Today

- Part 1 Lecture (~45 mins)
  - 10 min break
- Part 2: Tech Talks (30 mins)
  - Two tech talks
- Part 3: In-Class Activity (1 hour)
Logistics

• HW 4 due 11/29
Overview

• Where do defects come from?

• How can defects be prevented?

• How should potential defects be communicated to developers?
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?

<table>
<thead>
<tr>
<th>Bug Type</th>
<th>Description</th>
<th>Requires Understanding Of</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment bug</td>
<td>Software errors in assigning variables’ values</td>
<td>behavior</td>
</tr>
<tr>
<td>Iteration bug</td>
<td>Software errors in iteration algorithms</td>
<td>language</td>
</tr>
<tr>
<td>Array bug</td>
<td>Software errors in array index expressions</td>
<td>language</td>
</tr>
</tbody>
</table>

**Gould [14]**
Novice Fortran

<table>
<thead>
<tr>
<th>Bug Type</th>
<th>Description</th>
<th>Requires Understanding Of</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visual bug</td>
<td>Grouping related parts of expression</td>
<td></td>
</tr>
<tr>
<td>Naive bug</td>
<td>Iteration instead of parallel processing</td>
<td>‘…need to think step-by-step’</td>
</tr>
<tr>
<td>Logical bug</td>
<td>Omitting or misusing logical connectives</td>
<td></td>
</tr>
<tr>
<td>Dummy bug</td>
<td>Experience with other languages interfering</td>
<td>‘…seem to be syntax oversights’</td>
</tr>
<tr>
<td>Inventive bug</td>
<td>Inventing syntax</td>
<td></td>
</tr>
<tr>
<td>Illiteracy bug</td>
<td>Difficulties with order of operations</td>
<td></td>
</tr>
<tr>
<td>Gestalt bug</td>
<td>Unforeseen side effects of commands</td>
<td>‘…failure to see the whole picture’</td>
</tr>
</tbody>
</table>

**Eisenberg [15]**
Novice APL

Adapted from Ko & Myers, JVLC05
Where do defects come from?

<table>
<thead>
<tr>
<th>Category</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm awry</td>
<td>Improperly implemented algorithms</td>
</tr>
<tr>
<td>Blunder or botch</td>
<td>Accidentally writing code not to specifications</td>
</tr>
<tr>
<td>Data structure</td>
<td>Software errors in using data structures</td>
</tr>
<tr>
<td>Forgotten function</td>
<td>Missing implementation</td>
</tr>
<tr>
<td>Language liability</td>
<td>Misunderstanding language/environment</td>
</tr>
<tr>
<td>Module mismatch</td>
<td>Imperfectly knowing specification</td>
</tr>
<tr>
<td>Robustness</td>
<td>Not handling erroneous input</td>
</tr>
<tr>
<td>Surprise scenario</td>
<td>Unforeseen interactions in program elements</td>
</tr>
<tr>
<td>Trivial typos</td>
<td>Incorrect syntax, reference, etc.</td>
</tr>
</tbody>
</table>

Adapted from Ko & Myers, JVLC05
Where do defects come from?

<table>
<thead>
<tr>
<th>Category</th>
<th>Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eisenstadt [19]</td>
<td>Clobbered memory: Overwriting memory, subscript out of bounds</td>
</tr>
<tr>
<td>Industry experts</td>
<td>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.</td>
</tr>
<tr>
<td>COBOL, Pascal, Fortran, C</td>
<td></td>
</tr>
<tr>
<td>Vendor problems</td>
<td>Buggy compilers, faulty hardware</td>
</tr>
<tr>
<td>Design logic</td>
<td>Unanticipated case, wrong algorithm</td>
</tr>
<tr>
<td>Initialization</td>
<td>Erroneous type or initialization of variables</td>
</tr>
<tr>
<td>Variable</td>
<td>Wrong variable or operator used</td>
</tr>
<tr>
<td>Lexical bugs</td>
<td>Bad parse or ambiguous syntax</td>
</tr>
<tr>
<td>Language</td>
<td>Misunderstandings of language semantics</td>
</tr>
</tbody>
</table>

Adapted from Ko & Myers, JVLC05
Where do defects come from?

