Preventing Defects

SWE 795, Spring 2017

Software Engineering Environments

GEORGE MASON UNIVERSITY

Today

- Part 1 (Lecture)(~60 mins)

- Part 2 (HW3 Checkpoint Presentations)(20 mins)

- Break!

- Part 3 (Discussion)(~60 mins)
  - Discussion of readings

Preventing 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

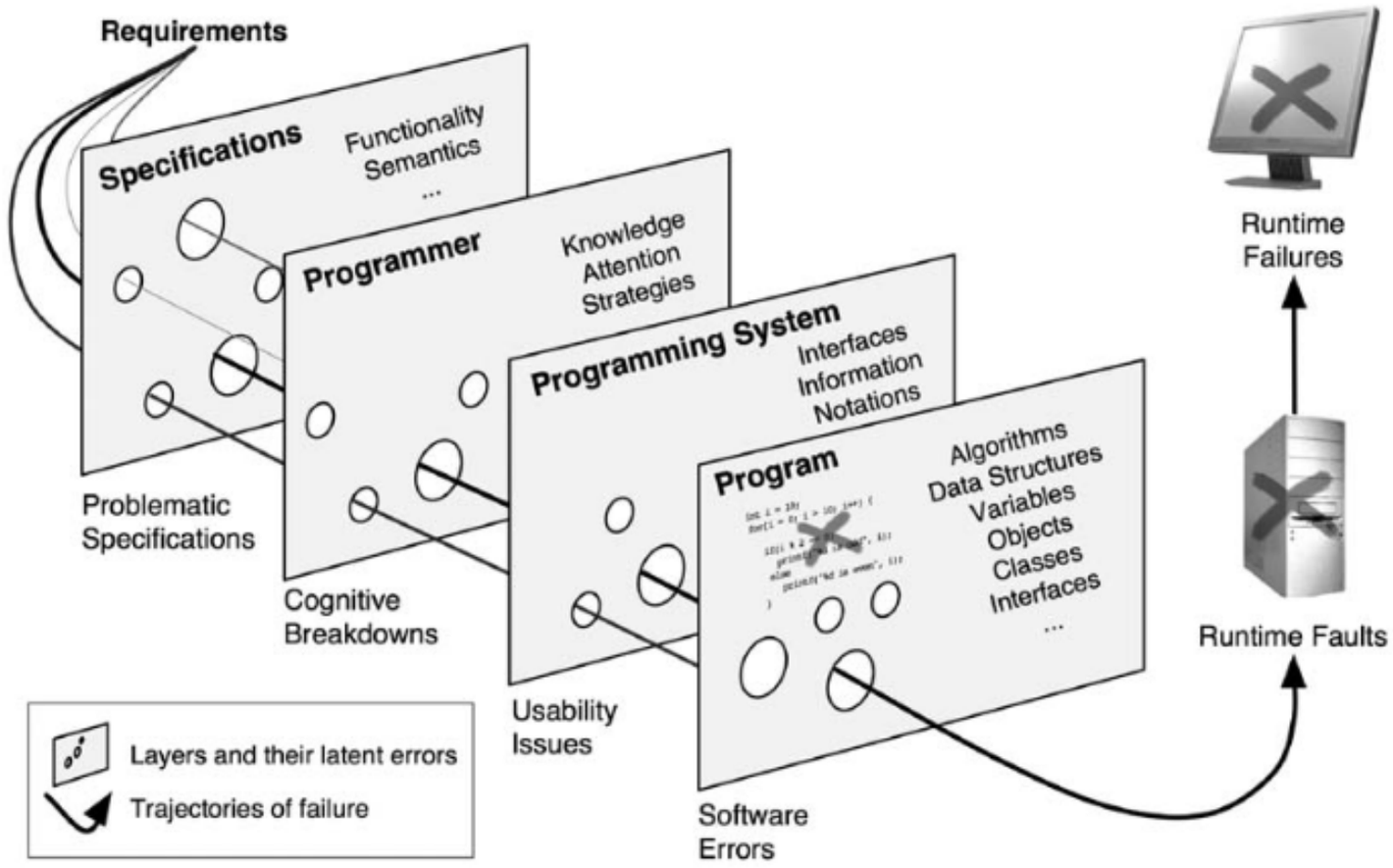
Where do defects come from?

