THE TECHNICAL DEBT METAPHOR: PRINCIPLES, PRACTICAL CONSIDERATIONS, TOOL SUPPORT, AND RESEARCH DIRECTIONS

Davide Falessi
STC 2015
ABOUT THE PRESENTER

Dr. Davide Falessi (dfalessi@calpoly.edu)
Associate Professor
Computer Science Department
College of Engineering, Cal Poly

- **IEEE Software**, Associate Editor of Software Engineering Economics and Multimedia Editor
- Guest Editor of the Special Issue of *Technical Debt* in the Journal of Systems and Software – Elsevier “Impact Factor: 1.245”.
- At least 8 international peer-review publications about *Technical Debt*.
- Organizer of several international peer-reviewed workshops on *Technical Debt*.
- First author of several papers in the most important software engineering journals including TSE, TOSEM, EMSE, and IEEE Software.
- PC member of several international software engineering conferences (~8 per year).
Motto: Learn by doing.

Faculty: 1303

Student: 20k

Bachelor programs: 64

Master programs: 31

CS Department:

- Mission statement: teaching students computer science and software engineering principles and how to apply them to solve practical problems in a socially responsible way.

- Faculty: 22 tenure/tenure track, and approx. 10 lecturers.
Aim

- Provide an overview about Technical Debt
- Why?
DEFINITIONS
DEFINITIONS

- Technical Debt (TD) can be seen as the result of an optimization for the **short-term** which leads to **long-term** handicaps.
  - Examples: low comments density, code smell, code clone, etc.
- TD can:
  - Emerge *organically* because every system, while evolving, improves complexity.
  - Be *opportunistically* chosen; e.g., “let’s release now, we’ll deal with it later on”.
DEFINITIONS

- TD is the application of financial concepts to software engineering domain.

- TD is a metaphor:
  - Pros: widely applicable.
  - Cons: prone to misuses.
DEFINITIONS

TD consists of two important concepts:

- **Principal**: the cost of eliminating the debt.
- **Interest**: the penalty to be paid in the future if the debt is not eliminated.

E.g.: a high complex module

- **Principal**: the required effort for refactoring it
- **Interest**: extra defects or the reduced speed (caused by the extra complexity).
VIOLATIONS VS DEFECTS

- **Quality Rule**: An empirically validated software engineering principle dictating how the code should be. Example of quality rules include: high comments density, low code complexity, short conditional statements, etc.

- **Violation**: A portion of a code that is not compliant with a specific rule.

- **Defect**: A problem in the code that require some effort to be fixed. In this context, quality rules are not defect. Quality rules relate to the **internal quality** of the software product and are **invisible** to the user. Vice versa, defects relate to the external quality of the code.
EXAMPLE OF RULE VIOLATION

```java
throws IOException {

    // Have we already authenticated someone?
    Principal principal = request.getUserPrincipal();
    //String ssoId = (String) request.getNote( Constants.REQ_SSOID_NOTE);

    // This block of commented-out lines of code should be removed.
```
MEASUREMENT APPROACH

- Violation density = Violations / Size in LOC
- Defects injection frequency = Defect proneness = Defects / LOC touched

![Diagram showing release cycles and calculations for violations and LOC touched]
INDUSTRIAL CASE STUDY RESULT
PRACTICAL CONSIDERATIONS
PRACTICAL CONSIDERATIONS

- TD can organically expire
  - All projects have a lifetime.
  - Contracts may not include maintainability.
  - According to the type of contract, maintainability problems can be an opportunity.
PRACTICAL CONSIDERATIONS

- **All projects** have some TD
  - In a world of finite resources, **imperfection** is the norm.
  - Setting **clients’ expectations** is vital.
Shall we eliminate TD?

Yes... but not always!

The success of a project depends also (especially) on business goals: cost, time-to-market and customers’ satisfaction.

