How to improve the quality of your application

(I wish I'd known this earlier!)

Ioannis Kolaxis – Coding Architect / Senior Expert

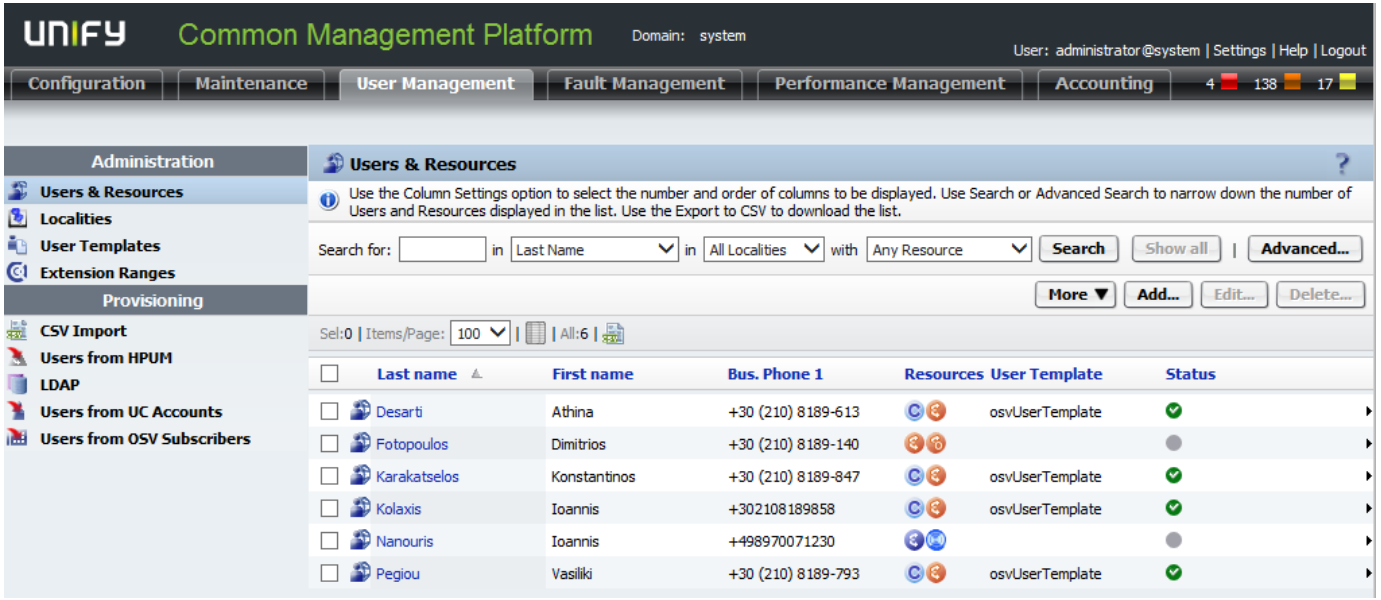
Monday 16th September 2019
Oracle Code One, San Francisco / USA

Trusted partner for your Digital Journey

Atos

Our application: CMP

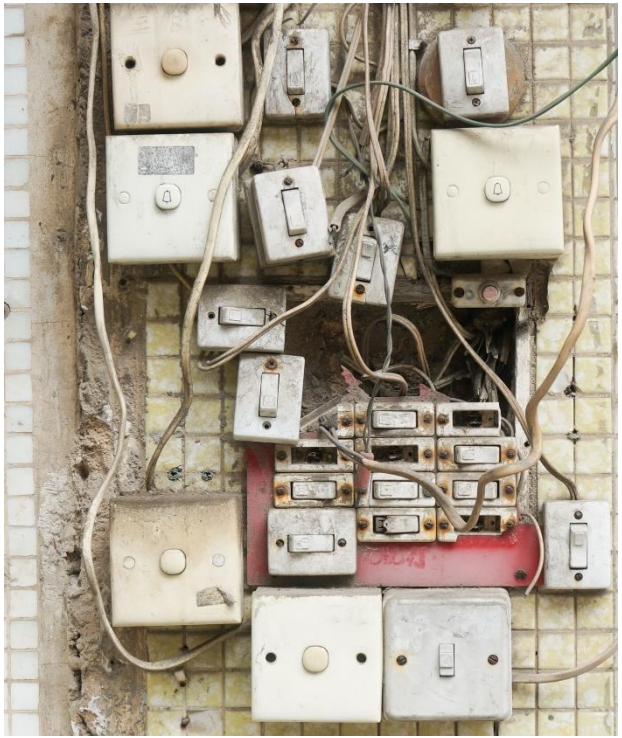
- Automates the configuration & provisioning of our products, achieving significant time savings for our service.



