MS	Static Field Modifiable By Untrusted Code
NP	Null Pointer Dereference
NS	Non-Short-Circuit Boolean Operator
OS	Open Stream
RCN	Redundant Comparison to Null
RR	Read Return Should Be Checked
RV	Return Value Should Be Checked
Se	Non-serializable Serializable Class
UR	Uninitialized Read In Constructor
UW	Unconditional Wait
Wa	Wait Not In Loop

Current list of Findbugs bug patterns

BC: Equals method should not assume anything about the type of its argument
BIT: Check for sign of bitwise operation
CN: Class implements Cloneable but does not define or use clone method
CN: clone method does not call super.clone()
CN: Class defines clone() but doesn't implement Cloneable
CNT: Rough value of known constant found
Co: Abstract class defines covariant compareTo() method
Co: compareTo()/compare() incorrectly handles float or double value
Co: compareTo()/compare() returns Integer.MIN_VALUE
Co: Covariant compareTo() method defined
DE: Method might drop exception
DE: Method might ignore exception
DMI: Adding elements of an entry set may fail due to reuse of Entry objects
DMI: Random object created and used only once

http://findbugs.sourceforge.net/bugDescriptions.html

Some challenges in preventing defects

- How do you know what is incorrect behavior?
- How do you explain to a developer the cause of the (potential) defect?
- What happens if the tool approximates program behavior and comes to an incorrect conclusion?

Use of defect prevention tools in OSS projects (Dec 2014)

Source	Projects	Use 1 ASAT	Use > 1 ASATs
GitHub	83	34%	30%
OpenHub	9	67%	22%
SourceForge	10	30%	0%
Gitorious	20	30%	5%
Total	122	36%	23%

TABLE III
DESCRIPTION OF THE ASATS FOR RQ 2 AND 3.

Tool	Language	Format	Extendable	Released	# of Rules
CHECKSTYLE [41]	Java	XML	Yes	2001	179
FINDBUGS [42]	Java	Text	Yes	2003	160
PMD [43]	Java	XML	Yes	2002	330
ESLINT [44]	JavaScript	JSON	Yes	2013	157
JSCS [45]	JavaScript	JSON	Yes	2013	116
JSHINT [46]	JavaScript	JSON	No	2011	253
JSL [47]	JavaScript	Text	No	2005	63
PYLINT [48]	Python	Text	Yes	2006	390
RUBOCOP [49]	Ruby	YAML	Yes	2012	221

M. Beller, R. Bholanath, S. McIntosh and A. Zaidman, "Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software," *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Suita, 2016, pp. 470-481. doi: 10.1109/SANER.2016.105

Why developers don't use defect prevention tools

- Not integrated. The tool is not integrated into the developer’s workflow or takes too long to run
- Not actionable. The warnings are not actionable;
- Not trustworthy. Users do not trust the results due to, say, false positives
- Not manifest in practice. The reported bug is theoretically possible, but the problem does not actually manifest in practice
- Too expensive to fix. Fixing the detected bug is too expensive or risky
- Warnings not understood. Users do not understand the warnings.

Challenges with customizability

- Many tools have many false positives
- Want to have the ability to turn on and off useful and not useful rules
- Teams may customize settings, but then results in issues when different teams use different settings and find different issues with shared code

Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In Proceedings of the 2013 International Conference on Software Engineering (ICSE '13). IEEE Press, Piscataway, NJ, USA, 672-681.

Working with developer intent

- How do you know what behavior is incorrect? (i.e., the oracle problem)
  - Have developers write specifications for a program for properties they care about
  - Build rules about how an API should be used, check that clients use it correctly
  - Look at lots of code, find atypical behaviors

Writing specifications

Model classes should have 'private' fields and getters.
//CompilationUnit[PackageDeclaration/Name[@Image="com.bankapplication.model"]]//ClassOrInterfaceDeclaration[count(ClassOrInterfaceBody/ClassOrInterfaceBodyDeclaration/FieldDeclaration[@Private="true"])=0 or count(ClassOrInterfaceBody/ ClassOrInterfaceBodyDeclaration/MethodDeclaration/MethodDeclarator[starts-with(@Image,"get")])=0]

Natural language spec and corresponding implementation in PMD

- Specifying constraints on code often requires learning and using a new language defined by tool
- Often done by dedicated tool expertise with expertise in writing necessary specs
- May capture company-wide policies

How should potential defects be communicated to developers?

- Static analysis tools increasingly part of the build process
- Builds compile code, run static analysis tools
- Individual teams may build their own static analysis rules

- How should these tools communicate analysis results to developers?

