Detecting and Evaluating Technical Debt of Software Systems and System of Systems

Qiao Zhang
Advisor: LiGuo Huang

Dept. of Computer Science and Engineering
Southern Methodist University
{qiaoz, lghuang}@smu.edu

SERC Doctoral Student Forum
What is Technical Debt (TD)?

**Technical Debt:** Delayed technical work or rework that is incurred when shortcuts are taken or short-term needs are given precedence over the long-term objectives. It is the result of intentional or unintentional decisions that impact the viability of a system and usually incur interests (i.e., additional cost) to eliminate.

- *Process/Methods*
- *Schedule*
- *Capabilities*
- *Budget*
- *Documentation*
- ...

@SMU-CSE
World Record for Producing Technical Debt

The Health Insurance Marketplace is Open!

Enroll now in a plan that covers essential benefits, pre-existing conditions, and more.

Plus, see if you qualify for lower costs.

APPLY NOW

WANT TO LEARN MORE FIRST? START HERE

Get covered: A one-page guide
Find the Marketplace in your state
Get lower costs on health insurance
See what Marketplace insurance covers
Healthcare.gov System
Development Problem Analysis

- **Size and Complexity:** more than 5000 pages in the Affordable Care Act and more than 78,000 qualified health plans across the states
  - Concurrent multidisciplinary engineering: engaging 55 individual contractors to perform over $600 million in federally contracted work
  - Ambiguous language and conflicting requirements
  - Schedule was a key driver

- **Development Experience:** Top-level government leadership in the U.S. Dept. of Health and Human Services (HHS) had no significant experience in overseeing a system development effort anywhere close to the size and complexity of Healthcare.gov and therefore little understanding of commitments.
  - Shortfalls in coordination

- **Result:** The White House and HHS leaders were informed in a March 2013 independent assessment from McKinsey & Co., that the “launch was fraught with risks.”
  - All-at-once deployment of the immature systems of systems (SoSs)
  - Proliferation of errors that could have been resolved earlier
  - Key problems documented in the report (unstable requirements, little time for testing, little to no time for fixing problems) were never resolved before the rollout.
The demand for software systems increases by 900% each decade.

Although expenditure on software development increases by 200% each decade, productivity of software system engineers only increases by 35%.

Approximately 75% of the total life cycle cost for a given system is related to system maintenance and operation.

Gartner estimates that the Global “IT Debt” to be $500 billion in 2014, with the potential to grow to $1 trillion by 2015.

CAST Research Lab analyzed 1,400 applications containing 550 million lines of code (LOC) submitted by 160 organizations and estimated that the Technical Debt (TD) of an average-sized application of 300 KLOC is $1,083K, which represents an average TD per LOC of $3.61.
## Major Sources of Technical Debt

### Inconsistent definitions of technical debt!

<table>
<thead>
<tr>
<th>Source</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poor Process capabilities and conformance</td>
<td>4.53</td>
</tr>
<tr>
<td>Artifact-centric (poor, outdated), task oriented (left undone)</td>
<td>4.85</td>
</tr>
<tr>
<td>Code or application quality sacrificed to deliver to market</td>
<td>6.24</td>
</tr>
<tr>
<td>Code quality problems/poor evolvable code due to inexperience developers</td>
<td>5.44</td>
</tr>
<tr>
<td>Not enough spent on training resources</td>
<td>4.76</td>
</tr>
<tr>
<td>Complexity in design and code evolved</td>
<td>5.97</td>
</tr>
<tr>
<td>Architecture/design shortcuts and issues without refactoring</td>
<td>6.09</td>
</tr>
<tr>
<td>Insufficient application renovation over time</td>
<td>5.12</td>
</tr>
<tr>
<td>Bugs/defects accumulation over time</td>
<td>4.74</td>
</tr>
</tbody>
</table>
## Overlaps between Technical Debt Categories