@IoannisKolaxis

Atos

Software quality issues



Are you working for a software product, where …?

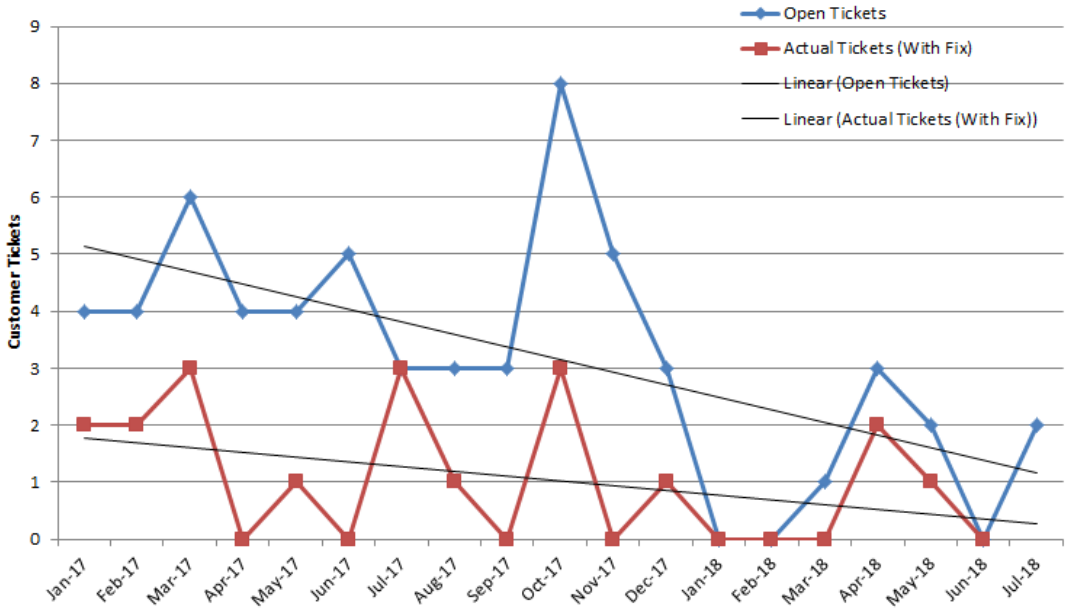
- *Customers* keep complaining about **bugs**
- *New features* take **too much time** to be implemented

What can you do?

- Can you improve the quality of your software?
- How?

Customer tickets ✓

- We usually measure *quality* via customer tickets:



