Technical Debt: The metaphor, the challenges and the opportunities

Alexander Chatzigeorgiou
• A few words about our team
• TD: the struggle for a definition
• Types of TD
• TD Activities
• Our research on TD
We are part of ..

Software Engineering Group  http://se.uom.gr/

Software and Data Engineering Lab  http://sde.uom.gr/

Department of Applied Informatics  http://mai.uom.gr/

University of Macedonia  http://www.uom.gr/
Our team

Apostolos Ampatzoglou
Elvira Arvanitou
Theodoros Amanatidis
Panagiotis Kouros
Areti Ampatzoglou
Prof. Paris Avgeriou
Alexander Chatzigeorgiou
Nikolaos Mittas
George Digkas
Dionisis Kehagias
Theodoros Chaikalis
Dimitris Tsoukalas
Prof. Lefteris Angelis
Our expertise

Code/Design smell identification & Refactoring Application

Design Pattern Detection

Software Evolution Analysis

Software Quality Assessment & Metrics

Technical Debt

Tool Development
Design Pattern Detector with Similarity Scoring

JDeodorant

Percerons

JCaliper

SEAgle

SEAgle

Easy Software Evolution Analysis

Change Proneness Estimator

Technical Debt Interest Calculator
A brief history on software quality

1960’s
“Code and Ship”

1970’s
Emergence of SQA

1977
McCall Quality Model

1991
ISO/IEC 9126

2000’s
Legislation Standards (ISO, IEEE, CMM)
SQA certification


N. Fenton
1991
GoF 1994

1978
ISO/IEC

1991
Debt, Cunningham

1999
1st MTD workshop

2002
Bansiya’s QMOOD model

2007-08
Sonar Platform

2010
1st SQALE Method inspearit

2011
ISO/IEC 25010

2012
1st Dagstuhl Seminar on TD

2016
1st TechDebt Conference

2018
Sonar Platform

Quality Gates

CAST software

2004
Cost of Poor Quality (J. Harrington, IBM, 1987): costs that would disappear if systems, processes and products were perfect

Harrington adopted the name to emphasize the belief that investment in detection and prevention of product failures is more than offset by the savings in reductions in product failures.
"Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite... The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt"—Ward Cunningham, 1992
In software-intensive systems, technical debt is a collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible. Technical debt presents an actual or contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability.
Maintenance Effort vs. Fitness Function Value

- **Optimum**: Represents the ideal or best possible effort level.
- **Actual**: Represents the actual effort level.
- **Optimum’**: Represents an updated or new optimum effort level.
- **Actual’**: Represents an updated or new actual effort level.

- **Effort\_m(\text{optimum})**: Maintenance effort at the optimum level.
- **Effort\_m(\text{actual})**: Maintenance effort at the actual level.

**TD principal**

- **Effort\_r**: Refactoring effort.
- **interest**: Interest or additional effort.

**Feature A**

- Highlighted point or area indicating a specific feature or aspect.

**Principal and Interest**

- Refers to the relationship between maintenance effort and fitness function value, emphasizing the principal and interest aspects.

The diagram illustrates the relationship between maintenance effort and fitness function value, with a focus on optimizing effort levels and the impact of refactoring or additional effort.
Project backlogs often contain the green elements; rarely the yellow ones.

Defects often reside in other systems; Black items are nowhere to be found.

Rios et al. A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners, IST 2018
Code TD is the **poorly written code** that violates best coding practices or coding rules. Examples include code duplication and over-complex code.

**Code duplication** was found to be the *most frequent type of TD* in 66 projects of the Apache ecosystem (Digkas et al. ECSA’2017).

*Source code TD* has the **strongest association** on the amount of the wasted time (Besker et al. TechDebt 2018).