TD must be managed, monitored, and analyzed. Always!
TOOL SUPPORT
Tool - State of the Art: SonarQube

- Open Source.
- Maintained by a big community.
- Largely used in industry.
- More than 20 programming languages.
- More than 50 plugins.
- More than 1200 quality rules like code smells (e.g., god classes) and coding style (e.g., comment density).
- Fast and easy to use.
- Several visualizations techniques.
### Example of SonarQube Visualization

<table>
<thead>
<tr>
<th>Category</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debt</td>
<td>4,048d</td>
</tr>
<tr>
<td>Issues</td>
<td>155,969</td>
</tr>
<tr>
<td>Blocker</td>
<td>548</td>
</tr>
<tr>
<td>Critical</td>
<td>8,294</td>
</tr>
<tr>
<td>Major</td>
<td>96,131</td>
</tr>
<tr>
<td>Minor</td>
<td>50,150</td>
</tr>
<tr>
<td>Info</td>
<td>846</td>
</tr>
<tr>
<td>Lines Of Code</td>
<td>1,171,561</td>
</tr>
<tr>
<td>Files</td>
<td>7,757</td>
</tr>
<tr>
<td>Functions</td>
<td>92,245</td>
</tr>
<tr>
<td>Java</td>
<td>1,150,816</td>
</tr>
<tr>
<td>Web</td>
<td>19,379</td>
</tr>
<tr>
<td>XML</td>
<td>679</td>
</tr>
<tr>
<td>Directories</td>
<td>472</td>
</tr>
<tr>
<td>Lines</td>
<td>2,428,925</td>
</tr>
<tr>
<td>Classes</td>
<td>11,243</td>
</tr>
<tr>
<td>Statements</td>
<td>519,281</td>
</tr>
<tr>
<td>Accessors</td>
<td>3,230</td>
</tr>
</tbody>
</table>

Source: nemo.sonarqube.org Projekt: JDK7
limits of sonarqube

What is the consequence of not spending 4,048 days?

If I have just one day:
- **Which** violation(s) shall I remove?
- Can I **objectively quantify** the improvement on my software?
MIND
MIND CHARACTERISTICS

- Apache licensed.
- Hosted in SourceForge.net.
- Written in Java as a SonarQube plug-in: 7K LOC.
- Versioned with Git.
- Tracked with Redmine.
- Tested on an open source project (Silverpeas: 56K LOC).
- Supports Git version control system.
- Support Redmine Issue Tracker system.
- Supports all programming languages supported by SonarQube.
MIND

- MIND assumptions and pre-conditions
  - High Linkage between commits and tickets.
  - High number of releases.
  - High variation in rule violations.
  - Git and Redmine.

- MIND deliverables
  - Instruction and Demo: http://goo.gl/Ydjtkq
  - Installation files: http://goo.gl/VvDv75
  - Redmine repository: http://goo.gl/4r7Y1a
  - Source files (Git): http://goo.gl/TWRnat
MIND ARCHITECTURE

MIND

Redmine Reader  Git Reader  Sonar Reader

Data Analysis

GUI

SonarQube System

Static Analyzer

Rules

GUI

Static Analysis

Configuration
User Story 1: Which is the most important quality rule?

As a developer I want to know which rule, when violated, provides the highest interest to pay so that I can avoid violating it or I can refactor when violated.
**USER STORY 1: TECHNICAL DETAILS**

- Importance of a rule:: Spearman's rank correlation coefficient between *defects injection frequency* and *violation density*.
### User Story 1: Technical Details