CMP Customer tickets

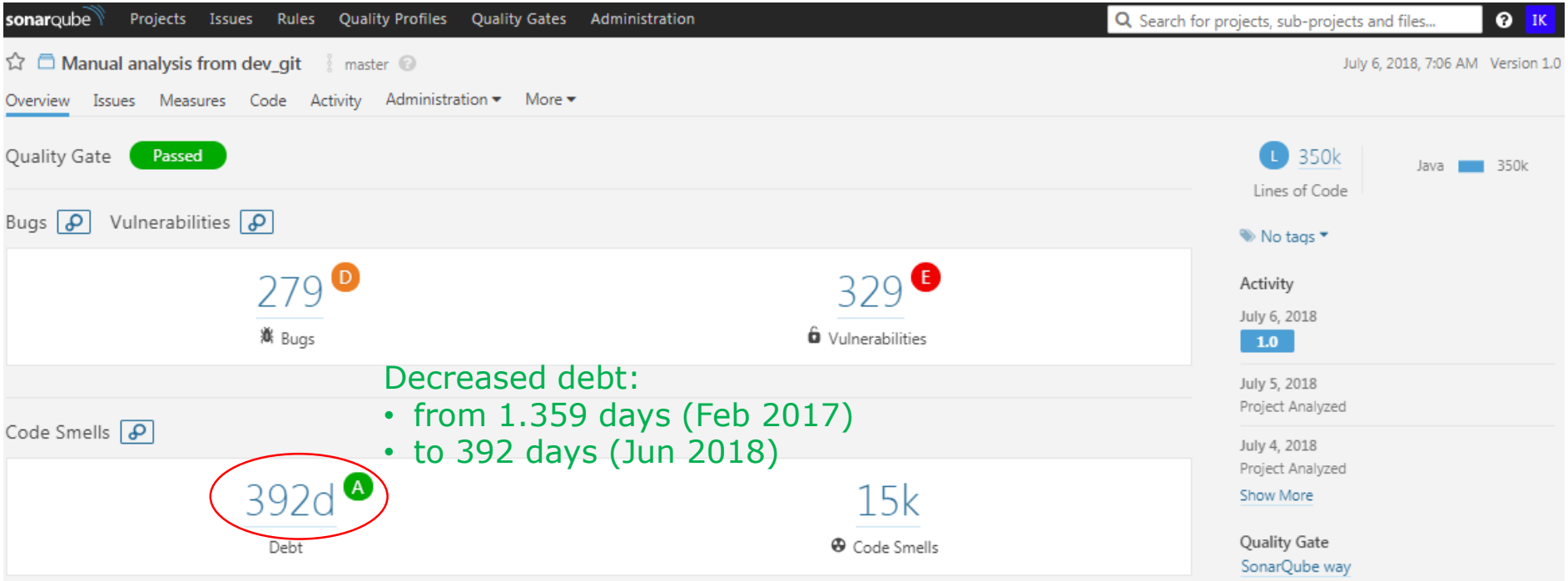
Code coverage ✓

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

Increased coverage:
- from 67% (Feb 2017)
- to 72% (Jun 2018)

@IoannisKolaxis

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!

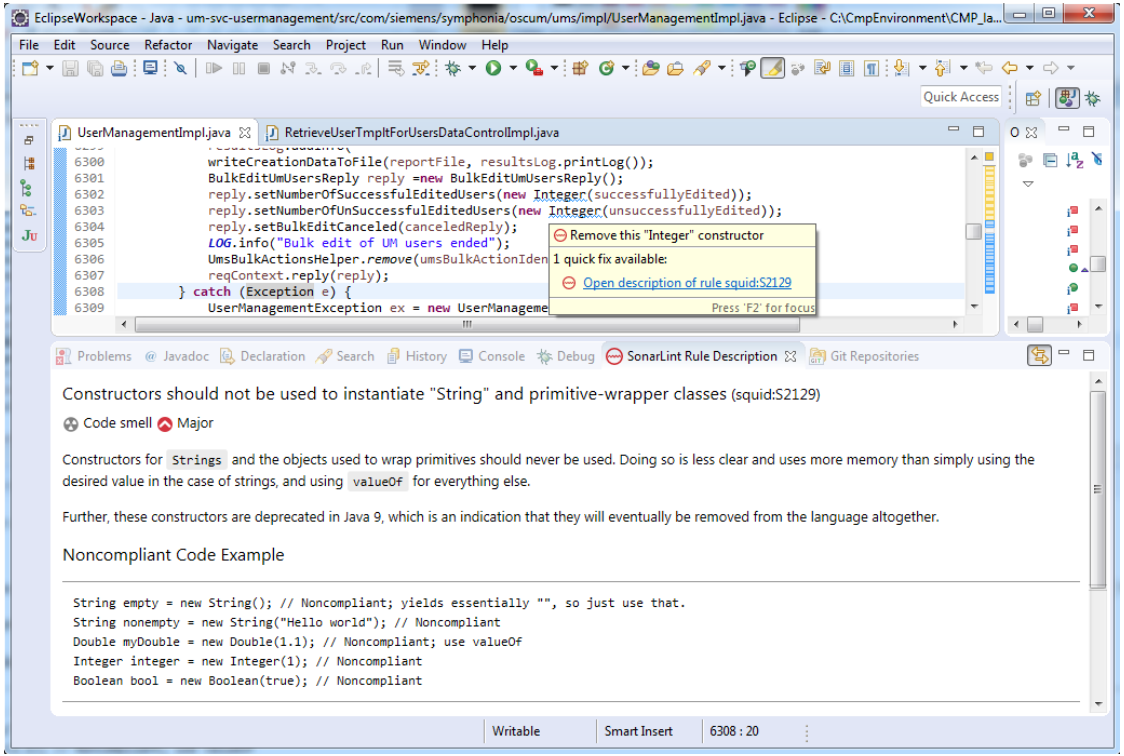
"If a module is, on the average, a year older than an otherwise similar module, the older module will have roughly a third fewer faults."

T. L. Graves, A. F. Karr, J. S. Marron and H. Siy, "Predicting fault incidence using software change history" in *IEEE Transactions on Software Engineering*, vol. 26, no. 7, pp. 653-661, Jul 2000.

