Technical Debt

Dr. Mark C. Paulk

SE 4367 – Software Testing, Verification, Validation, and Quality Assurance

Technical Debt

Ward Cunningham first drew the comparison between technical complexity and debt in 1992.

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.

Technical Debt Definitions

Should-fix violations are violations of good architectural or coding practice known to have an unacceptable probability of contributing to severe operational problems or to high costs of ownership.

Principal is the cost of remediating should-fix violations in production code.

<u>Interest</u> is the continuing costs attributable to should-fix violations in production code that haven’t been remediated, such as greater maintenance hours and inefficient resource usage.

<u>Technical debt</u> is the future costs attributable to known violations in production code that should be fixed – a cost that includes both principal and interest.

Extending the Metaphor

Steve McConnell: technical debt includes both intentional and unintentional violations of good architectural and coding practice.

Expands Cunningham's original focus on intentional decisions to release suboptimal code to achieve objectives such as faster delivery.

Monetizing Technical Debt

Once you monetize technical debt, any stakeholder can appreciate its operational, financial, and business implications.

I. Gat, "Technical Debt as a Meaningful Metaphor for Code Quality," IEEE Software, November/December 2012.

A good debt is one we carefully use to evolve our products, make money, grow our business, and pay our debt back.

A bad debt includes expenses that are wasteful and doesn't include plans for how to pay it back within a reasonable time.

C. Ebert, "A Useful Metaphor for Risk – Poorly Practiced," IEEE Software, November/December 2012.

The Trade-Off

Most definitions of technical debt come down to a trade-off between quality, time, and cost.

- **intentional decisions to trade off competing concerns during development**
- **negative effects tend to be longer term**
  - **increased complexity, poor performance, low maintainability, and fragile code**

As long as a project properly manages technical debt, it can achieve selected goals sooner than would have otherwise been possible

E. Lim, N. Taksande, and C. Seaman, "A Balancing Act: What Software Practitioners Have to Say About Technical Debt," IEEE Software, November/December 2012.

Tackling Technical Debt

Awareness: identifying debt and its causes

Manage debt-related tasks explicitly, in a backlog

- **tasks to attend to in the future to increase value, such as adding new features**
- **investing in the architecture**
- **reduce the negative effects on value of defects**
- **reduce the negative effects on value of technical debt**

Estimating Technical Debt

A function of three variables

- **number of should-fix violations in an application**
- **hours to fix each violation**
- **cost of labor**

B. Curtis, J. Sappidi, and A. Szynkarski, "Estimating the Principal of an Application's Technical Debt," IEEE Software, November/December 2012.

Three estimates with different parameters

- **high, medium, and low severity violations**
- **US \$70- \$80 per hour**
- **700 applications, 357 MLOC, all larger than 10KLOC, many different industries (bias toward business-critical applications)**

Tool Support

CAST's Application Intelligence Platform (AIP)

Uses more than 1200 rules to detect violations of good architectural and coding practice

The Top Violations in C++

Violation	Frequency
Avoid undocumented functions, methods, constructors, destructors	267,861
Avoid data members that are not private	182,076
Avoid unreferenced methods	73,888
Avoid using global variables	47,834
Avoid artifacts with high internal complexity	18,065

The Top Violations in C

Violation	Frequency
Avoid undocumented functions	**56,027**
Avoid artifacts with high internal complexity	**32,943**
Avoid functions with SQL statement including subqueries	**30,153**
Never use strcpy() function — use strncpy()	**29,332**
Never use sprintf() function or vsprintf() function	**21,608**

The Top Violations in Java EE

Violation	Frequency
Avoid methods missing JavaDoc comments	4,028,727
Avoid methods missing appropriate JavaDoc @param tags	3,227,014
Avoid methods missing appropriate JavaDoc @return tags	3,018,182
Avoid private fields missing JavaDoc Comments	1,737,620
Avoid using fields (nonstatic final) from other classes	556,046

The Top Violations in Visual Basic

Violation	Frequency
Avoid undocumented functions and methods	45,680
Avoid using global variables	32,258
Avoid unreferenced functions and methods	23,675
Avoid direct usage of database tables	12,885
Avoid artifacts with high internal complexity	7,143

The Top Violations in .NET

Violation	Frequency
Avoid uncommented methods	203,651
Avoid declaring public class fields	152,972
Avoid artifacts with high fan-out	84,580
Avoid classes with a high lack of cohesion	56,486
Avoid instantiations inside loops	16,309

Themes for Violations

The number of violations per application is large for all applications.

Frequent problems include

- **undocumented modules / classes / functions**
- **use of global variables (public attributes)**
- **modules with high fan-out (high coupling)**
- **modules with low cohesion**
- **modules with high internal complexity**
- **unreferenced methods (dead code)**
- **security vulnerabilities, such as buffer overflow and SQL injection**

The Impact of Technical Debt

A conservative estimate of the technical debt principal for the average application is $361,000 for each 100 KLOC.

- **$3.61 per line of code**

More realistic estimates are likely to be dismissed as excessive.

Questions and Answers