2012 12th International Conference on Quality Software

Strategic Management of Technical Debt

Tutorial Synopsis

Philippe Kruchten
Electrical and Computer Engineering
University of British Columbia
Vancouver, BC, Canada
pbk @ ece.ubc.ca

***Abstract*—The technical debt metaphor acknowledges that software development teams sometimes accept compromises in a system in one dimension (for example, modularity) to meet an urgent demand in some other dimension (for example, a deadline), and that such compromises incur a "debt". If not properly managed the interest on this debt may continue to accrue, severely hampering system stability and quality and impacting the team's ability to deliver enhancements at a pace that satisfies business needs. Although unmanaged debt can have disastrous results, strategically managed debt can help businesses and organizations take advantage of time-sensitive opportunities, fulfill market needs and acquire stakeholder feedback. Because architecture has such leverage within the overall development life cycle, strategic management of architectural debt is of primary importance. Some aspects of technical debt—but not all technical debt—affects product quality. This tutorial introduces the technical debt metaphor, the various types of technical debt, and in particular structural or architectural debt, the techniques for measuring and communicating this technical debt, and its relationship with software quality, both internal and external quality.**

Keywords- technical debt; software evolution; software architecture

I. THE CONCEPT OF TECHNICAL DEBT

Ward Cunningham coined the term *technical debt* in his 1992 OOPSLA experience report [1] where he described a situation in which long-term code quality was traded for short-term gain, creating future pressure to remediate the expedient. He wrote:

> *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. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise.*

To this day, technical debt has been used as a metaphor and rhetorical device, primarily within the agile software community, to communicate between the technical and business community, between engineers and executive. The concept resonates well with both parties, and allows technical staff to attract attention on the long-term management of the accidental complexities created by short-term compromises [2].

The metaphor draws a parallel between financial debt, which incurs interest payments, and technical debt where interests take the form of future higher development cost to further evolve and maintain the software. Like financial debt, you can continue paying interest, or repay the 'principal' by restructuring the code and reduce or suppress future interest payments.

Steve McConnell was the first to introduce a taxonomy of technical debt [3]:

I. Debt incurred unintentionally due to low quality work
II. Debt incurred intentionally
- *II.A. Short-term debt, usually incurred reactively, for tactical reasons*
  - *II.A.1. Individually identifiable shortcuts (like a car loan)*
  - *II.A.2. Numerous tiny shortcuts (like credit card debt)*
- *II.B. Long-term debt, usually incurred proactively, for strategic reasons*

The Type I debt is typically due to lots of small imperfections in the code, often called nowadays "code smells" [4]: naming convention violations, numerous code clones, for example. This technical debt can be detected and evaluated using static analysis tools, such as Sonar [5].

Type II and in particular Type II.B are forms of technical debt usually encountered at the architectural level. It is intentional, deliberate and prudent (to use the terms of Martin Fowler [6]): the result of decisions made to overcome some difficulty, try a market, wait for some partner's release, etc. Sometimes it is the result of a shift in the market, something was not technical debt now becomes a debt.

In this tutorial we will mostly deal with McConnell's Type II.B technical debt, which we call *structural technical debt* or just *structural debt*.

Other forms of technical debt have been identified: in particular test debt: the absence of systematic means to do regression testing is dragging down many software development organizations who cannot safely release code

1550-6002/12 $26.00 © 2012 IEEE
DOI 10.1109/QSIC.2012.17

without incurring large testing expenses. Similarly we can speak about documentation debt or design debt, or even requirements debt.

Although unmanaged debt can have disastrous results, strategically managed debt can help businesses and organizations take advantage of time-sensitive opportunities, fulfill market needs and acquire stakeholder feedback. Because architecture has such leverage within the overall development life cycle, strategic management of architectural debt is of primary importance.