Stop creating new debt

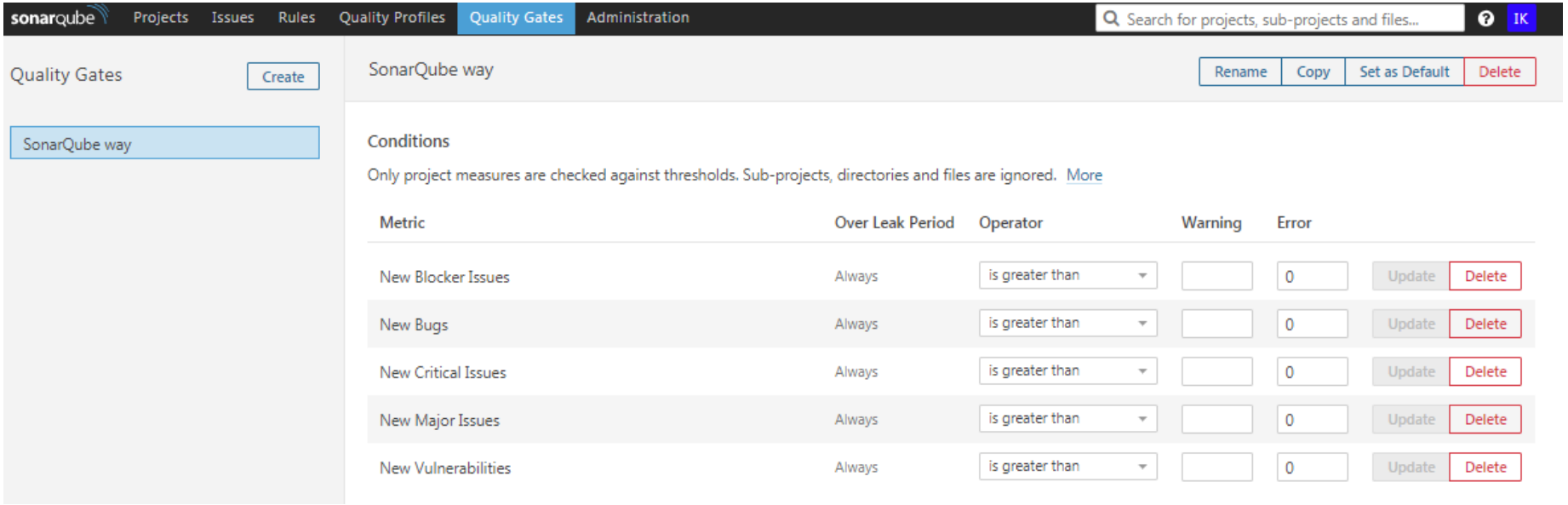
- Install SonarLint plugin in your IDE.
- It helps you detect, and fix quality issues as you write code.
- Download at:

  www.sonarlint.org



Stop creating new debt

- Setup Quality Gates in SonarQube



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

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:

