Editor: **Robert Blumen**
Symphony Commerce
robert@robertblumen.com

Technical Debt

Eberhard Wolff and Sven Johann

IN THIS ISSUE, we deviate from our usual format of one host interviewing one guest. Software Engineering Radio episode 224 features a conversation between our longest-tenured host, Eberhard Wolff, and our newest host, Sven Johann (although Sven was not yet a host when the show was recorded). The episode was inspired by their *InfoQ* article "Managing Technical Debt" (www.infoq.com/articles/managing-technical-debt).

Portions of the episode not featured in this column because of space include sources of technical debt, technical debt as a retrospective quality, how to analyze the costs and benefits of paying down debt, and accepting technical debt on a permanent basis. Besides listening to the entire episode, I also recommend reading the *InfoQ* article for a more detailed discussion. You can download the full episode at www.se-radio.net. —Robert Blumen

Eberhard Wolff (EW): Technical debt is obviously connected to the quality of software. There are actually two types of quality. One type is external quality, which is perceived by a user or customer. That might be the performance, security, scalability, whether the software is stable, and so on. It can be measured and experienced by users. Because it's a feature of the product, it should be managed by the product owners because they are interested in the quality and how the software will be perceived and used by the user.

The more complex part is that there's also internal quality. Internal quality can only be perceived by developers. It's anything that makes extending and maintaining the code easier or harder. That could be things like tests that are there or are missing—if there are more tests, the code is easier to change. Internal quality can be about architectural styles or problems, or it can be about coding issues—whether the code is too complex or too easy.

The hard thing about software development is that internal quality can't really be perceived by anyone except technical people. If you are not a technical guy, it's hard to see what this internal quality really is and how it influences the development process.

Sven Johann (SJ): You could say "technical debt" is a metaphor to describe not-quite-right code. The technical-debt metaphor helps us communicate that if we want to build something on top of not-quite-right code, it will be expensive to do something on this code base later on. So, it takes longer to implement a new feature on a not-so-good code base.

Also, internal quality sooner or later becomes external quality. If we have a bad code base and then get more and more bugs, it will eventually bubble

0740-7459/15/$31.00 © 2015 IEEE

up to the stakeholders of a project. Technical debt is actually not a developer problem; it's a company-wide problem. If you have too much technical debt, in an extreme case, whole engineering departments can stand still.

EW: You could argue that technical debt is actually one of the key points to successful software projects. The term was coined by Ward Cunningham in 1992; he said that shipping first-time code is like going into debt. A little debt speeds up development so long as it's paid back promptly with a rewrite. The danger occurs when the debt isn't repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a standstill under the debt load of an unconsolidated implementation.

This clearly says that the technical-debt metaphor is closely related to financial debt. It's about shipping something quickly and going into debt. Then you need to repay the debt by increasing the quality later. If you don't do that, you will have to pay interest rates because your productivity goes down.

This is a good metaphor to use when talking to management, because they should be familiar with financial terms and you can tell them this is just like getting a loan at the bank. You have some benefit for some time, but then at one point you need to repay it … plus the interest rate.

SJ: A while ago I had a discussion with Ward Cunningham about technical debt, and he said technical debt is actually a strategy because we can quickly reach a business goal by going into debt. For instance, it's much more important to bring something on the market very fast than having perfect code and being late. Eric Ries describes this in his book *The Lean Startup* (Crown Business, 2011). When he worked at startups, he was always so happy that he wrote perfect code, but in the end nobody used it. So it's better to build something quickly and bring it in front of the user to see if it's actually useful for anybody. If it's, we can pay back the technical debt. If we create perfect code for functionality and don't know if it's useful or not, it's a waste of time.

Henrik Kniberg from Spotify wrote a blog post describing this. We see perfect code as a waste if we don't know that the functionality it creates is really useful. [Developers] come up with the functionality extremely quickly, but it's not in very good shape. They bring it in front of the user, and if they see that the user likes it and they want to build on top of that functionality, they refactor it and make it nice. We see these strategies over and over again. Twitter seems to rewrite its system all the time. Amazon was also a very different system in the beginning.

EW: When I talk to software architects, I give them exercises and they often come up with solutions that have scalability in mind. They think scalability is their main concern, while in fact they should probably focus on time to market because otherwise they will not reach the point with the software when they would need the scalability, because then the company would already be bankrupt or the business case would be gone.

So as you see, [technical debt] isn't necessarily always a bad thing. Still, you need to deal with technical debt somehow. One of the great ideas I came across at one point was by Eric Evans, who wrote *Domain-Driven Design* (Addison-Wesley Professional, 2003). There is a part of this design that basically everyone knows about, the ubiquitous language and repositories. But there is a different part of that book that not too many people seem to have read. That's about strategic design and design on a more coarse-grained level.

[Evans] says that you can't have the same quality throughout the whole system. You will have good and bad developers on your team. Even if you have a very, very good team, there will still be better and worse developers. What can you do? You can leave it to chance which parts have better and worse quality, or you can make a conscious decision. Which parts of the system are really important concerning changeability? You might get that information from historical data, or you can think about it from a business perspective. "Which parts, if we can change them quickly, will give us a

Technical debt is actually a strategy because we can quickly reach a business goal by going into debt.

SOFTWARE ENGINEERING RADIO

Visit www.se-radio.net to listen to these and other insightful hour-long podcasts.

RECENT EPISODES

- 222—Apache Storm project creator Nathan Marz joins Jeff Meyerson in a discussion about a tool that brings Hadoop's processing model to streaming data.
- 223—Joram Barrez speaks with Josh Long about the Activiti business-process-management platform and building process-driven applications.
- 225—Systems performance guru Brandon Gregg discusses performance tuning, measurement, and visualization in the cloud with Robert Blumen.