#### Violations Density

<table>
<thead>
<tr>
<th>Class ID</th>
<th>Version</th>
<th>Rule 1</th>
<th>Rule 2</th>
<th>Rule 3</th>
<th>Defects Injection Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class 1</td>
<td>4.4</td>
<td>0.218182</td>
<td>0</td>
<td>0.054545455</td>
<td>0.89</td>
</tr>
<tr>
<td>Class 2</td>
<td>4.4</td>
<td>0</td>
<td>0</td>
<td>0.035714286</td>
<td>0.24</td>
</tr>
<tr>
<td>Class 3</td>
<td>4.4</td>
<td>0.2125</td>
<td>0</td>
<td>0.0375</td>
<td>0.37</td>
</tr>
<tr>
<td>Class 4</td>
<td>4.4</td>
<td>0.150442</td>
<td>0</td>
<td>0.01179941</td>
<td>0.61</td>
</tr>
<tr>
<td>Class 5</td>
<td>4.4</td>
<td>0.131183</td>
<td>0.012903</td>
<td>0.008602151</td>
<td>0.38</td>
</tr>
<tr>
<td>Class 6</td>
<td>4.4</td>
<td>0.027397</td>
<td>0</td>
<td>0.04109589</td>
<td>0.61</td>
</tr>
<tr>
<td>Class 7</td>
<td>4.4</td>
<td>0.046154</td>
<td>0</td>
<td>0.023076923</td>
<td>0.72</td>
</tr>
<tr>
<td>Class 8</td>
<td>4.4</td>
<td>0.09375</td>
<td>0</td>
<td>0.09375</td>
<td>0.2</td>
</tr>
<tr>
<td>Class 9</td>
<td>4.4</td>
<td>0.029412</td>
<td>0</td>
<td>0.007352941</td>
<td>0.68</td>
</tr>
<tr>
<td>Class 1</td>
<td>4.5</td>
<td>0.210526</td>
<td>0</td>
<td>0.052631579</td>
<td>0.46</td>
</tr>
<tr>
<td>Class 2</td>
<td>4.5</td>
<td>0.207317</td>
<td>0</td>
<td>0.036585366</td>
<td>0.11</td>
</tr>
<tr>
<td>Class 3</td>
<td>4.5</td>
<td>0.14956</td>
<td>0</td>
<td>0.011730205</td>
<td>0.81</td>
</tr>
<tr>
<td>Class 4</td>
<td>4.5</td>
<td>0.130064</td>
<td>0.012793</td>
<td>0.008528785</td>
<td>0.7</td>
</tr>
<tr>
<td>Class 5</td>
<td>4.5</td>
<td>0.027397</td>
<td>0</td>
<td>0.04109589</td>
<td>0.33</td>
</tr>
<tr>
<td>Class 6</td>
<td>4.5</td>
<td>0.044776</td>
<td>0</td>
<td>0.02238806</td>
<td>0.35</td>
</tr>
<tr>
<td>Class 7</td>
<td>4.5</td>
<td>0.09375</td>
<td>0</td>
<td>0.09375</td>
<td>0.28</td>
</tr>
</tbody>
</table>

#### Rho

<table>
<thead>
<tr>
<th>Rule 1</th>
<th>Rule 2</th>
<th>Rule 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.6</td>
<td>0.1</td>
<td>0.3</td>
</tr>
</tbody>
</table>
USER STORY 2: HOW MUCH INTEREST AM I PAYING?

- As a developer, I want to know the interest I will have if I will not apply any refactoring, so that I can decide if refactoring activities are worthwhile.
**INTEREST DEFINITIONS**

- **Interest** = Average *extra* defect proneness (among classes).

- **Defect proneness** = Defect injection frequency of a class = Predicted number of defects / LOC touched.

- **Extra defect proneness** = Defect proneness with violations – Defect proneness without violations.

- **Relative extra defect proneness** = (Defect proneness with violations – Defect proneness without violations) / Defect proneness without violations.
# Training Table for Prediction Model