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

Commits per file

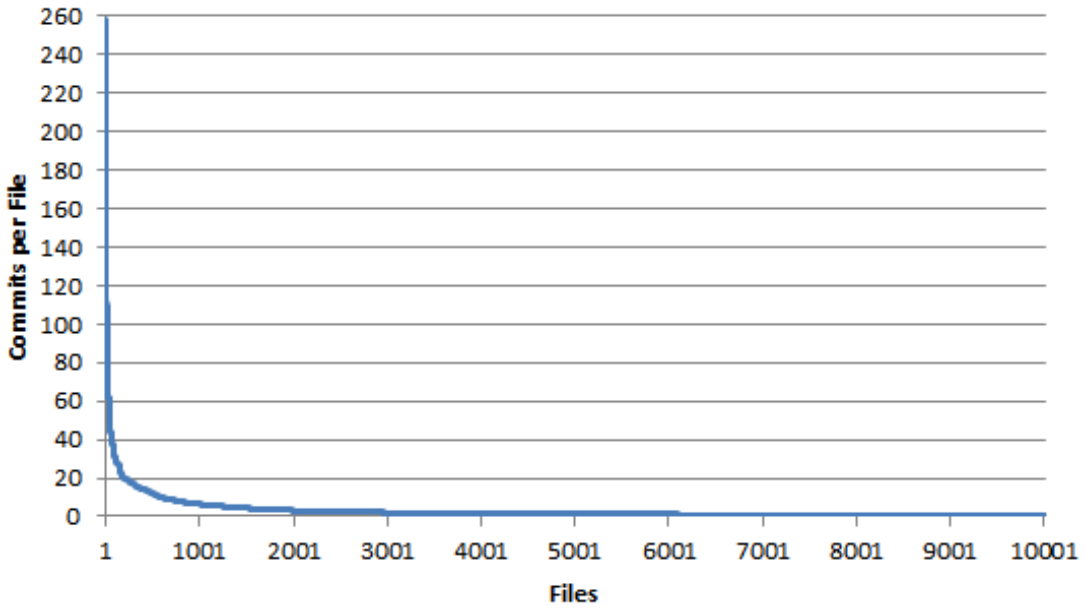
```
258 usermanagementportlet/…/UserManagement_de.properties
250 usermanagementportlet/…/UserManagement_en.properties
227 usermanagement/…/RetrieveUserTmpltForUsersDataControlImpl.java
205 usermanagement/…/UserManagementImpl.java
154 usermanagement/…/EditUserResourceTemplateRulesBean.java
135 usermanagementportlet/…/AddEditUserBean.java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

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*



Refactor frequently-changing files

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."*

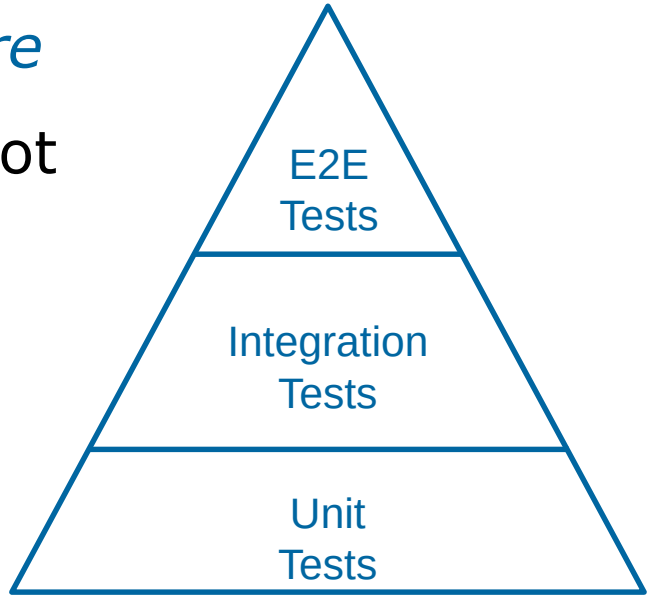