II. Outline of the Tutorial

The QSIC tutorial on technical debt has the following structure:

1. Introduction to Technical Debt (30 minutes)
   a. Definitions of technical debt
   b. Examples of various types of debt
2. Making Hard Choices about Technical Debt (60 minutes)
   a. Learning game to illustrate concepts
   b. Debrief: lessons learned form the game
3. Practical measures (60 minutes)
   a. Examples, results of interviews from industry
   b. Tools and techniques
   c. Requirements for support (tool or otherwise), metrics and measures of success
4. Future Directions (25 minutes)
   a. Agile architecting and technical debt
   b. Vision for technical debt analysis framework
   c. Discussion on key problems and challenges faced by practicing software engineers who need to elicit, communicate, and manage technical debt at different facets of their projects
5. Resources (5 minutes)
   a. Tools, literature, activities on technical debt

III. Key Take-aways

Participants will play the *Hard Choices game*, an engaging technique for communicating the trade-offs of technical debt management to fellow colleagues and managers [7, 8]. In your quest to release a quality product, you must decide whether to take shortcuts and incur penalties later, or to traverse more of the board to potentially earn more immediate value. We then use the game as a basis to discuss practices that can be applied to mitigate the impacts of various classes of technical debt in software development. The tutorial concludes by raising awareness of efforts in the research community to move beyond the original metaphor to provide a foundation for managing trade-offs based on models of their economic impacts.

We will provide a detailed analysis of structural (or architectural) technical debt:

- The causes, mechanisms, processes leading to this form of technical debt
- The properties of technical debt: visibility, value, impact.

While the concept of technical debt has been made more visible lately by the "agile" community, rapid iterations and a focused on delivering value early to the customer has made certain agile projects more susceptible to technical debt. These projects are also in danger of not paying enough attention to software architecture, (cast off as "BDUF" Big Design Up-Front, or YAGNI You Ain't Gonna Need It), and trusting their ability to refactor any deficiency in the code at anytime.

We will introduce various techniques and tools to assess technical debt, or to assist in strategic choices about incurring some technical debt, or repayment of technical debt [9].

- Static analysis: tools like Sonar [5]
- Maintainability index: tools like the one offered by SIGS [10] in the Netherlands
- Design Structure Matrices (DSM), and tools such as Lattix® [11]
- Use of the financial concept of Real Options
- Indices of complexity, such as propagation cost [12], etc.

We illustrate this part by looking at a new metric derived from propagation cost [12], called the *Change Propagation* metric [13, 14], and its application to a couple of case studies.

IV. Technical Debt and Quality

The main impact of technical on software quality is *maintainability*, especially the last 3 items in its ISO 9126 definition, if we speak about structural debt:

"Maintainability is the ease with which a product can be maintained in order to:

1. isolate defects or their cause
2. correct defects or their cause
3. *meet new requirements*
4. *make future maintenance easier, or*
5. *cope with a changed environmen*t [15]."

When we look at technical debt from the perspective of maintainability we realize that "technical debt" or "structural debt" may be just new fancy names on a rather old concept. Technical debt is represented in Lehmann 2nd law of software evolution: *Increasing Complexity: As a program is evolved its complexity increases unless work is done to maintain or reduce it* [16], and its remedies are well described in David Parnas's description of software aging [17], in particular his "software geriatrics": stop the deterioration, incremental modularization, amputation, major surgery (restructuring).

Technical debt is not simply about defects, visible to the users; it is about intrinsic, internal code and architecture quality, which is the reason it is often invisible until very late in the lifecycle, or only when we start evolving the software, going from release 1 to release 2,

and this explains also why it is so hard to convince managers to reimburse the debt: customers do not see it or only indirectly through slower evolution.

Another connection of technical debt with quality is through "testing debt": the lack of systematic or automatic ways to do regression testing and unit testing, which the Cutter consortium consultants found to be the leading type of technical debt [9].

V. BIOGRAPHY OF THE SPEAKER

Philippe Kruchten is a full professor of software engineering in the department of electrical and computer engineering of the University of British Columbia, in Vancouver, Canada. He holds an NSERC Chair in Design Engineering.

He joined UBC in 2004 after a 30+ year career in industry, where he worked mostly in with large software-intensive systems design, in the domains of telecommunication, defense, aerospace and transportation. Some of his experience is embodied in the Rational Unified Process® (RUP) ® [18] whose development he directed from 1995 till 2003, when Rational Software was bought by IBM. His current research interests still reside mostly with software architecture, and in particular architectural decisions and the decision process, as well as software engineering processes, in particular the application of agile processes in large and globally distributed teams [19]. He teaches courses in entrepreneurship, software project management, and design.

He is a senior member of IEEE (Computer Society), an IEEE Certified Software Development Professional (CSDP), a member of ACM, INCOSE, CEEA, the founder of Agile Vancouver, and a Professional Engineer in British Columbia.

He has a diploma in mechanical engineering from Ecole Centrale de Lyon (France), and a doctorate degree in information systems from Ecole Nationale Supérieure des Télécommunications in Paris.

