Turning Eclipse Against Itself:
Finding Errors in Eclipse Sources

Benjamin Livshits
Stanford University
Summary

• We want to **find errors in Eclipse**
  – Implemented a tool called **Checklipse** (plugin)
  – Uses **lightweight static analysis**
• Looks for violations of **3 design rules**
  – One **API usage** rule
  – Two **resource management** rules
• Preliminary results are **encouraging**
  – Ran Checklipse on Eclipse sources
  – Found a total of **68** likely errors
  – Checklipse checks multiple plugins in **minutes**
Eclipse Code Base...

- **Eclipse:**
  - One of the biggest Java projects ever written
  - Very robust
- **Still, a multitude of bugs exist**
  - bugs.eclipse.org – hundreds of known errors
  - Think about *unknown* ones!
  - Certain types of errors are repeatedly introduced over and over
    - "Lapsed listener" errors discovered for different Eclipse releases
Error Patterns to the Rescue

• Lots of API-specific coding patterns
  – Patterns are “specified”
    • using comments
    • not at all
  – Misuse of these patterns leads to errors

• This is great news for us!
  – Discover what the error patterns are
  – Find and report *pattern violations*
    • Can do so using *dynamic* or *static* analysis

• On to the error patterns...
  – 3 patterns evaluated
  – Many more remain – looking to expand the scope of Checkclipse
Error pattern #1: Call super

- A common rule of thumb in Eclipse code
  - For many methods $m$
  - A subclass implementing method $m$ must call `super.m(...)` inside method $m$

Must call `super`, but don’t
Error pattern #2: Failing to Dispose

- OS resource leaks are common:
  - Many classes define method \textit{dispose()}
  - SWT design rule: dispose what you create
    - Interesting special case: maps
    - Need to clear most class-allocated maps in \textit{dispose()}
    - Failing to clear the maps, causes OS resource leaks
Error pattern #3: Lapsed Listeners

- Memory leaks exist in Java, despite GC!
- Common case of memory leaks:
  - Listeners are used to register handlers for events, such as mouse clicks, etc.
  - Not un-registering listeners properly leads to memory leaks
  - Memory leaks lead to crashes and instability
Checking for Pattern Violations: How?

- **Runtime or dynamic approaches**
  - Aspects allow run time checking of rules
  - Memory profilers and debuggers
  - Custom-made tools:
    - *sleak* by Steve Northover, the architect of SWT
    - a tool to check for memory leaks in Eclipse code
  - But: violations need to be triggered during a particular execution!

- **Instead, we analyze the Java source code of the plugins**

- **Advantages:**
  - No need to consider a particular execution
  - So, can find all potential pattern violations
Static Analysis State of the Art

- Sound and complete analysis approaches
  - Suffer from imprecision – false positives
  - Don’t scale to code bases the size of Eclipse

- We use unsound lightweight static analysis
  - Runs fast – took several minutes to analyze
    20 core Eclipse plugins
  - Produces false positives
    - Takes time to filter the false positives
  - May miss errors
Why Lightweight Static Checking?

- **Overall goal:**
  - Address whole *classes* of problems
  - Better target debugging efforts of Eclipse developers

- **Make it fast and easy to audit potential errors**
  - User is presented with three custom viewers
  - One for each error pattern
    - Extend super viewer
    - Dispose rule viewer
    - Lapsed listener viewer

- **Can run analysis and fix the errors without ever leaving Eclipse**
  - So, what did we find?
We Find...

<table>
<thead>
<tr>
<th>Extend Super</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>methods that require super to be called</td>
<td>38</td>
</tr>
<tr>
<td>calls to these methods</td>
<td>390</td>
</tr>
<tr>
<td>filtered calls</td>
<td>19</td>
</tr>
<tr>
<td>potential errors (methods not calling super)</td>
<td>13</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Disposal Rules</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>dispose methods checked</td>
<td>794</td>
</tr>
<tr>
<td>filtered methods</td>
<td>51</td>
</tr>
<tr>
<td>potential errors (leaking dispose methods)</td>
<td>42</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lapsed Listeners</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>subclasses of ViewPart checked</td>
<td>81</td>
</tr>
<tr>
<td>subclasses with matched listeners</td>
<td>6</td>
</tr>
<tr>
<td>subclasses not using listeners</td>
<td>53</td>
</tr>
<tr>
<td>subclasses with mismatched listeners</td>
<td>22</td>
</tr>
<tr>
<td>potential errors (classes with lapsed listeners)</td>
<td>13</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Total Errors</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>68</td>
</tr>
</tbody>
</table>
Status

Implemented Checklipse
- Working tool
- Runs fast
- Available for download
  suif.stanford.edu

Find 68 *likely* errors
- Many are hard to evaluate
- Used our best judgement to determining what is an error
- Need a strong knowledge of APIs
Future Work

Happy to pass the errors over to IBM engineers

Have a project to find and correct similar error patterns dynamically

Looking for new patterns to check