UPCOMING EPISODES

- Robert Blumen sits down with Fangjin Yang for an overview of OLAP (online analytical processing) and the Druid real-time analytical data store.
- Barry O'Reilly explains how to apply lean and agile principles in software organizations to achieve velocity and scale.
- Tobias Kaatz speaks with Jonathan Groß about the specific challenges software developers face in the scientific world.

competitive edge?" For example, the way you do your shipping is very important. In that case, the part of the system that does the shipping should be of high internal quality.

Let's say you have a really nice domain model that's highly sophisticated, and the code is of quite high quality. Then you have a different part of the system—for example, the part that deals with the customer—and that's just standard software that has low quality and a pretty awful domain model. To make sure this awful domain model doesn't leak into your valuable shipping system, you build in an anticorruption layer that separates those two models and translates between them.

You decide which parts are important, and you care about those and have your best developers working on them. You monitor the quality closely. There are other parts where you might even use standard software that you bought, or you can just stick to your legacy system. I think that's an interesting way of strategically developing the quality of a rather complex system.

The next question is whether it's realistic to have debt-free systems.

SJ: Is it possible to have a technical-debt-free system? I read quite often about "no more technical debt" and "how to be debt-free in 10 easy steps." I think we should just accept that there will always be technical debt. Even if you have a debt-free system, how do you achieve it? You probably have to invest a lot of time and money, and that'sn't necessarily tied to the success of the project.

EW: The key point to take away here is that the original implementations for Amazon and Twitter were hugely successful, but there was quite a lot of technical debt. Technical debt isn't tied to the commercial success of a project at all. You can have an enormously successful project or business that's based on a piece of software that's full of problems. Then you can do a rewrite.

Extreme programming came up with the idea to set the quality dial to the maximum and have no compromises about technical debt at all. That might be a bad idea, because then you invest a lot of resources, money, and effort in maintaining high quality even though it's not necessary. It might not even influence the commercial success at all, because it's something a user doesn't even see.

So what can we actually do about technical debt? One of the ways you can deal with technical debt is to create a buffer task per release. You could say, let's allocate 10 percent of the time to the team, and have the team work on technical things that they think would improve things. You can even spend more of your budget on technical debt. You could have technical releases that just improve the code base. That means the effort invested in handling technical debt isn't evenly distributed, like it would be with those buffer tasks that have 10 percent per sprint.

Let's say we want to change the registration process. When we change it, we improve the software quality to make implementing it easier. That way, you invest the budget for improving quality in those areas of the code where the changes are actually made. It's also factored into the use cases. So it's something that can be decided by management. They can say, "I don't want to do this story—it's so awfully expensive because the quality is so low."

However, at the end of the day, quality should be a business decision. It's about prioritizing quality over features. If you improve quality, it will pay [you] back in the long term. However, if you really need to get this feature done, because otherwise your business case is gone or you have other severe business consequences, then the quality doesn't really matter. I think the hard thing about handling technical debt is to enable the business to decide which part should have higher quality and where to invest effort.

Communication with management is also the core of the metaphor of technical debt; that's why it was introduced in the first place. To some extent it's also about trust. If the developers know how to handle quality and how to keep it up, you can have them decide where quality should be improved. Otherwise, you would need to basically beg for a budget to invest in quality. That might be very cumbersome and hard. So I think it's about trust but it's also about investigating where you can get a payoff.

SJ: I think we have one important point left. Frank Buschmann, who is widely known for the pattern-oriented software architecture, asked, "To pay or not to pay the technical debt?" He gave three answers. Point one is debt repayment. We have a very bad piece of code or component of a system, and we decide to completely refactor or replace the code with a stable, good design. You should only do that if the code is really bad and you know you will often have to build new functionality on top of that in the future.

The second point he proposes is debt conversion. You have a component or part of the system that has a very high technical debt, but replacing it's not a solution. For instance, you have a 30-year-old legacy application—you can't just throw it away because it's too expensive and risky. But you can try to transform the system to a good but not perfect solution, which has a lower interest rate. It's still not perfect, but it's much better than the old system.

The last one, which I hear quite often and think is a valid point, is that we must accept technical debt at some point. We just pay the interest. We know the code isn't very good, but we live with it. The cost of refactoring a not-so-good code base to a good one is more expensive than working with the not-quite-right code. I think that's something we always have to keep in mind. We have to constantly ask ourselves, "Should we really make it good? Or does that debt cost more than just living with it?"

EBERHARD WOLFF is a Fellow at innoQ in Germany. Contact him at eberhard.wolff@innoq.com.

SVEN JOHANN is a software developer at Trifork Amsterdam. Contact him at sven.johann77@gmail.com.

See www.computer.org/software-multimedia for multimedia content related to this article.

IEEE Software (ISSN 0740-7459) is published bimonthly by the IEEE Computer Society. IEEE headquarters: Three Park Ave., 17th Floor, New York, NY 10016-5997. IEEE Computer Society Publications Office: 10662 Los Vaqueros Cir., Los Alamitos, CA 90720; +1 714 821 8380; fax +1 714 821 4010. IEEE Computer Society headquarters: 2001 L St., Ste. 700, Washington, DC 20036. Subscribe to *IEEE Software* by visiting www.computer.org/software.

Postmaster: Send undelivered copies and address changes to *IEEE Software*, Membership Processing Dept., IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08854-4141. Periodicals Postage Paid at New York, NY, and at additional mailing offices. Canadian GST #125634188. Canada Post Publications Mail Agreement Number 40013885. Return undeliverable Canadian addresses to PO Box 122, Niagara Falls, ON L2E 6S8, Canada. Printed in the USA.