Tricorder

- Goals:
  - Low false positives—error reports should result in code changes
  - Empower users to contribute—let developers write their own checkers
  - Make data-driven usability improvements
  - Effective workflow integration
  - Quick fixes

Analyzer	Description
AffectedTargets	How many targets are affected
AndoidLint	Scans android projects for likely bugs
AutoRefaster	Implementation of Refaster [42]
BuildDeprecation	Identify deprecated build targets
Builder	Checks if a changelist builds
ClangTidy	Bug patterns based on AST matching
DocComments	Errors in javadoc
ErrorProne	Bug patterns based on AST matching
Formatter	Errors in Java format strings
Golint	Style checks for go programs
Govet	Suspicious constructs in go programs
JavacWarnings	Curated set of warnings from javac
JscompilerWarnings	Warnings produced by jscompiler
Linter	Style issues in code
Unused	Unused variable detection
UnusedDeps	Flag unused dependencies

Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: building a program analysis ecosystem. *International Conference on Software Engineering,* 598-608.

Tricorder Analysis Results

```
package com.google.devtools.staticanalysis;

public class Test {
```

Lint — Java — 1:02 AM, Aug 21

Missing a Javadoc comment.

Please fix | Not useful

```
  public boolean foo() {
    return getString() == "foo".toString();
```

ErrorProne — StringEquality — 1:03 AM, Aug 21

String comparison using reference equality instead of value equality (see http://code.google.com/p/error-prone/wiki/StringEquality)

Please fix

Suggested fix attached: show | Not useful

```
  }

  public String getString() {
    return new String("foo");
  }
}
```

Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: building a program analysis ecosystem. *International Conference on Software Engineering,* 598-608.

Communicating errors to developers

- Study at Google based on **26.6 million** builds
- Developers frequently see error messages
  - ~30% of builds fail due to compiler error
- Median resolution time is ~12 minutes

- Dependency errors are the most common

Count	Error	Fix
10	Misspelled identifier	Fix spelling
5	Wrong number of args to constructor call	Add or remove arguments
4	Missing import	Add import
2	Missing dependency	Add dependency to BUILD file
2	Incorrect type parameter in arg to method	Fix type parameter
1	Called a non-existent method	Removed method call
1	Accessed a non-existent field	Added field
1	Removed a class but didn't remove all uses	Removed remaining uses of class

Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. 2014. Programmers' build errors: a case study (at google). In Proceedings of the 36th International Conference on Software Engineering (ICSE 2014). ACM, New York, NY, USA, 724-734. DOI: https://doi.org/10.1145/2568225.2568255

Communicating error messages

```
void m() {
  final int x;
  while (true) {
    x = read();
  }
}
```

```
F.java:5: error: variable x might be assigned in loop
            x = read();
            ^
1 error
```

vs.

```
F.java:5: error: The blank final variable "x" cannot
be assigned within the body of a loop that may execute
more than once.
            x = read();
            ^
```

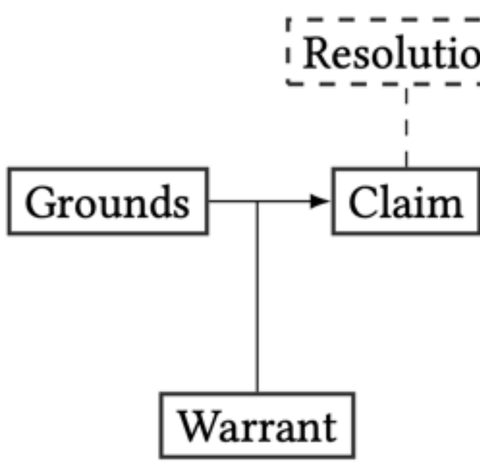
Titus Barik, Justin Smith, Kevin Lubick, Elisabeth Holmes, Jing Feng, Emerson Murphy-Hill, and Chris Parnin. 2017. Do developers read compiler error messages?. In Proceedings of the 39th International Conference on Software Engineering (ICSE '17). IEEE Press, Piscataway, NJ, USA, 575-585. DOI: https://doi.org/10.1109/ICSE.2017.59

Communicating errors

```
F.java:5: error: The blank final variable "x" cannot
be assigned within the body of a loop that may execute
more than once.
            x = read();
            ^
```

Claim: there is a problem

Grounds: why is this a problem



The claim is the concluding assertion or judgment about a problem in the code.

Resolutions suggest concrete actions to the source code to remediate the problem.