SonarSource introduced the “**Cognitive Complexity**” metric to reflect on the increased interest on measuring complexity that provides a "fair" representation of maintainability.
People are skeptical about the possibility of actually improving quality by removing for example “Pieces of code with large size, high complexity and low cohesion”

```java
int i;
int sum = 0;
int product = 1;
for(i = 0; i < N; ++i) {
    sum = sum + i;
    product = product * i;
}
System.out.println(sum);
System.out.println(product);
```
Architectural technical debt (ATD) is incurred by design decisions that intentionally or unintentionally compromise system-wide quality attributes, particularly maintainability and evolvability.

- Difficult to identify and quantify;
- ATD does not always receive the full attention from the architect and management teams;
- Impact is generally larger than code TD
Architectural drift occurs when the implementation of a program diverges from the initial design and purpose.
• depending on 3rd-party code requires that projects keep current with the latest version of each component.
• when projects do not stay current, they begin to incur a form of technical debt where API calls that have been deprecated remain in the code base

Snipes and Ramaswamy, A Proposed Sizing Model for Managing 3rd Party Code Technical Debt, TechDebt’ 2018
TD Activities

TD identification detects TD caused by intentional or unintentional technical decisions in a software system through specific techniques, such as static code analysis.
**TD measurement** quantifies the benefit and cost of known TD in a software system through estimation techniques, or estimates the level of the overall TD in a system.
TD prioritization ranks identified TD according to certain predefined rules to support deciding which TD items should be repaid first and which TD items can be tolerated until later releases.

https://refactoring.guru/
TD repayment resolves or mitigates TD in a software system by techniques such as reengineering and refactoring

https://refactoring.guru/
TD prevention aims to prevent TD from being incurred.
TD monitoring watches the changes of the cost and benefit of unresolved TD over time.
TD representation / communication

TD representation provides a way to represent and codify TD. TD communication makes identified TD visible to stakeholders.
Software developers report that **they waste on average 23%** of their working time due to TD.
if (TD == code smells)
TD removal = refactorings

Long Method
Feature Envy
Type Checking
Even so, do people perform refactoring?

**JFreeChart**

**Long Method**

~79% extend up to the latest version

~60% exist from the beginning → design problems are also a consequence of inefficient OOAD

-7.24% explicit smell removal

- only three cases of unambiguous

**Extract Method**
Even so, do people perform refactoring?

32% of 85 respondents stated that they *did not know about code smells* (Yamashita and Moonen, 2013)

**Why developers are scared to refactor code?**

- Lack of Confidence! They are not confident about their changes
- If it doesn’t work, what amount of existing code will it hamper?
- They are not sure about Regression testing

Interaction data from about 1268 hours of programming suggest that *programmers are reluctant to use automated refactorings* whose outcomes are difficult to foresee

Vakilian et al., *Use, Disuse, and Misuse of Automated Refactorings*, ICSE’2012
Then, what happens in some projects?

Is the drop due to deleted files, modified files, or new files with TD-clean code?
Self-Admitted Technical Debt

```c
/* This is a hack until we finish the code so that it only reads
 * the config file once and just operates on the tree already in
 * memory. rbb
 */
ap_conftree = NULL;
apr_pool_create(&ptemp, pconf);
apr_pool_tag(ptemp, "ptemp");
```

```java
/**
 * Tries to extract a useful string for comparison from the provided object.
 * This method is a workaround for bug 226547. Looking forward we need a
 * more sensible answer to this problem.
 *
 * @param o
 *   the object to test
 * @return the comparison string
 * @return the comparison string
 * TODO: remove this method and replace it with a sensible solution
 */
private String getComparisonString(Object o) {
    if (o instanceof IPreferenceNode) {
        return ((IPreferenceNode)o).getLabelText();
    }
    return o.toString();
}
```

Maldonado and Shihab, Detecting and Quantifying Different Types of Self-Admitted Technical Debt, MTD’2015
Self-Admitted Technical Debt

People write all sorts of funny (but truthful) comments!!

```java
public getRandom() {
    //chosen by a roll of dice
    return 12;
}
```

```java
#define TRUE FALSE
//Just kidding
```

```java
try {
} finally {
    //should never happen
}
```

```java
catch (Exception e) {
    //who cares??
}
```

//When I wrote this, only God and I understood what I was doing
//Now, God only knows

