Editor: **Frank Buschmann**
Siemens Corporate Technology
frank.buschmann@siemens.com

To Pay or Not to Pay Technical Debt

Frank Buschmann

WARD CUNNINGHAM COINED the term *technical debt* as a metaphor for the trade-off between writing clean code at higher cost and delayed delivery, and writing messy code cheap and fast at the cost of higher maintenance efforts once it's shipped.[1] Joshua Kerievsky extended the metaphor to architecture and design.[2]

Technical debt is similar to financial debt: it supports quick development at the cost of compound interest to be paid later. The longer we wait to garden our design and code, the larger the amount of interest. Discussions of the metaphor have distinguished different types of technical debt and how and when to best pay them off.[3] Most agree that, sooner or later, technical debt will come due. But is this assumption universally true? If it's better to pay interest, what factors influence the decision to service the debt? And if we decide to retire it, what approach should we take?

Technical debt is similar to financial debt: it supports quick development at the cost of compound interest to be paid later.

Three Stories

Let's begin with three stories from the real world.

Debt Repayment

In one platform project I participated in, component interfaces grew out of control: 350 or more methods per interface were no exception and the platform's core subsystem provided approximately 9,000 methods. Yet, other components used only 15 percent of the methods and up to 50 percent of the methods weren't implemented. This situation resulted from a rule not to modify method signatures after they'd been released; but many methods had been published prematurely and were thus hard to use. Developers coped by adding new methods to component interfaces—again, partly immature—so that over time, interfaces degenerated to method shopping lists rather than being meaningful and economically sound. Development, testing, and maintenance costs increased dramatically and the components were hardly usable. To save the investments into the platform, management initiated a large reengineering activity to clean up the mess. It took one platform release to offer a set of economic interfaces and another to migrate all products to using the new interfaces.

Debt Conversion

One system experienced significant performance penalties because an improper software modularization reflected organizational structures instead of the system's domain. The architects knew about the potential performance threat and intentionally incurred the debt to accelerate development but the severity of its effect surprised them. The interest translated to reduced business and higher performance tuning costs. A long discussion ensued on how and when to remodularize the system appropriately. Finally, the team decided to replace an interpreter component that executed significant parts of the system's functionality with a runtime compiler. This simple measure helped to resolve the performance problems—at the cost of reduced runtime flexibility. The team didn't touch the improper modularization. Metaphorically, the project team

0740-7459/11/$26.00 © 2011 IEEE

decided for a debt conversion. The new debt had lower interest rates and was within the budget.

Interest Payment

We once investigated the viability of reengineering an existing software product that experienced constantly increasing maintenance costs due to architecture erosion from ad hoc modifications. The analysis, however, revealed that the achievable benefits weren't worth the reengineering costs. Likewise for the estimated costs and potential risks of a product rewrite. Management thus decided to do "nothing." The project continued living with the technical debt and paid the associated interest. From a business perspective, this was the cheapest option.

Typically, several options exist that might not prove technically elegant but are "good enough" from a business perspective to alleviate the technical debt.

Where Does It Come from?

Let's face it, technical debt will happen. But where does it come from? The reasons for nontechnical constraints that I hear time and again include "we had tough deadlines to meet" and "we had strict cost targets." Yes, such circumstances can cause technical debt; but too often they're just excuses for a quick-and-dirty design and coding style resulting from other factors.

In fact, many other sources exist for technical debt. Some even fall under the purview of architects! Early installments of this column considered the primary causes of project failure and how to minimize them.[4–6] Although we can argue that the trouble caused by neglecting common development practices isn't a technical debt—because it was taken accidentally—I still find the metaphor useful to discuss the consequences of key design and coding decisions over a system's lifetime.

Yet the metaphor works best when technical debt is intentional. My current project's lead once told me: "We originally planned our architecture toward scalability for geographically distributed systems; this is a new market segment we want to address. But we would have missed the window for our main market of single-server systems if we had realized this concept in the first product release. We thus realized a simpler architecture, which we partly redesign in the next release to serve medium distributed deployments."

Project leaders made a conscious decision to simplify development to meet short-term business goals, and to incur a debt to be retired with the next release through additional effort. Although the final version proved more costly, the project team was in the driver's seat, controlling progress proactively, not reactively, as in the case of accidental technical debt.

Technical Debt Payback?