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

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

<table>
<thead>
<tr>
<th>Overattention</th>
<th>Type</th>
<th>Events resulting in breakdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attending to routine action causes false assumption about progress of action.</td>
<td>Omission</td>
<td>Attentional check in the middle of routine actions → assumption that actions are already completed → action skipped</td>
</tr>
<tr>
<td></td>
<td>Repetition</td>
<td>Attentional check in the middle of routine actions → assumption that actions are not completed → action repeated</td>
</tr>
</tbody>
</table>

Adapted from Ko & Myers, JVLC05
# Types of rule breakdowns

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

<table>
<thead>
<tr>
<th>Bad rule</th>
<th>Type</th>
<th>Events resulting in breakdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use of a rule with problematic conditions or actions.</td>
<td>Incomplete encoding</td>
<td>Some properties of problem space are not encoded → rule conditions are immature → inappropriate action</td>
</tr>
<tr>
<td></td>
<td>Inaccurate encoding</td>
<td>Properties of problem space encoded inaccurately → rule conditions are inaccurate → inappropriate action</td>
</tr>
<tr>
<td></td>
<td>Exception proves rule</td>
<td>Inexperience → exceptional rule often inappropriate → inappropriate action</td>
</tr>
<tr>
<td></td>
<td>Wrong action</td>
<td>Condition is right but action is wrong → inappropriate action</td>
</tr>
</tbody>
</table>
# Types of knowledge breakdowns

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

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

Rule breakdown creating specifications for Boolean logic for seeing if all dots are eaten
(wrong action)

Skill breakdown implementing Boolean logic
(problematic sign)

if not (dot1.isShowing and dot2.isShowing...)

Conditional becomes true after one dot is eaten

Missing reference to BigDot.isShowing

Conditional doesn't check BigDot.isShowing

Pac bounces before all the dots are eaten

Knowledge breakdown understanding runtime failure
(biased reviewing)

Rule breakdown modifying Boolean logic
(wrong action)

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

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

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
**CALLBACK IDIOMS**

**BIND TARGETS** Identifying or choosing an event, lifecycle hook, or trigger to register a callback

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

**CB2 Constrained Target:**
bind target code fragment → API rules making fragment (in)valid

**CB3 Confused Target:**
current & desired bind targets → API use differences, new target’s code fragment

**CALLBACK CONTEXTS** Identifying when the callback is dispatched, using its arguments, or other related objects

**CB4 Improper Scheduling:**
callback code fragments & desired schedule → correct callback order & code fix

**CB5 Unidentified State:**
desired state → API rationale for identifying state & code fragment to obtain it

**CB6 Missed Callbacks:**
callback code fragment → API rationale & state required for callback to occur

**BIND CONFIGURATIONS** Setting options of a callback trigger, or modifying parameters of its bind mechanism

**CB7 Incorrect Bind Parameters:**
callback parameter fragments & desired behavior → correct code fragments

**CB8 Misconfigured Framework:**
framework configuration fragments & desired behavior → correct framework code
Callbacks

`x.on("event", ..., function callback(arg){/***/})`

bind target: event to subscribe to
bind configuration: parameters controlling behavior

callback context: what args and additional state is available when invoked
Common problems with callbacks

Bind Targets: Identifying or choosing an event, life cycle hook, or trigger to register a callback

• CB1 **Unidentified** Target
  desired bind target → API name & code fragment

• CB2 **Constrained** Target
  bind target code fragment → API rules making fragment (in)valid

• CB3 **Confused** Target
  current & desired bind targets → API use differences, new target’s code fragment
**GRAPHICAL IDIOMS**

**GRAPHICAL SETTERS** UPDATING GRAPHICAL PROPERTIES OF THE LAYOUT VIA API (DOM ACCESS METHODS, CSS SELECTORS)

- **GB1 Unidentified Setter:**
  visual property change $\rightarrow$ code fragment to mutate property

- **GB2 Unobservable Setter:**
  setterA & visual property change $\rightarrow$ setterB to mutate property

