Exposing unforeseen consequences of software change

David Notkin
University of Washington
February 2010

Joint work with Reid Holmes
Thank you!

• My first trip to India … and I am sure not my last!
• Wonderful opportunity to talk to old friends and to meet new ones
• The ISEC organizers have been wonderful – so it is no surprise that the program is strong, which is notable for a third instance of a conference
• (My two favorite signs on this campus)
• An opportunity to present some fairly new work
Today’s objective

• Characterize some difficulties in achieving desired behaviors in software, especially in the face of change
• Distinguish, informally, between expected and unexpected changes
• Describe a simple technique based on partitioning static and dynamic call relations over a pair of versions
• Present some qualitative and quantitative assessments that suggest that some of the partitions represent small sets of unexpected changes
Today’s non-objectives include

• Showing that this technique finds more bugs more quickly and more precisely
• Showing external or in-the-field use of this technique
• Arguing that this technique is better than every other even vaguely related technique in the literature
  – At best, the technique is complementary to other approaches
A Letter from Dijkstra, 1968

A programmer manipulates source code to accomplish a desired effect in the resulting program behaviors.

The behaviors are the real focus – the program is the indirect vehicle for manipulating them.
Behaviors are daunting

- **Change-independent**
  - There are an unbounded number of behaviors associated with a single program

- **Change-related**
  - Local changes to a program can cause unexpected behaviors in non-local parts of the program
  - Unexpected behaviors may not cause faults or failures – and problems that do arise often only surface in later versions, or with new tests
New post-change behaviors

Expected

Unexpected

Bugs

Not bugs

The reasons for the change

May be caught by testing, analysis, etc.

Harder to catch, easier to cause problems later on
Unexpected

Bugs

Not bugs

Rajmani: an issue of observability? Maybe
Approaches to managing behaviors
A very incomplete list

- Structured control constructs – constrain behaviors
- Sound analyses – guarantee a property over all possible behaviors
- Regression testing – given a source change, ensure that no old tested behaviors fail
- Impact analysis – given a source change, identify other locations in the program that might see new behaviors
- Refactoring – provide guarantee about the semantics of new behaviors
- Visualization – allow people to see new behaviors
- …others…
Further confounded by degradation [Belady & Lehman, …]

The Beauty  The Beast
Approaches to reducing degradation
A very incomplete list

- Refactoring – small dip in complexity (maybe not for others on the project), small cost
- Reengineering – major change in complexity, major costs
- General anti-regressive work … [Lehman 1974]
Degradation confounds behaviors

- “Spaghetti” code is harder to associate with behaviors
- Changes are more likely to be inadvertently non-local
- Identifying faults and failures may be more difficult
- …

- So, not only does the program structure itself degrade, but the relationship between the program --- the static-dynamic relation --- degrades
Assumption 1

- Programmers are rarely surprised by the actual changes they make to the code – after all, they are making the changes with specific intent.
Assumption 2

- Despite Assumption 1, programmers are worried that there will be unexpected consequences to their changes [Ko, Deline, Venolia 2007]

Developers had three primary questions when exposing their code to their teammates:

\[ (s1) \text{ Did I make any mistakes in my new code?} \\
(s2) \text{ Did I follow my team’s conventions?} \\
(s3) \text{ Which changes are part of this submission?} \]
Assumption 3
Degradation takes place a little bit at a time

vs.
Program Change

Did it get worse?

Explicit change

Implicitly understood

Initial Behaviors

Modified Behaviors
That is...we assume

- The initial program is a baseline
- The initial static-dynamic relation is a baseline
- The modifications to the initial program are a baseline
- Other behaviors are unexpected

- Roughly
  - Figure out all the static and dynamic relations in both versions
  - Filter out the baseline information as expected
  - Report the rest
Why focus on small changes?

One day at a time.
• How does a project get late? How does a journal paper get delayed? …

One tuple at a time.
• How does the relationship between the source and the behaviors degrade?