  R. M. Bell, T. J. Ostrand, and E.J. Weyuker, "Does Measuring Code Change Improve Fault Prediction?", ACM Press, 2011.

- *Files involved in a lot of bug fixing activities are most likely to be defective*

  R. Moser, W. Pedrycz, and G. Succi, "A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction", Proceedings of the 30th International Conference on Software Engineering, 181-190, 2008.

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.

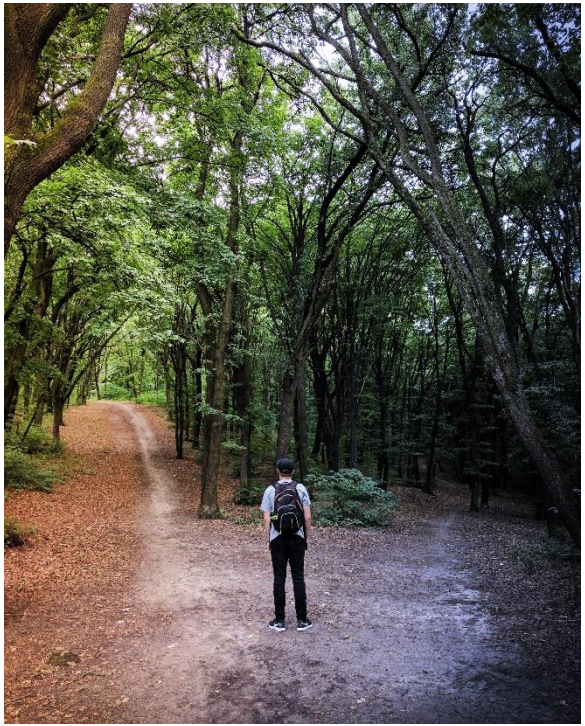
Go faster with deleted/extracted code

- 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

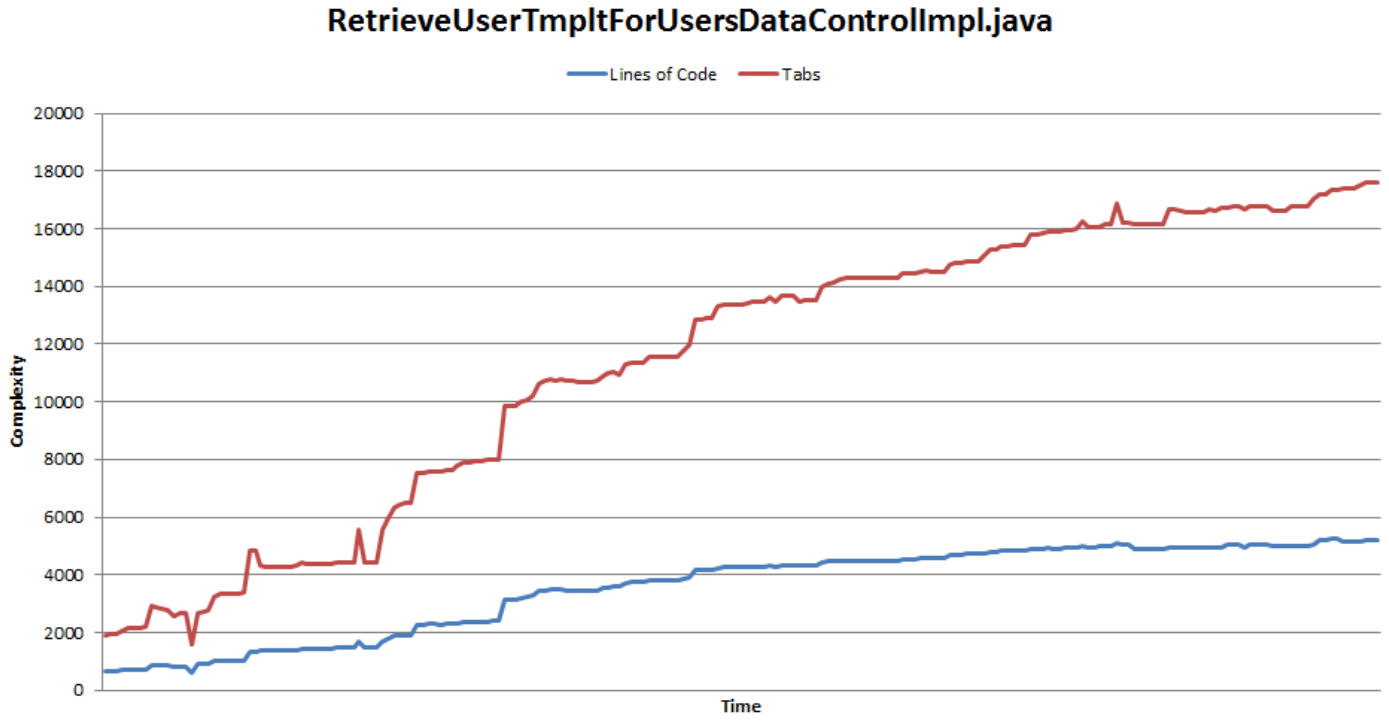


- How many times did you provide a bug fix, by adding a nested conditional in your code?