## Self-Admitted Technical Debt

<table>
<thead>
<tr>
<th>Project</th>
<th># of analyzed comments</th>
<th># of self-admitted TD comments</th>
<th>% of self-admitted TD per project</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ant</td>
<td>4,140</td>
<td>134</td>
<td>3.2</td>
</tr>
<tr>
<td>ArgoUML</td>
<td>9,788</td>
<td>1,653</td>
<td>16.8</td>
</tr>
<tr>
<td>Columba</td>
<td>6,569</td>
<td>295</td>
<td>4.4</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>4,433</td>
<td>219</td>
<td>4.9</td>
</tr>
<tr>
<td>Jmeter</td>
<td>8,163</td>
<td>375</td>
<td>4.6</td>
</tr>
</tbody>
</table>

Maldonado and Shihab, Detecting and Quantifying Different Types of Self-Admitted Technical Debt, MTD’2015
Self-Admitted Technical Debt

Bavota and Russo, A Large-Scale Empirical Study on Self-Admitted Technical Debt, MSR’2016
Landscape of Empirical Evidence on TD (perspective: secondary studies on TD)

- Research is still highly concentrated on few types of debt.
- Some researchers tend to put anything that is detrimental to the software product and development process under the umbrella of TD.
- Treatment of Architectural TD is still inconsistent and creates confusion.
- Lack of tools dealing with artifacts other than source code.
- Little evidence on how to prioritize TD items.
- TDM tools should be integrated into the work making TDM as part of daily activity.
- Traceability between TD and its related artifacts should be maintainable.
- Software visualization to support the TD management needs further investigation.
- Despite the availability of several TD indicators, there is no evidence on how to use them for TD identification in real settings.

Rios et al. A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners, IST 2018
Software Development ToolKit for Energy Optimization and Technical Debt Elimination

http://sdk4ed.eu/
Enhancing Programmability and boosting Performance Portability for Exascale Computing Systems

http://exa2pro.eu/

Work reported in this presentation has received funding from the European Union's Horizon 2020 Research and Innovation programme under grant agreement No. 801015 (project EXA2PRO)
Holy Grail of quality assessment tools

11k smells!!
Distribution of TD among developers

Gini index = 0.66

Gini index = 0.65

Gini index = 0.61
Factors affecting developers decision to repay TD

developers appear to be **largely influenced by the severity** of an issue (Info, Minor, Major, and Critical)

“**String literals should not be duplicated**

```javascript
function run() {
    prepare('action1');  // Non-Compliant
    execute('action1');
    release('action1');
}
```

“**Comments should not be located at the end of lines of code**”.

```javascript
$a = $b + $c; // This is a trailing comment that is non-compliant
```
The **broader characterization** of the TD issue seems to affect developers

```
if ((($condition1 && $condition2) || ($condition3 && $condition4)) {
    ...
}
```

“Expressions should not be too complex”

```
/*
 * function power($a, $b=2) {
 *     if ($b == 2) {
 *         return $a * $a;
 *     }
 *     $value = 1;
 *     for ($i = 0; $i < $b; $i++) {
 *         $value *= $a;
 *     }
 *     return $value;
 * }
 */
```

“Sections of code should not be commented out”

Testability, Changeability and Maintainability are more ‘real’ compared to security/reliability
Developers do not tend to accept suggestions for revising their own code.

Developers’ decisions appear to be unaffected by:

- the time required to fix an issue
- the total TD in the examined file
- the frequency of modifications to the file under study
Estimating the Breaking Point of TD

\[ \text{Time (Versions)} \]

\[ \text{Effort} \rightarrow \text{Cost} \]

\[ \text{Principal}(t) \]

\[ \text{Cumulative Interest} \]

\[ \text{Breaking point} \]

\[ \text{versions} = \frac{\text{Principal}(\$)}{\text{Interest}(\$)} \]
Estimating the Breaking Point of TD

**Principal** = Effort<sub>r</sub>

Interest = \( \Delta \text{Effort}_m \)

Optimum system is unknown! Optimum Effort is unknown!