Facts, rules, and evidence to support the claim.

Bridging statements that connect the grounds to the claim. Provides justification for using the grounds to support the claim.

Titus Barik, Justin Smith, Kevin Lubick, Elisabeth Holmes, Jing Feng, Emerson Murphy-Hill, and Chris Parnin. 2017. Do developers read compiler error messages?. In Proceedings of the 39th International Conference on Software Engineering (ICSE '17). IEEE Press, Piscataway, NJ, USA, 575-585. DOI: https://doi.org/10.1109/ICSE.2017.59

Examples

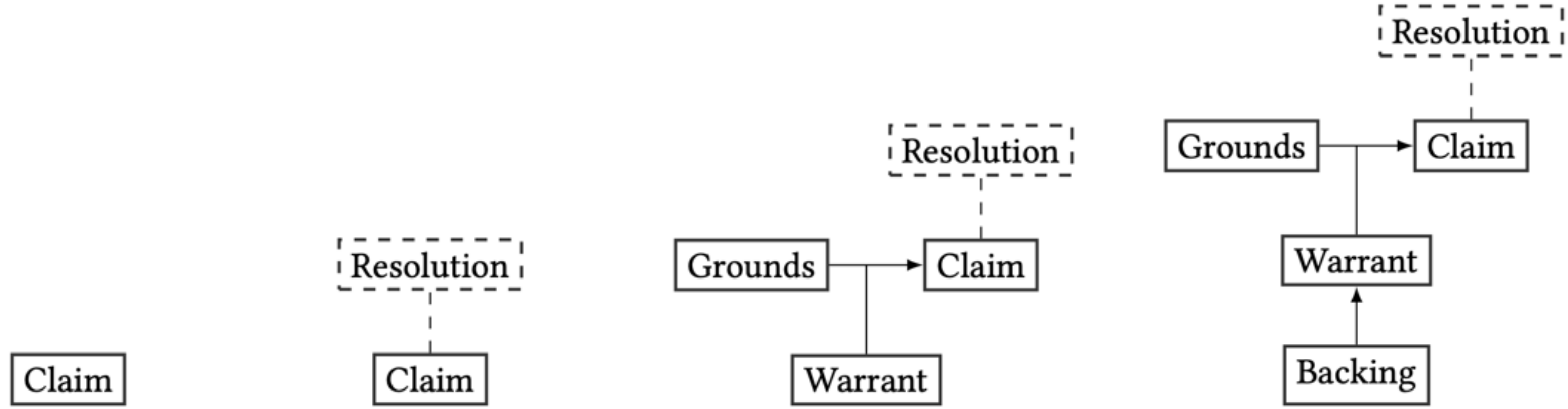
OpenJDK	`cannot find symbol`
	`symbol: variable varnam`
	`location: class Foo`

Jikes	`No field named "varnam" was found in type "Foo". However, there is an accessible field "varname" whose name closely matches the name "varnam".`

- OpenJDK only presents a claim. Jikes presents a ground (**there is an accessible field "varname"**), which is qualified through a rebuttal (However).

Titus Barik, Justin Smith, Kevin Lubick, Elisabeth Holmes, Jing Feng, Emerson Murphy-Hill, and Chris Parnin. 2017. Do developers read compiler error messages?. In Proceedings of the 39th International Conference on Software Engineering (ICSE '17). IEEE Press, Piscataway, NJ, USA, 575-585. DOI: https://doi.org/10.1109/ICSE.2017.59

How do developers themselves explain errors on StackOverflow?



(a) Claim-only (CEM = 191, SO = 0)

(b) Claim-resolution (CEM = 10, SO = 59)

(c) Simple argument layout (CEM = 8, SO = 49)

(d) Extended argument layout (CEM = 1, SO = 102)

Attribute	Description
Simple Argument Components	
CLAIM (Section 5.3.1)	The claim is the concluding assertion or judgment about a problem in the code.
RESOLUTION (Section 5.3.2)	Resolutions suggest concrete actions to the source code to remediate the problem.
GROUNDS (Section 5.3.3)	Facts, rules, and evidence to support the claim.
WARRANT (Section 5.3.4)	Bridging statements that connect the grounds to the claim. Provides justification for using the grounds to support the claim.

Extended Argument Components	
BACKING (Section 5.3.5)	Additional evidence to support the warrant, if the warrant is not accepted.
QUALIFIER (Section 5.3.6)	This is the degree of belief for a claim, often used to weaken a claim.
REBUTTAL (Section 5.3.7)	Exceptions to the claim or other components of the argument.