Learning from Mistakes
Andreas Zeller

Fixing the Process

• Any defect escaping into the wild should have been caught by local quality assurance
• Besides fixing the defect, we also must fix quality assurance!

Things to do

• Improve your test suite
• Set up assertions
• Improve training
• Improve the software process
• Improve the analysis tools
Things to Measure

- How much damage did the defect do?
- How much effort did it take to fix it?
- What is the risk we are taking in letting such defects go unnoticed?

Some Facts

- In Eclipse and Mozilla, 30–40% of all changes are fixes (Sliverski et al., 2005)
- Fixes are 2–3 times smaller than other changes (Mockus + Votta, 2000)
- 4% of all one-line changes introduce new errors (Purushothaman + Perry, 2004)

More Facts

- A module that is one year older has 30% less errors (Graves et al., 2000)
- New code is 2.5 times as defect-prone as old code (Ostrand + Weyuker, 2002)
Learning from History

The Risk of Change

• Some locations in a program are risky: many changes result in a fix

```java
public IRuntimeClasspathEntry[] resolveClasspath(IRuntimeClasspathEntry[] entries, ILaunchConfiguration configuration) throws CoreException {
    List all = new ArrayList(entries.length);
    for (int i = 0; i < entries.length; i++) {
        switch (entries[i].getType()) {
            case IRuntimeClasspathEntry.PROJECT:
                all.add(entries[i]);
                break;
            case IRuntimeClasspathEntry.OTHER:
                IRuntimeClasspathEntry2 entry = (IRuntimeClasspathEntry2)entries[i];
                if (entry.getTypeId().equals(DefaultProjectClasspathEntry.TYPE ID)) {
                    IRuntimeClasspathEntry[] children = entry.getRuntimeClasspathEntries(configuration);
                    IRuntimeClasspathEntry[] res =
                        JavaRuntime.resolveSourceLookupPath(children, configuration);
                    for (int j = 0; j < res.length; j++) {
                        all.add(res[j]);
                    }
                    break;
                }
            default:
                IRuntimeClasspathEntry[] resolved =
                    JavaRuntime.resolveRuntimeClasspathEntry(entries[i], configuration);
                for (int j = 0; j < resolved.length; j++) {
                    all.add(resolved[j]);
                }
                break;
        }
    }
    return (IRuntimeClasspathEntry[]) all.toArray(new IRuntimeClasspathEntry[all.size()]);
}
```
The most risky code in Eclipse

- 1.3 if (buildVM != null) { bug 16313
- 1.4 function deleted bug 7999
- 1.5 reimplementation bug 26681
- 1.7 Bug 44877 - Wrong JDK source lookup
- 1.8 Fall back fix for bug 44877 undid 1.7.
- 1.10 once again a switch statement
- 1.12 VariableClasspathEntry.TYPE_ID ...

8 out of 9 changes resulted in later fixes

Fixes and Changes

- How do we know a change is a fix?

The problem database relates fixes to problems

Hints for relating problems and fixes include

- Problem ID in the log message of the fix: Fixed bug 53784: .class file missing
- Changes before closing a problem: Before closing #53784, changed This.java
- For about 50% of all closed problems, we can identify the related fix
Fix-Inducing Changes

- Can I predict the risk of change?
- Which are the risky locations?
- Do they have common features?

What makes changes risky?

To determine whether changes induce risk, a number of metrics have been proposed:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of file being changed</td>
<td>Size of the change</td>
</tr>
<tr>
<td>Number of changes so far</td>
<td>Number of fixes so far</td>
</tr>
</tbody>
</table>

What makes changes risky?

Our claim: past risk at the change location is best predictor for future risk

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td># of past fix-inducing changes</td>
<td># of past fix-inducing fixes</td>
</tr>
<tr>
<td>at the change location</td>
<td>at the change location</td>
</tr>
</tbody>
</table>
Recall of 168 most risky files in Eclipse

Past risk is best predictor for future risk

Hatari

Change here is risky

Most risky locations
Risk along the Week

- Mozilla
- Eclipse

<table>
<thead>
<tr>
<th>Day</th>
<th>Mozilla</th>
<th>Eclipse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Tue</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>Wed</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Thu</td>
<td>75</td>
<td>75</td>
</tr>
<tr>
<td>Fri</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Sat</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Sun</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

What makes changes risky?

- Past risk at the location
- The day of the week
- Properties of the code?
**Risk ⇒ Complexity**

- A location is *complex* if it is risky to change
- *Factual complexity measure* – in contrast to *metrics* like McCabe and related
- Risk of change allows for *evaluation* and *mining* of metrics

**Mining Metrics**

Which features correlate with risk?

<table>
<thead>
<tr>
<th>do...while</th>
<th>multiple inheritance</th>
<th>DirectX API</th>
</tr>
</thead>
<tbody>
<tr>
<td>iterators</td>
<td>no iterators</td>
<td>method size</td>
</tr>
<tr>
<td>developer</td>
<td>use of XP</td>
<td>and more…</td>
</tr>
</tbody>
</table>

Correlation specific to project – or universal

**Requirements**

- Well-kept version and bug databases
- Link between changes and problems
- Willingness to change
- Policy on how to handle sensitive data
Space Shuttle Software

Problem Tracking

• When was the error discovered? How? Who? What flight?
• How was the error introduced? Why wasn’t it caught?
• How was the error corrected? Are there similar errors?
• What can we learn from previous errors?

The Process

• Software error = an error in the process
• Planning the software carefully in advance
• Reducing risk at all stages
• Keeping record of all activities
• “Not even rocket science” – just standard practice in engineering