```
if (…) {
    for (…) {
        if (customerSpecificSetup) {
            // Do some magic, so that the
            // application works for this customer!
        }
    }
}
```

tabs

Rising complexity calls for refactoring

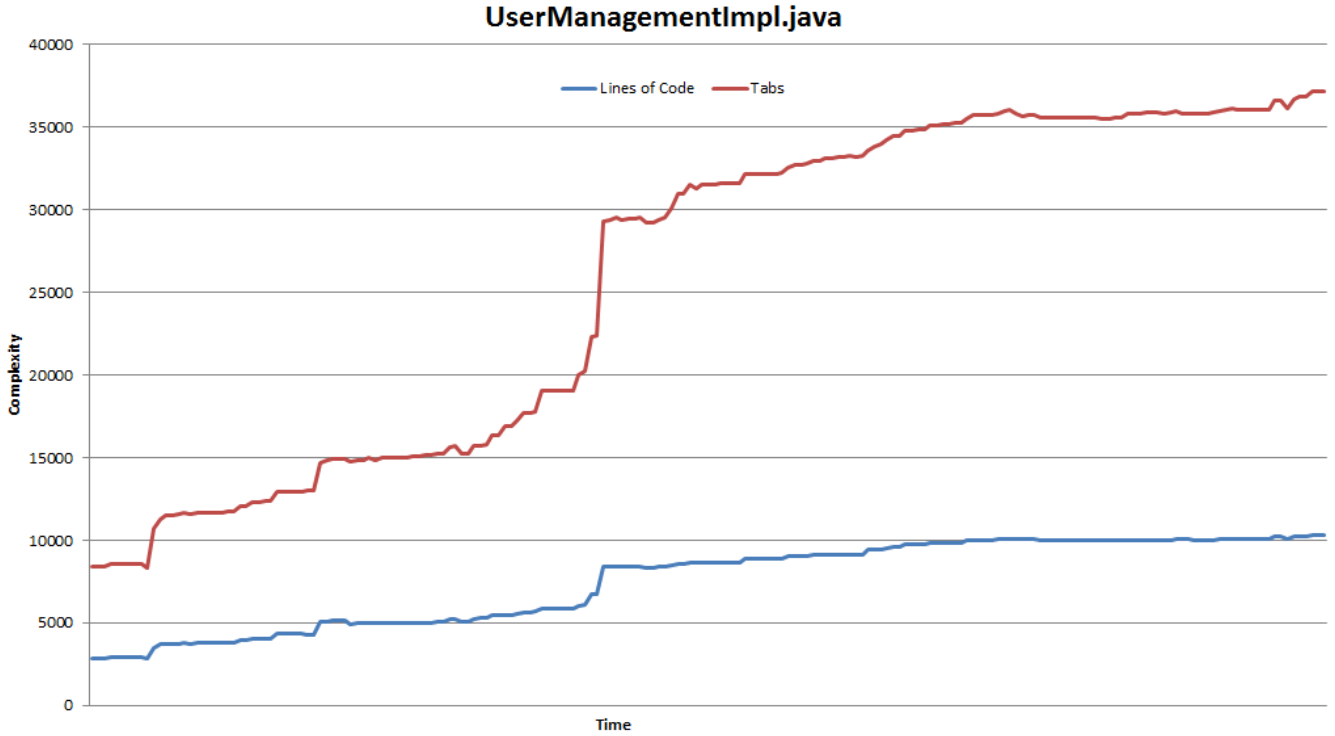


227 commits

@IoannisKolaxis

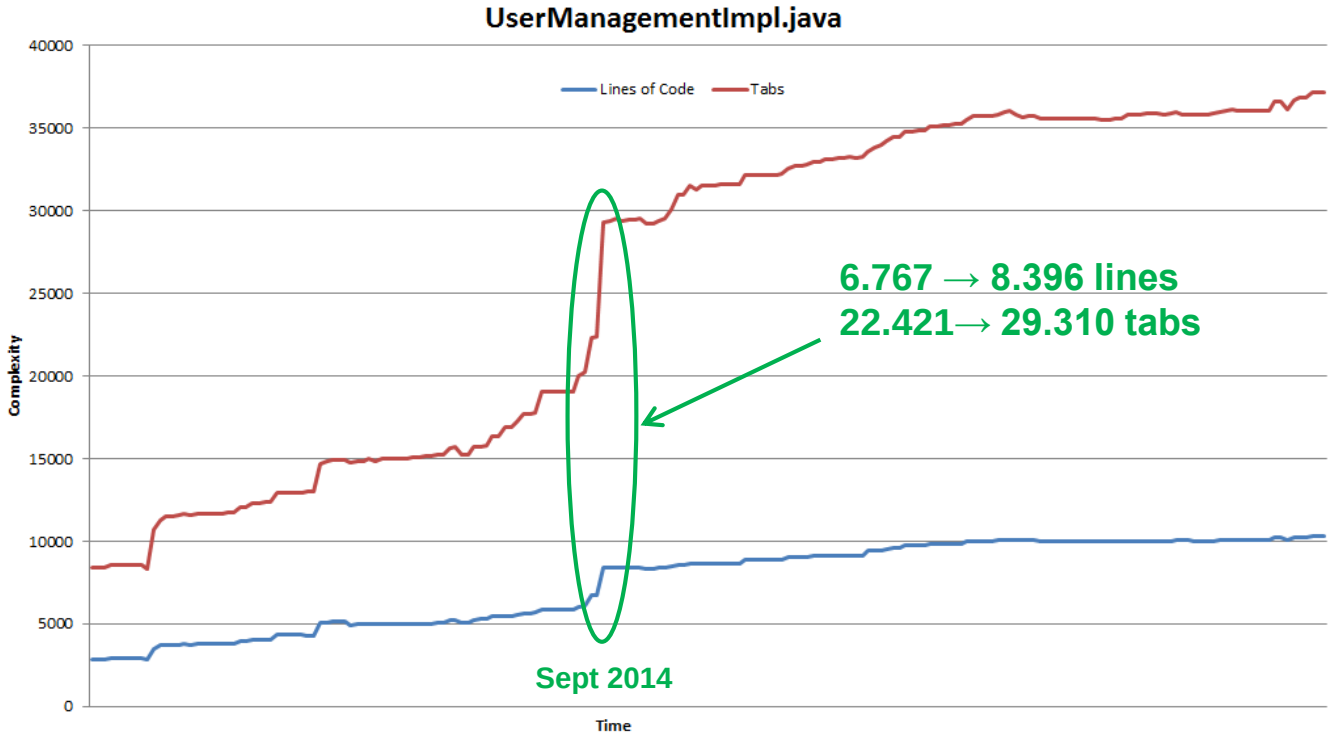
Atos

Our #1 priority for refactoring



205
commits

Our #1 priority for refactoring



205
commits

Do you remember Windows Vista?

- Released on 8th November 2006.
  - > 50 million lines of code.
  - ~ 2.000 developers.

Organizational structure vs Quality

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

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

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

N. Nagappan, B. Murphy, and V.R. Basili, "The Influence of Organizational Structure on Software Quality: An Empirical Case Study", ACM, 2008.

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.

Model	Precision
Organizational structure	**86,2%**
Code coverage	83,8%
Code complexity	79,3%
Code churn	78,6%
Dependencies	74,4%
Pre-release bugs	73,8%

N. Nagappan, B. Murphy, and V.R. Basili, "The Influence of Organizational Structure on Software Quality: An Empirical Case Study", ACM, 2008.

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.

C.Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't Touch My Code! Examining the Effects of Code Ownership on Software Quality", ACM, 2011.

Effects of minor contributors

- 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"*