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

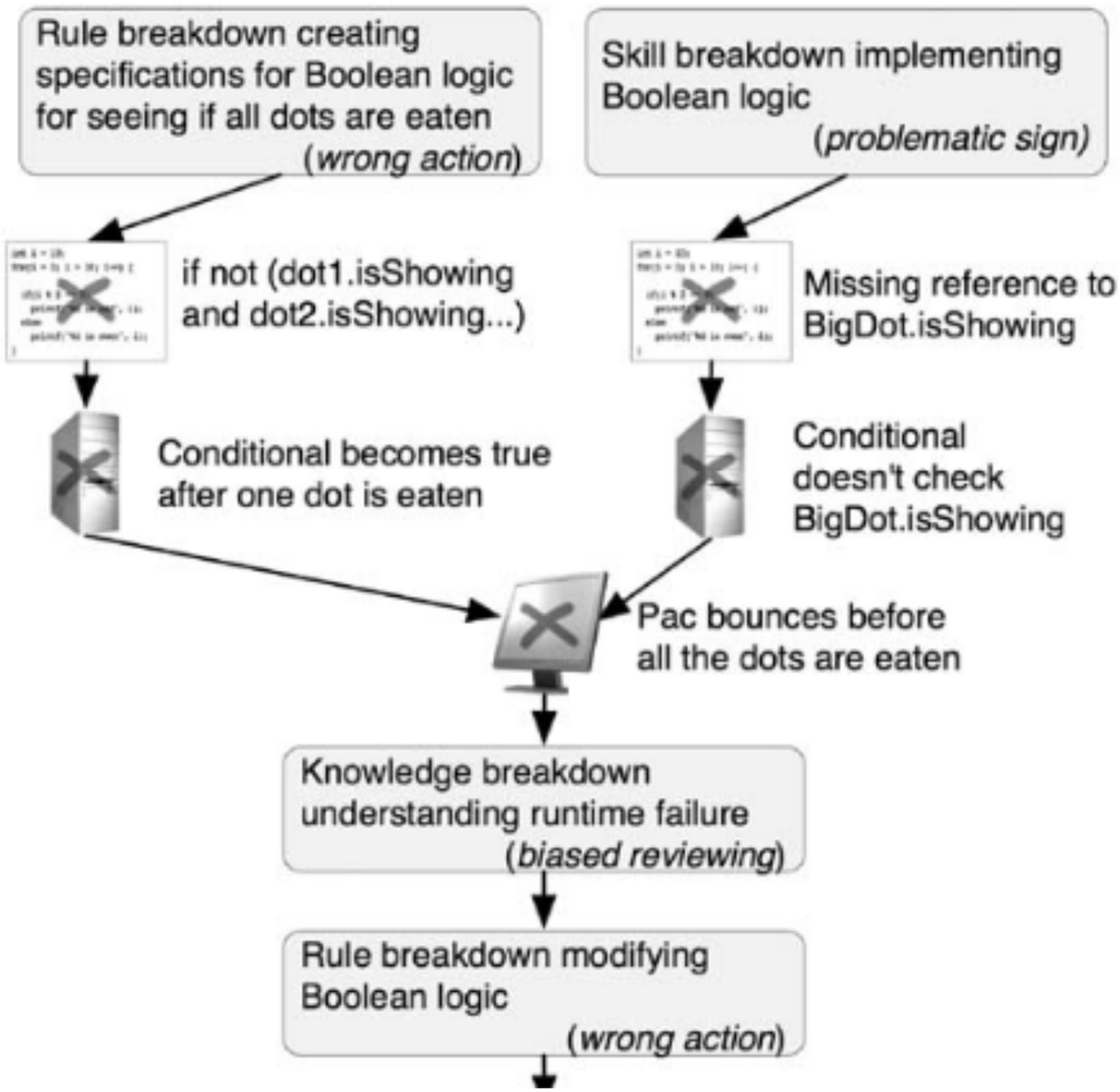
Wrong rule	Type	Events resulting in breakdown
Use of a rule that is successful in most contexts, but not all.	Problematic signs	Ambiguous or hidden signs → conditions evaluated with insufficient info → wrong rule chosen → inappropriate action
	Information overload	Too many signs → important signs missed → 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)

P2 has difficulty creating the specifications for the Boolean logic to check if all of the dots are eaten, as evidenced by verbal utterances; also, part of the expression was obscured, and she though the "BigDot" reference was off-screen.



She only forms one hypothesis about the cause of the failure, which is incorrect.

This causes a breakdown in modifying the Boolean logic.