Assumptions:
- Fitness Function:
  Maintainability model of ten metrics
- Effort is analogous to Fitness function value

\[
\frac{\text{Effort}_m(\text{optimum})}{\text{Effort}_m(\text{actual})} = \frac{c \cdot \text{Fitness Value}(\text{optimum})}{c \cdot \text{Fitness Value}(\text{actual})} \Rightarrow \\
\text{Effort}_m(\text{optimum}) = \frac{\text{Fitness Value}(\text{optimum})}{\text{Fitness Value}(\text{actual})} \cdot \text{Effort}_m(\text{actual})
\]
1. For each class of the system find structurally similar classes (in terms of size, complexity, NoM, Sqale Index, etc)

2. From the similar classes, we identify the ones that have the best metric values

3. Average distance between class under analysis and optimum classes is obtained -> denotes the optimum design for this class

Projecting past maintenance effort allows us to estimate interest
We have published 12 papers related to JDeodorant features.

- 15,000+ installations
- 35,000 refactorings applied
- 1000+ citations

Can tools really help?

#228 among all Eclipse plugins
Impact

~600 .com users

Refactoring Assistants Session
Software Reengineering M.Sc. course

Software Engineering
Undergraduate course projects
50% correspond to Extract Method
25% Extract Class
16% Move Method
Lessons learned

Tools should target both Practitioners & Researchers

The design of a tool should be guided by empirical evidence about developers’ practices, so that the tool is tailored to the actual needs of the developers.

Tool usage statistics should be recorded as early as possible, to allow focusing on improvements targeting the most popular tool features.
Lessons learned

A tool should be **offered as an IDE plug-in** instead of a standalone application.

A tool should require **minimum installation** and configuration effort.

A tool should be **tested in an industrial setting**

A tool **should be open-sourced** as early as possible, even if it is not as mature and stable.

A tool should be accompanied with **documentation**, tutorials, and code snippets demonstrating the use of its API.

---

to assist:

- researchers
- practitioners
- educators

- bug reports
- external contributions (30 forks)
Thank you very much!

Хвала!

The only valid measurement of code quality: WTFs/minute

Good code.

BAD code.

(c) 2008 Focus Shift
Smell Detectors

- **MethodBook** [Bavota et al. TSE’14]
  developers are more willing to perform refactorings based on *structural and textual* information

- **HIST** [Palomba et al. ASE’13, TSE’15]
  employs *change history* information
  Precision: 72-86%, Recall: 58-100%

- **TACO** [Palomba et al. ICPC’16, TSE’17]
  pure *textual analysis* approach
  developers tend to ignore structurally detected smells

- **JMove** [Sales et al. WCRE’13, JSS’18]
  assumption: methods in well designed classes usually establish dependencies to similar types.
  for Move Method refact. precision: 21-32%, recall: 21-60%

Move Method
• RefactoringNavigator [Lin et al. FSE’16] takes a given implementation, a desired high-level design and recommends a series of ref. steps. JD used by a control group to assess the time and efficiency of refactorings.

• SEMI [Charalampidou et al. TSE’17] determines statements to be extracted based on statement coherence (e.g. if they access the same variable, or call the same method). Best F-measure for large methods (>30 LOC).

• GEMS [Xu et al. ISSRE’17] learning-based approach. GEMS learns a prob. model from training sets.

• ARIES [Bavota et al., ICSE’12] Candidate classes represented as graphs: methods: nodes, edges: method pairs. MaxFlow – MinCut algorithm to split the class.
Milestones in the hunt for smells

- **2009**: Pure Structural + Metrics based
- **2010**: Structural + Semantic
  
  Bavota et al.
- **2011**: Change History based
  
  Palomba et al.
- **2012**: Interactive Genetic algorithm
  
  Bavota et al.
- **2013**: Game Theory based
  
  Bavota et al.
- **2014**: Pure Textual
  
  Palomba et al.
- **2015**: Feedback based
  
  Liu et al.
- **2016**: Learning based
  
  Xu et al.