C.Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't Touch My Code! Examining the Effects of Code Ownership on Software Quality", ACM, 2011.

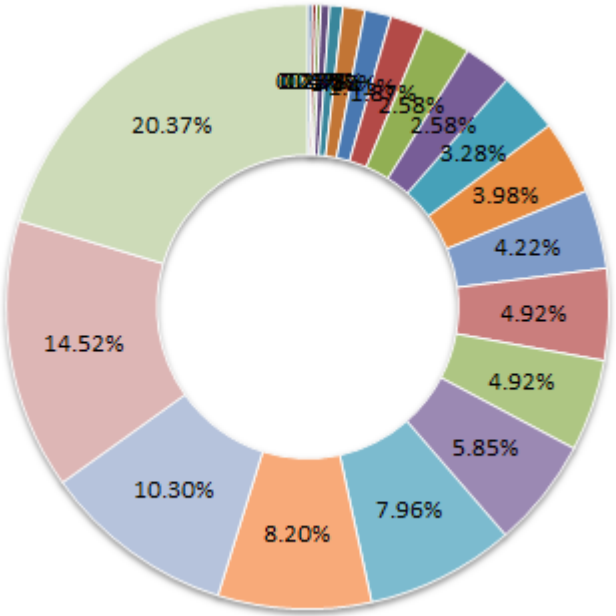
Gain insight into your components

- In one of our software components, we had a total of 427 commits:
  - The top contributing developer made 87 commits:

87/427 = **20,37%** ownership

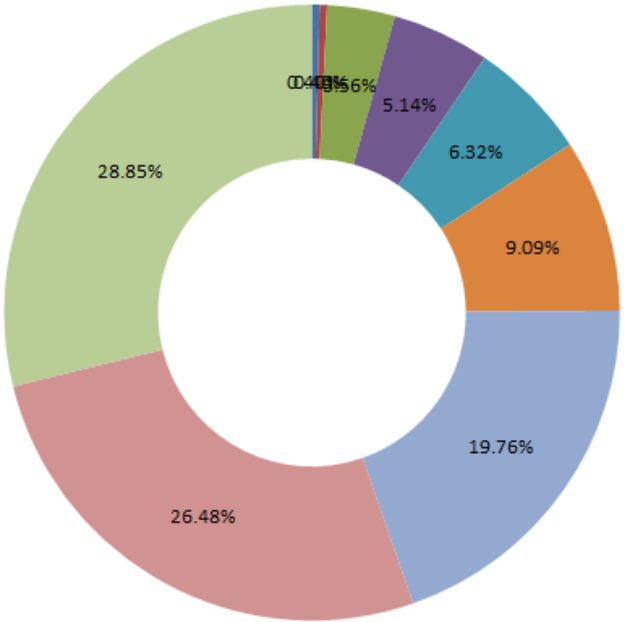
Metric	Value
Minor contributors	15
Major contributors	6
Total contributors	**21**
Ownership	**20,37%**

Commits per developer



Gain insight into your components

Commits per developer



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

Metric	Value
Minor contributors	3
Major contributors	6
Total contributors	**9**
Ownership	**28,85%**

- The top contributing developer made 73 commits:

73/253 = **28,85%** ownership

Know where you are standing …

Metric	Component A	Component B
Minor contributors	15	3
Major contributors	6	6
Total contributors	**21**	**9**
Ownership	**20,37%**	**28,85%**

- Which component will probably have more defects?
- Where would you focus your testing efforts?

Beware of minor contributors!

Metric	Component A	Component B
Minor contributors	15	3
Major contributors	6	6
Total contributors	**21**	**9**
Ownership	**20,37%**	**28,85%**

- 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***.

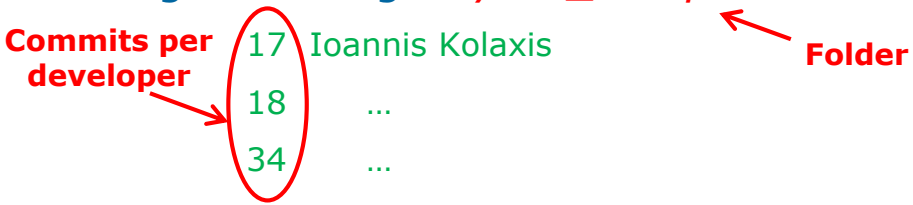
Planning new features

- 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:

  git shortlog -s *your_component* > contributors.txt



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

  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

Atos