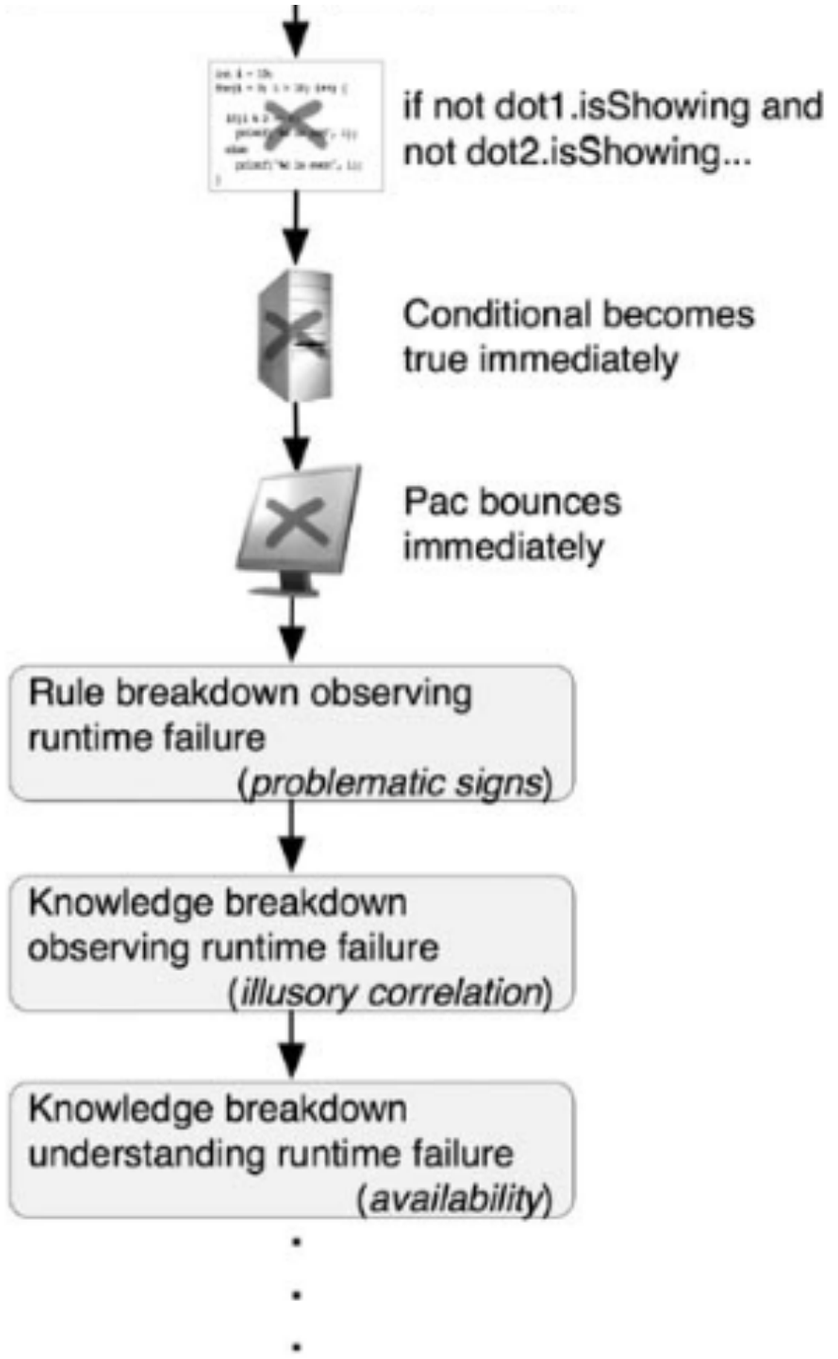
Adapted from Ko & Myers, JVLC05

Breakdown chain example (Part 1)

Because camera was pointing down at Pac, she was unaware that Pac was bouncing.

The fact that Pac doesn't seem to be bouncing leads her to believe he is not.

After 20 minutes, P2 reorients the camera and notices that Pac is bouncing, but assumes it was due to more recent changes and not the earlier error.



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

How can defects be prevented?

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

How can defects be prevented?

- Techniques we'll examine today
  - Specification checkers
  - Idiom detectors
  - Heuristic based detectors

Specification checkers

- Key idea
  - Components (functions, classes, modules, …) have interfaces with rules that describe how they should be correctly used
  - Violations of these rules cause defects
  - Goal: express these rules in a machine-readable form, check that code conforms to these rules, issue compile warnings when it does not

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.