<table>
<thead>
<tr>
<th>Department</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture Debt</td>
<td>Degraded architecture quality or sub-optimal solutions</td>
</tr>
<tr>
<td>Design Debt</td>
<td>Violation of the principles of good object-oriented design</td>
</tr>
<tr>
<td>Code Debt</td>
<td>Source code that negatively affect the legibility of the code</td>
</tr>
<tr>
<td>Test Automation Debt</td>
<td>Test automation tasks that fail to support continuous integration and faster development cycle</td>
</tr>
<tr>
<td>Maintenance Debt</td>
<td>Delayed certain software maintenance tasks</td>
</tr>
<tr>
<td>Defect Debt</td>
<td>Known latent defects that have not been fixed</td>
</tr>
<tr>
<td>Documentation Debt</td>
<td>Missing, inadequate, or incomplete documentation</td>
</tr>
<tr>
<td>People Debt</td>
<td>People issues that can delay or hinder development activities</td>
</tr>
<tr>
<td></td>
<td>Imperfect software design or architecture that affects future maintenance negatively</td>
</tr>
<tr>
<td></td>
<td>Code smell and design violation of good coding practice that leads frequent and future maintenances activities</td>
</tr>
<tr>
<td></td>
<td>Problematic test automation code that are unable to fulfill the planned testing tasks</td>
</tr>
<tr>
<td></td>
<td>Found defects but not timely fixed during maintenance activities</td>
</tr>
<tr>
<td></td>
<td>Insufficient documentation that impacts the knowledge distribution of personnel alternation</td>
</tr>
</tbody>
</table>
### A Systematic Framework for Technical Debt Measurement

<table>
<thead>
<tr>
<th>Sources and Root Causes</th>
<th>Categories</th>
<th>Principal Metrics</th>
<th>Interest Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Degraded architecture quality or sub-optimal solutions</td>
<td>Architecture Debt</td>
<td>New element and pre-existing element cost</td>
<td>Distance between estimated and actual rework cost</td>
</tr>
<tr>
<td>Violation of the principles of good object-oriented design</td>
<td>Design Debt</td>
<td>WMC, TCC, ATFD ...</td>
<td>Maintenance effort from present to higher quality level</td>
</tr>
<tr>
<td>Delayed certain software maintenance tasks</td>
<td>Maintenance Debt</td>
<td>RF, RV, and RA</td>
<td></td>
</tr>
<tr>
<td>Source code that negatively affect the legibility of the code</td>
<td>Code Debt</td>
<td>Violation of coding standard</td>
<td></td>
</tr>
<tr>
<td>Distance between optimal requirements and actual solution</td>
<td>Requirement Debt</td>
<td>Different set of specification</td>
<td>Increasing rate of the distance</td>
</tr>
</tbody>
</table>
*ABC* is a mobile app as a client of server system *E*, which is developed by team *M*. In the previous releases *(R₁-Rₙ)*, all features were built upon the protocol *A* for communication with *E*. During preparation of release *Rₙ₊₁*, team *M* declares that the protocol *A* needs to be replaced with protocol *B* since it is no longer supported by the new server system *E’*. As a result, app *ABC* may need to support both protocol *A* and *B* in the near future.

- **TD Principal (P):** $\forall M_k \in R_n, P = \sum_k D(M_k, A) \times C_r(M_k)$
- **TD Interest Amount:** $\forall M_j \in R_{n+1}, P = \sum_j D(M_j, A) \times C_i(M_j)$
- **TD Interest Probability:** probability that system *E* is upgraded to *E’* by customers

- $M_k$: module *k*.
- $D(M_k, A)$: dependency between module *k* and protocol *A*.
- $C_r(M_k)$: rework cost of module *k*.
- $C_i(M_j)$: development cost of new feature module *j*.

---


---

**What is TD? Can TD be Measured?**

**An Exploratory Example**
Gap Analysis: Mapping between Technical Debt and COCOMO Cost Drivers

State of the Art

- Categorizing Technical Debt (TD)
- Defining TD Categories
- Analyzing Metrics (or Indicators) of TD Categories
- Exploring Measurement Methods of TD Categories

Objective

- Definition of Cost Drivers
- Rating Guidelines of Cost Drivers

- Relationships between Technical Debt (TD) Categories and Cost Drivers
Qualitative Analysis: Top-level Mapping between TD Categories & COCOMO II Cost Drivers
Example 1:
Code Debt & COCOMO II Cost Drivers

Definitions
- Code duplication, complexity, style that impacts readability [47, 51]
- Problems in source code that negatively impacts legibility and maintainability [49]

Indicators
- No. of code violation [56, 80]
- Code-structure and developer-behavior comprehension metrics in class level [91]
- Tagged comments that admits problematic works [77]

COCOMO Cost Drivers
- RELY
- RUSE
- PCAP
- PCON
- LTEX
- PREC
- SITE

Potential Effect
- Extra efforts due to refactoring existing problematic code

Example 2: Architecture Debt & COCOMO II Cost Drivers

Definitions
- Degraded architecture quality [50]
- Sub-optimal upfront solution
- Sub-optimal solution due to technologies or patterns superseded [47, 51]

Potential Effect
- Extra efforts due to refactoring existing architecture design
- Extra efforts due to high-level dependencies of problematic module

Architecture Debt

COCOMO Cost Drivers
- RELY
- CPLX
- RUSE
- PVOL
- ACAP
- PCAP
- RESL

Measurement Metrics
- Rework efforts spent on existing modules due to dependencies of new architecture elements [50]