- **GB3 Indirect Setter:**
  setterA $\rightarrow$ elements which inherit properties from setterA or occlude mutations

- **GB4 Overwritten Setter:**
  setterA $\rightarrow$ setterB overwriting setterA & code fragments with alternative fixes

**GRAPHICAL QUERIES** RETRIEVING GRAPHICAL ELEMENTS OR SIMILAR REPRESENTATIONS VIA API (DOM ACCESS METHODS, CSS SELECTORS)

- **GB5 Incomplete Query:**
  queryA and desired elements to be matched $\rightarrow$ queryB matching those elements

- **GB6 Outdated Query:**
  queryA $\rightarrow$ changes to query result set over time & code fragment fixing it

- **GB7 Overwritten Query:**
  queryA $\rightarrow$ queryB intersecting queryA’s mutations & code fragment fixing queryA

**GRAPHICAL GETTERS** OBTAINING GRAPHICAL PROPERTIES OF THE LAYOUT VIA API METHODS

- **GB8 Unidentified Getter:**
  visual property $\rightarrow$ getter code fragment to retrieve it
Graphical idioms

const \([r1, r2] = \) queryInterface(params);

\(r1.\text{get}("prop") \&\& r2.\text{set}({aProp: value, ...})\);

graphical query

graphical getter

graphical setter
Common problems with graphical idioms

Graphical Setters: Updating graphical properties of the layout via API (DOM access methods, CSS selectors)

- GB1 **Unidentified** Setter
  visual property change → code fragment to mutate property

- GB2 **Unobservable** Setter
  setterA & visual property change → setterB to mutate property

- GB3 **Indirect** Setter
  setterA → elements which inherit properties from setterA or occlude mutations

- GB4 **Overwritten** Setter
  setterA → setterB overwriting setterA & code fragments with alternative fixes
**Valid References**
Determining defined standard, or framework identifiers at compile time or runtime

- **OB1 Inactionable Reference Error:**
  statement generating error & error message → explanation of error message
- **OB2 Silent Invalid Reference:**
  invalid statement → warning message & statement fixing warning

**Back-End Requests**
Sending structured data to a server, or handling server responses

- **OB6 Misconfigured Request:**
  back-end request & desired behavior → modified request matching behavior
- **OB7 Unclear Transmission:**
  back-end request as sent → back-end request as received
- **OB8 Mishandled Response:**
  back-end request → code fragment for response(s) listening and parsing

**Scope Contexts**
Identifying the context given to the keyword `this` within a code block, or a variable’s visibility

- **OB12 Unclear Scope:**
  `this` statement → owner scope of `this`

**Collections and Formats**
Creating or manipulating a collection, or formatting data for use in a framework or library

- **OB3 Unidentified Iteration Construct:**
  collection object → code fragment with corresponding iteration construct
- **OB4 Occluded Modification:**
  collection object & loop fragment → modifications of collection per iteration
- **OB5 Confused Formatting:**
  object in format A → code fragment converting object to format B

**Method Chains**
Determining the effects of a method invocation within a sequence of consecutive calls

- **OB9 Incomplete Sequence:**
  `o.m1(...)..m2(...)..mn(...)` → `o.m1(...)..m2(...)..mk(...)..mn(...)`
- **OB10 Incorrect Sequence:**
  `o.m1(...)..m2(...)..mn(...)` → `o.mk(...)..m1(...)..mn(...)`
- **OB11 Overwritten Effect:**
  `o.m1(...)..m2(...)..mn(...)` → methods `mk` and `ml` where both mutate object
Common problems w/ object-interaction idioms

Valid References: Determining defined standard or framework identifiers at compile time or runtime

• OB1 **Inactionable** Reference Error
  statement generating error & error message \(\rightarrow\) explanation of error message

• OB2 **Silent** Invalid Reference
  invalid statement \(\rightarrow\) warning message & statement fixing warning
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

