How to improve the quality of your application

(I wish I’d known this earlier!)

Ioannis Kolaxis – Coding Architect / Senior Expert

Tuesday 10th December 2019
Java2Days, Sofia / Bulgaria
Our application: CMP

- Automates the configuration & provisioning of our products, achieving significant time savings for our service.
Are you working for a software product, where ...?

- *Customers* keep complaining about *bugs*
- *New features* take *too much time* to be implemented
What can you do?

- Can you improve the quality of your software?
- How?
Customer tickets ✔

- We usually measure *quality* via customer tickets:
Code coverage ✓

• When we refer to *quality*, we usually think of code coverage!

Increased coverage:
• from 67% (Feb 2017)
• to 72% (Jun 2018)
Should you pay off your debt?

Decreased debt:
- from 1.359 days (Feb 2017)
- to 392 days (Jun 2018)
Old code is more reliable

Do not touch old code!
You will probably introduce new defects!

“If a module is, on the average, a year older than an otherwise similar module, the older module will have roughly a third fewer faults.”

Stop creating new debt

- Install SonarLint plugin in your IDE.
- It helps you detect, and fix quality issues as you write code.
- Download at: www.sonarlint.org
Stop creating new debt

- Setup Quality Gates in SonarQube
Quizz

As a developer, where do you spend most of your time?

A. Reading existing code,
B. Writing new code,
C. Waiting for a full build to complete,
D. Other
Quiz

- As a developer, where do you spend most of your time?
  A. Reading existing code,
  B. Writing new code,
  C. Waiting for a full build to complete,
  D. Other
Just think …

Which parts of your code do you read most often?
Data never lies

- Use **git** to find out where you spend most of your development efforts:

  ```bash
  git log --format=format:--name-only | grep -v '^$' | sort | uniq -c | sort -r > files_change_frequency.txt
  ```

Commits per file:

- 258 usermanagementportlet/.../UserManagement_de.properties
- 250 usermanagementportlet/.../UserManagement_en.properties
- 227 usermanagement/.../RetrieveUserTmpltForUsersDataControlImpl.java
- 205 usermanagement/.../UserManagementImpl.java
- 154 usermanagement/.../EditUserResourceTemplateRulesBean.java
- 135 usermanagementportlet/.../AddEditUserBean.java
- 109 usermanagementportlet/.../ConfigureNewUserResourceBean.java
- 103 usermanagementportlet/.../addEditUser.jsp

@IoannisKolaxis
The pattern

- Only a few files change frequently!
- This is where you spend most of your time!

From a total of 10,007 files:
- 11 files → more than 100 commits
- 91 files → 31 < commits < 100
- 455 files → 10 < commits < 30
- 9,450 files → less than 10 commits
A well-aimed refactoring will help you:

• Spend **less time** to read code & extend functionality.

• Become **more productive**!
Changing files predict system failures

• “Churn measures based on counts of lines added, deleted, and modified are very effective for fault prediction.”

• Files involved in a lot of bug fixing activities are most likely to be defective
Focus your Quality Assurance efforts

• Do not waste your time testing *mature* functionality (=components that do not change).

• Focus all your testing efforts on the *frequently-changing parts*; those are most likely to fail!
Ask the right questions

What is the coverage of your new/changing code?
Identify stable components

- Files not changed in the past years → stable components → mature features

- Is every mature feature still used by your customers?
  - If a feature is not used, then delete its code!
  - Else, extract stable features in separate libraries.
Go faster with deleted/extracted code

• Save time from your builds.
• Achieve faster onboarding of new developers, by:
  • Focusing only on actively developed code.
  • Not having to familiarize with old/stable code.
Measure code complexity

• Gain more insight, by measuring code complexity for each one of the frequently changing files.

• Language-neutral metrics for code complexity:
  • Number of lines
  • Number of tabs
Tabs increase complexity

- How many times did you provide a bug fix, by adding a nested conditional in your code?

```java
if (...) {
    for (...) {
        if (customerSpecificSetup) {
            tabs // Do some magic, so that the // application works for this customer!
        }
    }
}
```
Rising complexity calls for refactoring

227 commits
Our #1 priority for refactoring

205 commits
Our #1 priority for refactoring

UserManagementImpl.java

6.767 → 8.396 lines
22.421 → 29.310 tabs

Sept 2014

205 commits
Refactor frequently changing files

• The identified files are being changed by many developers in parallel.

• Is it feasible to perform refactoring on a private branch?

• Can we afford to stop development, while someone works for a long time on refactoring the identified files?
Break large file by responsibilities

**Original file has too many responsibilities**

UserManagementImpl.java

- findUser()
- addUser()
- editUser()
- deleteUser()
- findExtensionRange()
- addExtensionRange()
- editExtensionRange()
- deleteExtensionRange()
- getAssignedPhones()
- getUnassignedPhones()

---

**Delegate old method calls to new classes**

UserMgmt.java

- findUser()
- addUser()
- editUser()
- deleteUser()

ExtensionRangeMgmt.java

- findExtensionRange()
- addExtensionRange()
- editExtensionRange()
- deleteExtensionRange()

PhonesMgmt.java

- getAssignedPhones()
- getUnassignedPhones()
Divide and conquer

- When you refactor, always try to stabilize new/changing code!
## Stabilizing code by refactoring

<table>
<thead>
<tr>
<th>Original file</th>
<th>Refactor</th>
<th>Refactored file</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UserManagementImpl.java</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>findUser()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addUser()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>editUser()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>deleteUser()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Original file</th>
<th>Refactor</th>
<th>Refactored file</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UserManagementImpl.java</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>findExtensionRange()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addExtensionRange()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>editExtensionRange()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>deleteExtensionRange()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Original file</th>
<th>Refactor</th>
<th>Refactored file</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UserMgmt.java</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>findUser()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addUser()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>editUser()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>deleteUser()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Original file</th>
<th>Refactor</th>
<th>Refactored file</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ExtensionRangeMgmt.java</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>findExtensionRange()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addExtensionRange()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>editExtensionRange()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>deleteExtensionRange()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Original file</th>
<th>Refactor</th>
<th>Refactored file</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PhonesMgmt.java</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>getAssignedPhones()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>getUnassignedPhones()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

@IoannisKolaxis
Do you remember Windows Vista?