ACKNOWLEDGMENTS

This tutorial was prepared in collaboration with Robert L. Nord and Ipek Ozakaya of the Software Engineering Institute (SEI), at Carnegie Mellon University in Pittsburgh, PA, USA. The Hard Choice game was developed also with Nanette Brown (SEI) and Erin Lim (UBC).

REFERENCES

[1] W. Cunningham, "The WyCash Portfolio Management System " in *OOPSLA'92*. Vancouver, 1992. Available on April 9, 2012 from http://c2.com/doc/oopsla92.html

[2] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. B. Seaman, K. Sullivan, and N. Zazworka, "Managing Technical Debt in Software-Intensive Systems," *Proc. of Future of Software Engineering Resarch (FoSER 2010) Workshop, part of FSE 2010*, Santa Fe, New Mexico, USA, 2010, ACM. {doi: 10.1145/1882362.1882373}

[3] S. McConnell, *Technical debt*, 2007. Available on April 9, 2012 from http://blogs.construx.com/blogs/stevemcc/archive /2007/11/01/technical-debt-2.aspx

[4] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code.* Reading, MA: Addison-Wesley, 1999. {ISBN: 978-0201485677}

[5] Sonar. Available on April 9, 2012 from http://www.sonarsource.org

[6] M. Fowler, *Technical Debt*, 2009. Available on April 9, 2012 from http://martinfowler.com/bliki/TechnicalDebt.html

[7] N. Brown, P. Kruchten, E. Lim, I. Ozkaya, and R. Nord, "The Hard Choices Game Explained (whitepaper)." Pittsburgh: Software Engineering Institute, 2010. Available on April 9, 2012 from http://www.sei.cmu.edu/library/abstracts/whitepapers/hard-choices-game-explained-v1-0.cfm

[8] N. Brown, R. Nord, I. Ozkaya, P. Kruchten, and E. Lim, "Hard Choice: A game for balancing strategy for agility," *Proc. of the 24th IEEE CS Conference on Software Engineering Education and Training (CSEE&T 2011),* Honolulu, HI, USA, 2011, IEEE Computer Society. {doi: 10.1109/CSEET.2011.5876149 }

[9] I. Gat (ed.). *How to settle your technical debt--a manager's guide.* Arlington Mass: Cutter Consortium, 2010. Available on April 9, 2012 from http://bookstore.cutter.com/products-page/agile-management/how-to-settle-your-technical-debt-a-managers-guide/

[10] Software Improvement Group (SIG). Available on April 9, 2012 from http://www.sig.eu/en/

[11] Lattix. Available on April 9, 2012 from http://www.lattix.com

[12] A. MacCormack, J. Rusnak, and C. Y. Baldwin, "Exploring the structure of complex software designs: An empirical study of open source and proprietary code," *Management Science*, vol. 52(7), pp. 1015-1030, 2006. {doi: 10.1287/mnsc.1060.0552 }

[13] R. Nord, I. Ozakaya, P. Kruchten, M. Gonzalez, "In Search of a Metric for Managing Architectural Technical Debt," Submitted to WICSA1ECSA 2012 conference, April 2012.

[14] P. Kruchten, M. Gonzalez, I. Ozkaya, R. Nord, N. Kruchten, "Change Propagation Metric as a Measure of Structural Technical Debt," Submitted to WICSA/ECSA conference, April 2012.

[15] ISO/IEC 9126:2004 *Software engineering-- Product Quality*. Geneva: ISO, 2004.

[16] M. M. Lehman, "Laws of software evolution revisited," *Proc. of the 5th European Workshop on Software Process Technology, EWSPT'96, Nancy, Fr,* 1996, LNCS 1149, Springer-Verlag, pp.108-124. {doi: 10.1007/BFb0017737}

[17] D. L. Parnas, "Software aging," *Proc. of the 16th international conference on Software engineering (ICSE 1994*), 1994, IEEE Computer Society, pp. 279-287. {ISBN:0-8186-5855-X }

[18] P. Kruchten, *The Rational Unified Process—An introduction*, 3rd ed, Boston, MA: Addison-Wesley-Longman, 2003. {ISBN: 0321197704}

[19] S. Adolph, P. Kruchten, and W. Hall, "Reconciling Perspectives: A Grounded Theory of How People Manage the Process of Software Development," *Journal of Software and Systems*, 2012 (in press). {doi: 10.1016/j.jss.2012.01.059}