<table>
<thead>
<tr>
<th>Class ID</th>
<th>Version</th>
<th>Violations Density</th>
<th>Defect Proneness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class 1</td>
<td>v1.0</td>
<td>0.052</td>
<td>0.019</td>
</tr>
<tr>
<td>Class 2</td>
<td>v1.0</td>
<td>0.164</td>
<td>0.423</td>
</tr>
<tr>
<td>Class 3</td>
<td>v1.0</td>
<td>0.252</td>
<td>0.238</td>
</tr>
<tr>
<td>Class 4</td>
<td>v1.0</td>
<td>0.013</td>
<td>0.795</td>
</tr>
<tr>
<td>Class 5</td>
<td>v1.0</td>
<td>0.333</td>
<td>0.763</td>
</tr>
<tr>
<td>Class 6</td>
<td>v1.0</td>
<td>0.410</td>
<td>0.760</td>
</tr>
<tr>
<td>Class 1</td>
<td>v1.1</td>
<td>0.778</td>
<td>0.717</td>
</tr>
<tr>
<td>Class 2</td>
<td>v1.1</td>
<td>0.177</td>
<td>0.663</td>
</tr>
<tr>
<td>Class 3</td>
<td>v1.1</td>
<td>0.429</td>
<td>0.307</td>
</tr>
<tr>
<td>Class 4</td>
<td>v1.1</td>
<td>0.853</td>
<td>0.057</td>
</tr>
<tr>
<td>Class 5</td>
<td>v1.1</td>
<td>0.868</td>
<td>0.451</td>
</tr>
<tr>
<td>Class 6</td>
<td>v1.1</td>
<td>0.166</td>
<td>0.774</td>
</tr>
<tr>
<td>Class 1</td>
<td>v1.2</td>
<td>0.232</td>
<td>0.736</td>
</tr>
<tr>
<td>Class 2</td>
<td>v1.2</td>
<td>0.260</td>
<td>0.271</td>
</tr>
<tr>
<td>Class 3</td>
<td>v1.2</td>
<td>0.439</td>
<td>0.229</td>
</tr>
<tr>
<td>Class 4</td>
<td>v1.2</td>
<td>0.051</td>
<td>0.853</td>
</tr>
<tr>
<td>Class 5</td>
<td>v1.2</td>
<td>0.174</td>
<td>0.370</td>
</tr>
<tr>
<td>Class 6</td>
<td>v1.2</td>
<td>0.426</td>
<td>0.548</td>
</tr>
</tbody>
</table>
HOW MUCH INTEREST AM I PAYING?

- **Prediction details:**
  - Input variable: Current violation density.
  - Output variable: Defect proneness without violations.
  - Prediction model: Linear regression.
  - Validation technique: Leave-one-out (for optimizing data usage).
  - Type of error: Mean Absolute Error = Average of |Exp. – Act. value|
### MIND – USER STORY 2

**MIND (Managing Technical Debt)**

<table>
<thead>
<tr>
<th>Extra Defect Proneness</th>
<th>Defect Proneness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relative</td>
<td>Maximum per 100LOC</td>
</tr>
<tr>
<td>3.6%</td>
<td>0.1</td>
</tr>
</tbody>
</table>

### Quality Of Data

<table>
<thead>
<tr>
<th>Linkage</th>
<th>Estimation Error (MAE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>72.4%</td>
<td>0.0</td>
</tr>
</tbody>
</table>
MIND – User Story 3

“As a developer, I want to know the interest I paid, whether increased or decreased over time, so that I can do root-cause analysis.”

Figure 5: Snapshot of MIND’s visualization of the trend of the interest.
RESEARCH DIRECTIONS
MY RESEARCH DIRECTIONS

- Measuring interest as **decreased speed**.
- Prioritization of debt items also according to decreased speed.
- Better predictions by using several prediction models (other than Linear Regression) and several metrics like number of commits.
- Supporting **more technologies** like Rally as Issue Tracker System.
- Better architecture to easily integrate (plug and play) more technologies.
GENERAL RESEARCH DIRECTIONS

- Application of economic models.

- Understanding when to refactor.

- Understanding how to seamlessly integrate refactoring activities into software development.
ACKNOWLEDGEMENTS

- **Andreas Reichel**, Master student at Fraunhofer CESE

- **Cisco** research project CG# 583194
REFERENCES 1/2


REFERENCES 2/2


CONTACT INFORMATION

Davide Falessi
dfalessi@calpoly.edu