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

Static Driver Verifier

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

<table>
<thead>
<tr>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PcAddAdapterDevice</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>PcAllocateAndMapPages</strong></td>
<td>The PcAllocateAndMapPages rule specifies that a PortCls miniport driver calls the following interfaces, using the correct parameters:</td>
</tr>
<tr>
<td></td>
<td>• IPortWaveRTStream:AllocatePagesForMdl</td>
</tr>
<tr>
<td></td>
<td>• IPortWaveRTStream:AllocateContiguousPagesForMdl</td>
</tr>
<tr>
<td></td>
<td>• IPortWaveRTStream:MapAllocatedPages</td>
</tr>
<tr>
<td><strong>PcAllocatedPages</strong></td>
<td>The PcAllocatedPages rule specifies that a PortCls miniport driver frees previous allocated pages by calling AllocatePagesForMdl or AllocateContiguousPagesForMdl methods.</td>
</tr>
<tr>
<td><strong>PcIrqlDDIs</strong></td>
<td>The PcIrqlDDIs rule specifies that a PortCls miniport driver must call PortCls DDIs at the correct IRQL level.</td>
</tr>
<tr>
<td><strong>PcIrqlIport</strong></td>
<td>The PcIrqlIport rule specifies that a PortCls miniport driver must call PortCls IPort interfaces at the correct IRQL level.</td>
</tr>
</tbody>
</table>
FindBugs

### Some initial Findbugs bug patterns

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

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)

<table>
<thead>
<tr>
<th>Source</th>
<th>Projects</th>
<th>Use 1 ASAT</th>
<th>Use &gt; 1 ASATs</th>
</tr>
</thead>
<tbody>
<tr>
<td>GitHub</td>
<td>83</td>
<td>34%</td>
<td>30%</td>
</tr>
<tr>
<td>OpenHub</td>
<td>9</td>
<td>67%</td>
<td>22%</td>
</tr>
<tr>
<td>SourceForge</td>
<td>10</td>
<td>30%</td>
<td>0%</td>
</tr>
<tr>
<td>Gitorious</td>
<td>20</td>
<td>30%</td>
<td>5%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>122</strong></td>
<td><strong>36%</strong></td>
<td><strong>23%</strong></td>
</tr>
</tbody>
</table>

### TABLE III

**DESCRIPTION OF THE ASATs FOR RQ 2 AND 3.**

<table>
<thead>
<tr>
<th>Tool</th>
<th>Language</th>
<th>Format</th>
<th>Extendable</th>
<th>Released</th>
<th># of Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHECKSTYLE [41]</td>
<td>Java</td>
<td>XML</td>
<td>Yes</td>
<td>2001</td>
<td>179</td>
</tr>
<tr>
<td>FindBugs [42]</td>
<td>Java</td>
<td>Text</td>
<td>Yes</td>
<td>2003</td>
<td>160</td>
</tr>
<tr>
<td>ESLINT [44]</td>
<td>JavaScript</td>
<td>JSON</td>
<td>Yes</td>
<td>2013</td>
<td>157</td>
</tr>
<tr>
<td>JSCS [45]</td>
<td>JavaScript</td>
<td>JSON</td>
<td>Yes</td>
<td>2013</td>
<td>116</td>
</tr>
<tr>
<td>JSHINT [46]</td>
<td>JavaScript</td>
<td>JSON</td>
<td>No</td>
<td>2011</td>
<td>253</td>
</tr>
<tr>
<td>JSL [47]</td>
<td>JavaScript</td>
<td>Text</td>
<td>No</td>
<td>2005</td>
<td>63</td>
</tr>
<tr>
<td>PYLINT [48]</td>
<td>Python</td>
<td>Text</td>
<td>Yes</td>
<td>2006</td>
<td>390</td>
</tr>
<tr>
<td>RuboCop [49]</td>
<td>Ruby</td>
<td>YAML</td>
<td>Yes</td>
<td>2012</td>
<td>221</td>
</tr>
</tbody>
</table>