• The question, “what changed?” may be much simpler to comprehend than the question, “what is true?”
• Report degradation of the relationship incrementally, as it happens
• Use explicit changes and previous behaviors to filter out expected behaviors
Scenario

<table>
<thead>
<tr>
<th>Static Analysis</th>
<th>Dynamic Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>V1</td>
</tr>
<tr>
<td>V2</td>
<td>s</td>
</tr>
<tr>
<td>~V2</td>
<td>s+</td>
</tr>
<tr>
<td>~V2</td>
<td>s-</td>
</tr>
<tr>
<td>V2</td>
<td>d</td>
</tr>
<tr>
<td>~V2</td>
<td>d+</td>
</tr>
<tr>
<td>~V2</td>
<td>d-</td>
</tr>
</tbody>
</table>
Simple example

```java
private void genStore() {
    ...
    cache()
}

private void cache() {
    LocalType l = new LocalType();
    _collection.add(l);
}
```
s+: { genStore() → cache(),
       cache() → LocalType()
       cache() → Collection.add(Object) }
Dynamic

d+: {  genStore() → cache(),
            cache() → LocalType()
            cache() → Collection.add()
            Collection.add(...) → LocalType.equals(...) }
Partition

\[
\begin{align*}
\text{genStore()} & \rightarrow \text{cache()} \\
\text{cache()} & \rightarrow \text{LocalType()} \\
\text{cache()} & \rightarrow \text{Collection.add(\ldots)}
\end{align*}
\]

\[
\text{Collection.add(\ldots)} \rightarrow \text{LocalType.equals(\ldots)}
\]
Unchanged dependencies

- Probably expected
Consistent change

- Probably expected
Unlikely

- Could happen but…
Evaluation

• Analyzed 10 consecutive versions of three different systems
  – Simulated a continuous build system
• Qualitative and quantitative questions include
  – Do the partitions arise in practice?
  – Are the partitions of interest small enough?
  – Are the surfaced facts interesting?

<table>
<thead>
<tr>
<th>Project</th>
<th>KLOC</th>
<th>#Tests</th>
<th>Last Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visualization</td>
<td>17</td>
<td>365</td>
<td>30</td>
</tr>
<tr>
<td>JodaTime</td>
<td>76</td>
<td>2525</td>
<td>1396</td>
</tr>
<tr>
<td>RFC 2445</td>
<td>7</td>
<td>171</td>
<td>22</td>
</tr>
</tbody>
</table>
Qualitative example: Visualizer v22→v23

• Callback disappears (d-)
  – Callback originally happened within an anonymous class

```java
com.google.common.collect.Ordering.givenOrder(List) →
AggregationColumn.equals(Object)
```
Qualitative example: JodaTime v1366⇒v1367

- Inconsistency between \texttt{s+} and \texttt{s+d+}
  - A thrown exception prevents an added method call

\begin{verbatim}
TestDuration_Basics.testToStandardSeconds_overflow()
    → Assert.fail()
\end{verbatim}

- Three unexpected method calls (\texttt{d+})
  - New tests executed by JUnit

\begin{verbatim}
unknown → TestDuration_Basics.testGetStandardSeconds()
unknown → TestDuration_Basics.testToStandardSeconds()
unknown → TestDuration_Basics.testToStandardSeconds_overflow()
\end{verbatim}
Qualitative example – comparing libraries: JodaTime with JDK5 and JDK6

– One new method call (\textcolor{red}{sd+})
  
  • [via reflection to new API]

\begin{verbatim}
DateTimeUtils.getDateFormatSymbols(Locale) \rightarrow Method.invoke(Object, Object[])
\end{verbatim}

– One old method call disappeared (\textcolor{red}{sd-})

\begin{verbatim}
DateTimeUtils.getDateFormatSymbols(Locale) \rightarrow DateFormatSymbols(Locale)
\end{verbatim}
Conciseness

- Total # of facts in each partition across all 30 program versions evaluated
- 99+% reduction in interesting partitions

![Venn Diagram]