Regardless of whether the debt is intentional or accidental, the most interesting question is how to deal with it. Most literature concludes that sooner or later, it must be retired.[1–3,7] The three previous tales, however, indicate that this isn't always true. In some situations, it seems appropriate to incur a technical debt and not pay it back. So, how should we act?

Business Rules

Business considerations have priority when it comes to handling technical debt, especially when the debt is intentional. In this situation, it's possible to compare the debt's fiscal benefits with the relevant business drivers, the costs of servicing the debt, and the costs of its retirement—for instance, weighing the business an early product release generates against nonconformance costs due to quick-and-dirty design and code. Business stakeholders should consider these figures and select the best option. Surprisingly—at least in my experience—the calculation reveals that it's often better to incur the debt and pay it off later. It can also prove most profitable for a project to service the debt and not retire it, as in the third story. The good thing about technical debt is that it retires automatically with a system's end of life, unlike a financial debt.

Technical aspects should influence but not drive how to handle a debt. While it's beneficial to pay off technical debt, doing so doesn't always provide the most value.[7] Typically, several options exist that might not prove technically elegant but are "good enough" from a business perspective to alleviate the technical debt. At times, a less optimal solution is also required because a project team or a development organization doesn't have the necessary skills to implement the cleanest solution,[8] as the second story shows. To support choosing the most appropriate solution for handling technical debt, architects must sketch and assess potential alternatives in terms of both benefits and costs.

Handling Technical Debt

When a project team decides to pay back technical debt, it's the architects' responsibility to help choose the appropriate realization. A rule of thumb is to be "minimally invasive"—to keep costs

low, but mostly because there isn't just something to win, there's also something to lose. Most systems that suffer from technical debt are in productive operation and thus have value that shouldn't be destroyed. When tackling technical debt, there's always the danger of doing more harm than good. In the last two installments of this column, we discussed three approaches to architecture gardening: refactoring, reengineering, and rewriting.[9,10] Architects can utilize all three approaches to pay back a system's technical debt. Yet they consider that it might take more than one release until the debt is retired—as our first story demonstrated.

Technical debt is a widely used metaphor to illustrate, discuss, and consider the consequences of design and coding decisions over time. It's a valuable currency to balance a development project's business and organizational factors with its technical aspects to maximize economic success. Pragmatic architects thus approach the subject from a strict business perspective. Only then can they assess when to incur technical debt, whether to retire it, when to retire it, and how. Otherwise, the entire discussion is too biased toward debates on the most elegant design and code, regardless of its impact on a system's business case. And in that case, the metaphor isn't worth a single penny, it's just costly.

References

1. W. Cunningham, "The WyCash Portfolio Management System," *Addendum to Proc. Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA 92), ACM Press, 1992, pp. 29–30, doi:10.1145/157709.157715.
2. J. Kerievsky, *Refactoring to Patterns*, Addison-Wesley, 2004.
3. B. Appleton, "Technical Debt—Definition and Resources," blog, 24 Jun. 2009, http://bradapp.blogspot.com/2009/06/technical-debt-definition-and-resources.html.
4. F. Buschmann, "Learning from Failure, Part 1: Scoping and Requirements Woes," *IEEE Software*, vol. 26, no. 6, 2009, pp. 68–69.
5. F. Buschmann, "Learning from Failure, Part 2: Featuritis, Performitis, and Other Diseases," *IEEE Software*, vol. 27, no. 1, 2010, pp. 10–11.
6. F. Buschmann, "Learning from Failure, Part 3: On Hammers and Nails, and Falling in Love with Technology and Design," *IEEE Software*, vol. 27, no. 2, 2010, pp. 49–51.
7. S. McConnell, *Managing Technical Debt*, white paper, Construx Software, 2008, www.construx.com/Page.aspx?cid=2801.
8. F. Buschmann, "The Pragmatic Architect—Gardening Your Architecture: Part 1," *IEEE Software*, vol. 28, no. 4, pp. 92–94.
9. F. Buschmann, "The Pragmatic Architect – Gardening Your Architecture, Part 2, *IEEE Software*, vol. 28, no. 5, pp. 21–23.

FRANK BUSCHMANN is a senior principal engineer at Siemens Corporate Technology, where he's a principal architect in the System Architecture and Platform Department. Contact him at frank.buschmann@siemens.com.