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
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.

```java
//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

Tricorder Analysis Results

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

public class Test {

    // Lint
    // Java
    // 1:02 AM, Aug 21
    // Please fix
    // Missing a Javadoc comment.

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

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

}

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

### sjmaple/goof
- **package.json**: 12 H 13 M 8 L
  - View report and fix: Test daily
  - Tested an hour ago

### sjmaple/java-goof
- **pom.xml**: 0 H 0 M 0 L
  - View report: Test daily
- **todoist-core/pom.xml**: 1 H 3 M 0 L
  - View report and fix: Test daily
- **todoist-web-common/pom.xml**: 3 H 4 M 0 L
  - View report and fix: Test daily
- **todoist-web.struts/pom.xml**: 17 H 22 M 2 L
  - View report and fix: Test daily

### sjmaple/jdk9-jigsaw
- **session-3-jshell/jShell-Examples/pom.xml**: 0 H 0 M 0 L
  - View report: Test daily
- **session-3-jshell/shellexFX/pom.xml**: 0 H 0 M 0 L
  - View report: Test daily
- **session-3-jshell/teamjshell/pom.xml**: 7 H 9 M 0 L
  - View report and fix: Test daily

### sjmaple/shallow-goof
- **package.json**: 2 H 2 M 0 L
  - View report and fix: Test daily
  - Tested 28 minutes ago

### sjmaple/spring-goof
- **pom.xml**: 6 H 5 M 0 L
  - View report and fix: Test daily
  - Tested an hour ago
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

<table>
<thead>
<tr>
<th>Count</th>
<th>Error</th>
<th>Fix</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Misspelled identifier</td>
<td>Fix spelling</td>
</tr>
<tr>
<td>5</td>
<td>Wrong number of args to constructor call</td>
<td>Add or remove arguments</td>
</tr>
<tr>
<td>4</td>
<td>Missing import</td>
<td>Add import</td>
</tr>
<tr>
<td>2</td>
<td>Missing dependency</td>
<td>Add dependency to BUILD file</td>
</tr>
<tr>
<td>2</td>
<td>Incorrect type parameter in arg to method</td>
<td>Fix type parameter</td>
</tr>
<tr>
<td>1</td>
<td>Called a non-existent method</td>
<td>Removed method call</td>
</tr>
<tr>
<td>1</td>
<td>Accessed a non-existent field</td>
<td>Added field</td>
</tr>
<tr>
<td>1</td>
<td>Removed a class but didn’t remove all uses</td>
<td>Removed remaining uses of class</td>
</tr>
</tbody>
</table>
Communicating error messages

```java
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();
    ^

F.java:5: error: The blank final variable "x" cannot be assigned within the body of a loop that may execute more than once.

\[ x = \text{read}(); \]

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.

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).
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)

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Claim</strong></td>
<td>The claim is the concluding assertion or judgment about a problem in the code.</td>
</tr>
<tr>
<td>(Section 5.3.1)</td>
<td></td>
</tr>
<tr>
<td><strong>Resolution</strong></td>
<td>Resolutions suggest concrete actions to the source code to remediate the problem.</td>
</tr>
<tr>
<td>(Section 5.3.2)</td>
<td></td>
</tr>
<tr>
<td><strong>Grounds</strong></td>
<td>Facts, rules, and evidence to support the claim.</td>
</tr>
<tr>
<td>(Section 5.3.3)</td>
<td></td>
</tr>
<tr>
<td><strong>Warrant</strong></td>
<td>Bridging statements that connect the grounds to the claim. Provides justifica-</td>
</tr>
<tr>
<td>(Section 5.3.4)</td>
<td>tion for using the grounds to support the claim.</td>
</tr>
</tbody>
</table>

**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.
10 min break
Tech Talks
In-Class Activity

• In groups of 2 or 3, try out Snyk, eslint (JavaScript), or SpotBugs (Java)
• Find a codebase that you can run it on (e.g., your 695 project)
• Based on the language, choose an appropriate tool
• Download and setup the tool, run it on your codebase
• Try to make a really small change to your codebase (or intentionally write incorrect code), check for defects
• Write a reflection on your experiences using the tool:
  • Which tool did you pick
  • Review of the installation and setup process. What steps were required to get it to work with your project?
  • Did it find any defects in your project? If so, what were they? Were they defects you think are worth fixing?
  • What’s hardest to use about the tool? What information would you like to see that it doesn’t currently provide?

• Submission
  • Submit a pdf with your reflection through Blackboard. 1 submission per group. Due 7:10pm today.