Idiom detectors

- Key idea
  - Code may contain bug patterns that are often (but not always) errors.
  - Provide an open-ended framework for authoring bug pattern detectors
  - Flag instances of bug patterns as warning to developers

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

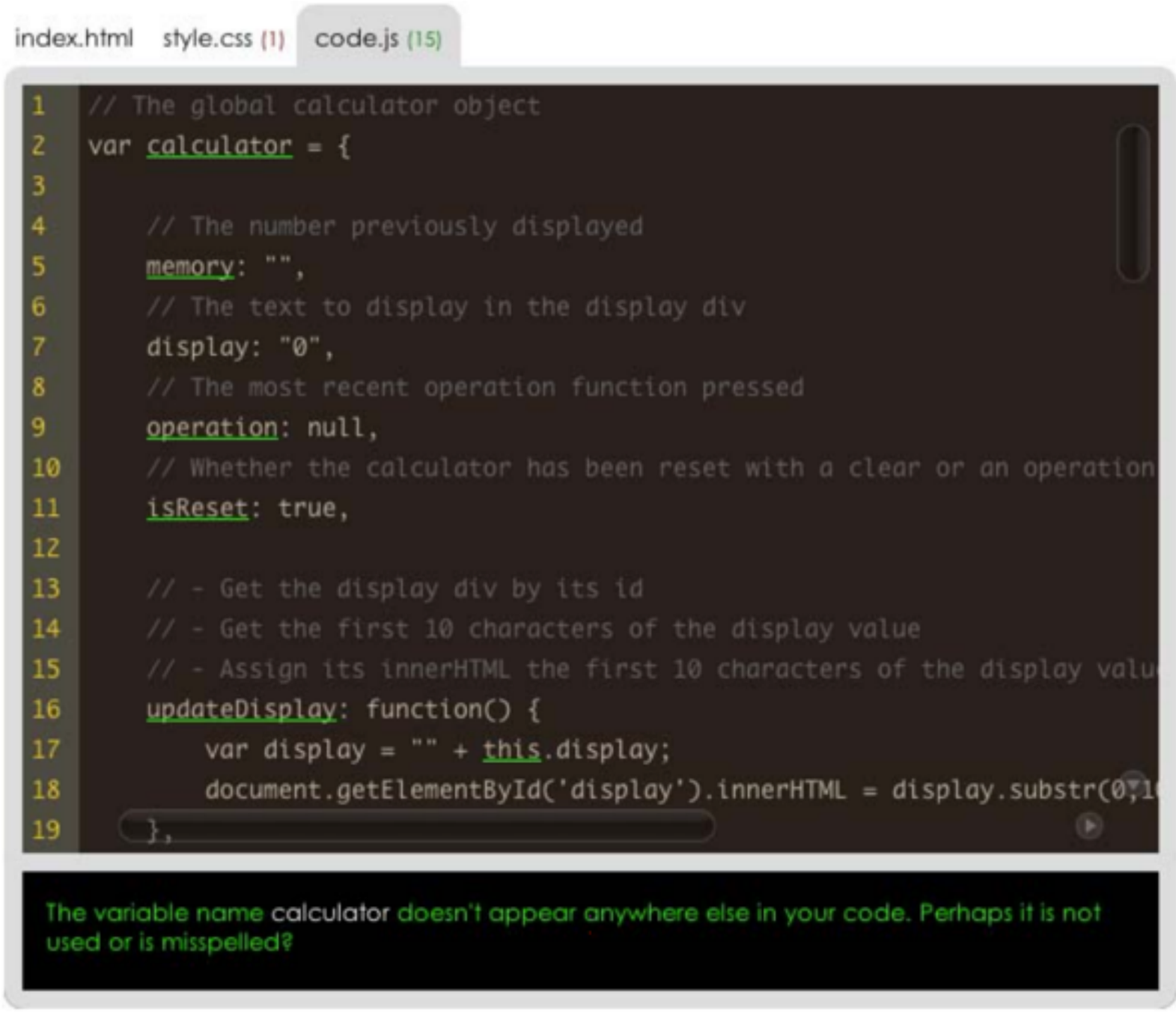
http://findbugs.sourceforge.net/bugDescriptions.html

Heuristic based detectors

- Key idea
  - Use heuristics about how code is usually written to determine what might likely indicate an error
  - Offer developers a warning message

Cleanroom

Uniqueness heuristic: flag any identifier that occurs only once in a software project



A. J. Ko and J. O. Wobbrock, "Cleanroom: Edit-Time Error Detection with the Uniqueness Heuristic," *2010 IEEE Symposium on Visual Languages and Human-Centric Computing*, Leganes, 2010, pp. 7-14.

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 — Missing a Javadoc comment.
Java
1:02 AM, Aug 21

Please fix — Not useful

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

▾ ErrorProne — String comparison using reference equality instead of value equality (see http://code.google.com/p/error-prone/wiki/StringEquality)
StringEquality
1:03 AM, Aug 21

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.