- \( s+d+ \): 122
- \( s-d+ \): 176
- \( s \): 31
- \( 399K \)
- \( d+ \): 58
- \( d- \): 15
- \( d \): 68K
- \( 284K \)
- \( 0 \)
- \( 0 \)
- \( 11 \)
- \( 0 \)
- \( 15 \)
- \( 0 \)
<table>
<thead>
<tr>
<th>Visualizer</th>
<th>Uninteresting</th>
<th>Consistent</th>
<th>Inconsistent</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unchanged</td>
<td>Statically Removed</td>
<td>Statically Added</td>
</tr>
<tr>
<td></td>
<td>s/sd/d</td>
<td>s'</td>
<td>s'd'</td>
</tr>
<tr>
<td>17→19</td>
<td>9,027</td>
<td>19</td>
<td>9</td>
</tr>
<tr>
<td>19→20</td>
<td>9,061</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20→21</td>
<td>9,063</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>22→23</td>
<td>9,062</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23→24</td>
<td>9,073</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>24→28</td>
<td>9,074</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>28→29</td>
<td>9,070</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>29→30</td>
<td>9,093</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RFC-2445</td>
<td></td>
<td>2,582</td>
<td>4</td>
</tr>
<tr>
<td>9→12</td>
<td>2,582</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>12→13</td>
<td>2,587</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>13→15</td>
<td>2,587</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>15→16</td>
<td>2,641</td>
<td>14</td>
<td>10</td>
</tr>
<tr>
<td>17→18</td>
<td>2,618</td>
<td>2</td>
<td>23</td>
</tr>
<tr>
<td>18→20</td>
<td>2,642</td>
<td>2</td>
<td>23</td>
</tr>
<tr>
<td>20→21</td>
<td>2,648</td>
<td>2</td>
<td>23</td>
</tr>
<tr>
<td>JodaTime</td>
<td></td>
<td>1366→1367</td>
<td>71,795</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1367→1374</td>
<td>71,812</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1374→1378</td>
<td>71,816</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1378→1379</td>
<td>71,819</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1379→1380</td>
<td>71,814</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1380→1381</td>
<td>71,827</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1381→1388</td>
<td>71,832</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1388→1389</td>
<td>71,831</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1389→1396</td>
<td>71,832</td>
</tr>
</tbody>
</table>
Help developer focus

- Most changes are not interesting or surprising
- By identifying inconsistent changes when they happen, the developer can apply their judgment immediately – reducing, we hope, latent, hard-to-detect errors
Related work: visualization

• De Pauw et al. [OOPSLA 93] and Baecker [ICSE 88] use visualization techniques to relate the static and dynamic nature of a given program – but not on a pair of related programs
Related work: unintended change

- The idea that fixing errors is itself an erroneous process — often called imperfect debugging — has been modeled as part of software reliability engineering since the mid-1970’s [Miyamoto 75]
- Risks Digest documents thousands of computer-related risks to the public; many of these, with effects from trivial to catastrophic, have been traced to unintended consequences of changes
- One example of a change of a single character in a program cost a company US (circa 1980)$1.6 billion [Weinberg 83]
- Purushothaman & Perry have shown that approximately 10% of changes altered a single line of code and of these about 4% resulted in additional faults. They also showed that approximately 40% of all changes resulted in additional faults.
Related work: static analysis

- One common approach for regression testing relies on comparing two static dependence graphs
  - Safe regressions
  - Test prioritization
  - ...
- Where dynamic analysis is not part of the analysis itself – as is most common, our approach makes distinctions not possible using only static techniques
Related work: dynamic analysis

- Several approaches use a single program and compare executions across multiple behaviors
  - Reps et al. [ESEC/FSE ‘97] – compare program spectra to identify Y2K bugs
  - Software reconnaissance to distinguish program features [Wilde and Scully 1995]
  - Tripoli can be used to compare two arbitrary executions of a system and determine their coverage differences, but assumes the source code has not changed [Sherwood and Murphy]
Ongoing/future work

- Additional empirical analysis – different systems, different analyzers, etc.
- External qualitative analysis of the value of reported partitions
- Granularity of reporting – on desktop, pre-commit, nightly-build, etc.
- Searching historical partitions to answer questions like, “when did this callback stop happening?” or, “when was this behavior added to the code?”
- Software complexity measures – why are they always based on static structure alone?
Conclusions

• Program behaviors can be hard to discern from the source code
• Most behaviors are not surprising --- by combining simple static and dynamic analyses we are able to concisely partition inconsistent behavior
• The developers must assess such behaviors, and guidance to specific such behaviors can be valuable
• Long-term: can an approach like this dampen degradation of the source-behavior relationship? And if so, does it matter?
Questions?