- Released on 8\textsuperscript{th} November 2006.
- > 50 million lines of code.
- \(\sim\) 2,000 developers.
Organizational structure vs Quality

- Microsoft measured several organizational metrics, and studied their correlation with the defects of Windows Vista.

<table>
<thead>
<tr>
<th>Organizational metric</th>
<th>Assertion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Engineers</td>
<td>The more people who touch the code, the lower the quality.</td>
</tr>
<tr>
<td>Number of Ex-Engineers</td>
<td>A large loss of team members affects the knowledge retention, and thus quality.</td>
</tr>
<tr>
<td>Organization Intersection Factor</td>
<td>The more diffused the different organizations contributing code, the lower is the quality.</td>
</tr>
</tbody>
</table>

- Can the **structure of your organization** affect the **quality** of your software application?

Organizational structure impacts Quality

- **Organizational metrics** are better predictors of **failure-proneness** than the traditional metrics used so far, such as **code coverage**, **code complexity**, etc.

<table>
<thead>
<tr>
<th>Model</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Organizational structure</td>
<td>86,2%</td>
</tr>
<tr>
<td>Code coverage</td>
<td>83,8%</td>
</tr>
<tr>
<td>Code complexity</td>
<td>79,3%</td>
</tr>
<tr>
<td>Code churn</td>
<td>78,6%</td>
</tr>
<tr>
<td>Dependencies</td>
<td>74,4%</td>
</tr>
<tr>
<td>Pre-release bugs</td>
<td>73,8%</td>
</tr>
</tbody>
</table>

More organizational metrics

- In another research, focused on Windows 7, Microsoft distinguished between the following kinds of developers, depending on their commits for a given component:
  - **Owner:** has the most commits to that component.
  - **Major contributor:** has *more than 5%* of total commits.
  - **Minor contributor:** has *less than 5%* of total commits.

The researchers concluded that:

• “The number of minor contributors has a strong positive relationship with both pre- and post-release failures ...”
• “Higher levels of ownership for the top contributor to a component results in fewer failures when controlling for the same metrics, but the effect is smaller than the number of minor contributors”

In one of our software components, we had a total of 427 commits:

- The top contributing developer made 87 commits: $\frac{87}{427} = \textit{20,37\%}$ ownership

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minor contributors</td>
<td>15</td>
</tr>
<tr>
<td>Major contributors</td>
<td>6</td>
</tr>
<tr>
<td>Total contributors</td>
<td>21</td>
</tr>
<tr>
<td>Ownership</td>
<td>20,37%</td>
</tr>
</tbody>
</table>
Gain insight into your components

• In another software component, we had a total of 253 commits for the same period:

- **Minor contributors**: 3
- **Major contributors**: 6
- **Total contributors**: 9

**Ownership**: 28.85%

• The top contributing developer made 73 commits:

\[
\frac{73}{253} = 28.85\% 
\] ownership
Know where you are standing …

<table>
<thead>
<tr>
<th>Metric</th>
<th>Component A</th>
<th>Component B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minor contributors</td>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>Major contributors</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Total contributors</td>
<td>21</td>
<td>9</td>
</tr>
<tr>
<td>Ownership</td>
<td>20.37%</td>
<td>28.85%</td>
</tr>
</tbody>
</table>

- Which component will probably have more defects?
- Where would you focus your testing efforts?
Beware of minor contributors!

<table>
<thead>
<tr>
<th>Metric</th>
<th>Component A</th>
<th>Component B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minor contributors</td>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>Major contributors</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Total contributors</td>
<td>21</td>
<td>9</td>
</tr>
<tr>
<td>Ownership</td>
<td>20,37%</td>
<td>28,85%</td>
</tr>
</tbody>
</table>

- More **minor contributors** → More defects
- Bigger **ownership** → Less defects
Use metrics to build better software

- Minor contributors must be consulting a major contributor of a component before making any changes to it.
- Pay more attention when reviewing code submitted by minor contributors.
- More extensive testing should be performed for components with low ownership.
Planning new features

- A customer asks for a **new feature** to be implemented, but the **major contributors** of that component are **not available**. What will you do?
  - Ask from **minor contributors**, to start implementing this new feature right away, or
  - Delay the implementation of the feature, until one or more **major contributors** are available?
Learn your contributors

• Use `git` to find out all the contributors for a component:

```bash
$ git shortlog -s your_component > contributors.txt
```

17  Ioannis Kolaxis
18  ...
34  ...

• Or, to limit the results to contributors after a given date (e.g. due to an organizational restructuring)

```bash
$ git shortlog -s --after=2018-05-01 your_component > contributors.txt
```
Summary of proposed actions

1. Stop creating new quality issues.
2. Don’t touch old code.
3. Refactor your most complex, frequently changing files.
4. Focus your testing on frequently changing files.
5. Pay attention to minor contributors.
How do you build quality software?

Let’s share our knowledge & experience!
Thank you!

Email: ioannis.kolaxis@atos.net
Twitter: @ioannisKolaxis