Indicators
- Architectural dependencies [66]

Four Types of Relationships:

- **Cost drivers that directly impact on TD principal and interest**
  - E.g., RESL and architecture debt

- **Cost drivers as indicators of the possibility of introducing TD**
  - E.g., ACAP, PCAP, PCON, APEX, PLEX, LTEX and architecture debt

- **Cost drivers that indicate the importance of TD management**
  - E.g., RELY, RUSE, PVOL and architecture debt

- **Cost drivers that may be extended to incorporate TD consideration in their rating guidelines**
  - E.g., CPLX and architecture debt

As the result, extra efforts will be invested on paying off the TD items and usually performed as various refactoring operations.
The percentage of design changes due to the refactoring operations on TD items

\[ AAF = (0.4 \times DM) + (0.3 \times CM) + (0.3 \times IM) \]

The percentage of LOC changes due to the refactoring operations on TD items

\[ PM_M = A \times Size_M^E \times \prod_{i=1}^{15} EM_i \]
Technical Debt and Refactoring Operations

**TD Items**

- God Class
- Data Class
- Duplicate Code

**Refactoring Operations**

- Extract Class
- Extract Subclass
- Encapsulate Field
- Remove Setting
- Hide Method
- Extract Method
- Pull Up Field
- Pull Up Method
Refactoring God Class: Sizing Change for Extracting Class Operation

- **Baseline**: Extract from a source class $c_s$ with a set of attributes $\{a_k\}$ (where $n = |\{a_k\}|$) and a set of methods $\{m_h\}$, to a target class $c_t$. While $l = 6$ when program language is JAVA.

$$
LOC_p(c_s) = LOC_b(c_s) - (nl - n + \sum_h LOC(m_h)) 
$$

$$
LOC_p(c_t) = nl + \sum_h LOC(m_h) + p 
$$

- **Objective**: Investigating additional factors that impact the final change of software size in order to improve the prediction accuracy

@SMU-CSE
Investigation of Additional Factors for Extract Class/Subclass Operation

LOC added in target class but not included in the baseline prediction

LOC added in source class but not included in the baseline prediction

LOC deleted in source class but not included in the baseline prediction

LOC added in source class due to the dependencies on target class
Example: Comparing LOC Deleted in Source Class with Target Class

<table>
<thead>
<tr>
<th>Base Text</th>
<th>New Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>/*</td>
<td>*/</td>
</tr>
<tr>
<td>* Licensed to the Apache Software Foundation (ASF) under one or more contributor license agreements. The ASF licenses this file to You under the Apache License, Version 2.0 (the &quot;License&quot;); you may not use this file except in compliance with the License. You may obtain a copy of the License at <a href="http://www.apache.org/licenses/LICENSE-2.0">http://www.apache.org/licenses/LICENSE-2.0</a> * Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an &quot;AS IS&quot; BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. * See the License for the specific language governing permissions and limitations under the License. For additional information regarding copyright in this work, please see the NOTICE file in the top level directory of this distribution. */</td>
<td></td>
</tr>
</tbody>
</table>
Example (Cont.): Additional Size Change Due to Refactoring Operation

```java
@Override
protected ResponseContext process(
    Provider provider,
    RequestContext request) {
    TargetType type = request.getTarget().getType();
    TypeHandler handler = getHandler(type);
    if (handler == null) {
        String method = request.getMethod();
        handler = getMethodHandler(type, method);
        if (handler == null) {
            return super.process(provider, request);
        } else {
            return handler.process(provider, request);
        }
    } else {
    return handler.process(provider, request);
    }
}
```
TD Items

Refactoring Operations

Mapping Results

Prediction Model

Evaluation

- Systematic Literature Review on Software Refactoring and Measurement
- Systematic Mapping Study on TD items and Refactoring Operations
- Replicate Baseline Sizing Change Prediction Functions
- Sizing Change Prediction
- Evaluate the Accuracy of Sizing Change Prediction with Commit History
Conclusions

• Technical Debt (TD) is a growing concern for software systems.

• The effect of TD on development/maintenance effort is promising to be estimated with COCOMO.

• Gap analysis indicates that there are four types of relationships between TD categories and COCOMO II cost drivers.

• Predicting sizing change due to TD item refactoring can be a bridge to link TD measurement with the reuse and maintenance models in COCOMO.
Thank you!

Qiao Zhang
Advisor: LiGuo Huang

Dept. of Computer Science and Engineering
Southern Methodist University
{qiaoz, lghuang}@smu.edu

SERC Doctoral Student Forum

December 2, 2015