Managing Technical Debt: A Focus on Automation, Design, and Architecture

featuring Ipek Ozkaya and Robert Nord as Interviewed by Suzanne Miller

---

Welcome to the SEI Podcast Series, a production of the Carnegie Mellon University Software Engineering Institute. The SEI is a federally funded research and development center, sponsored by the U.S. Department of Defense. A transcript of today's podcast is posted on the SEI website at sei.cmu.edu/podcasts.

Suzanne Miller: My name is Suzanne Miller. I'm a principal researcher here in the SEI Software Solutions Division. I'm very pleased today to introduce you, again for some of you because they've spoken with us before, to Dr. Robert Nord—we call him Rod—and to Dr. Ipek Ozkaya, also from Software Solutions Division. They're going to talk to us today about technical debt, how it impacts software costs and other aspects of technical debt that software engineers and managers need to know about. But before we get started on that topic, for those that haven't listened to our podcasts before, could you give us each a little bit about your background and what drew you to the SEI and, in particular, to this research? And why don't we start with you, Rod.

Rod Nord: Well, thank you, Suzanne. Prior to working at the SEI, I was working in an industry R&D lab, about the time when the idea of software architecture was just emerging as a software engineering discipline. So I had an opportunity to talk to developers and engineers throughout the company and in the business units and to ask them, *How do you describe and represent the software in the large-scale systems that you're building?* They would draw different pictures, and from that I would start to think about, *How do we start to codify those representations in terms of documentation?* so that we could start to codify those practices, and everyone could take advantage of it. Once you write those things down, then you can start to ask some questions. And then from there, start to think about, *How do we codify the analysis, the evaluation that takes place?* Then we talk about the rationale, and that gives us a little bit better insight into design.

When I joined the SEI, we had these different methods for analyzing, describing, designing architectures. Then, I got interested in thinking about how do they fit together, or how do we fit them in with other practices, like Agile, or in terms of a larger software development lifecycle. Since there was such a great interest in Agile, that was one of the areas that I focused on. With Agile, it brings a new challenge because, with Agile, we wanted to start to think about what can we produce in terms of a minimal viable product every week or two in terms of these short sprints.

Now, we have that challenge. How do I decide what I want to do in the short term in terms of meeting my minimal viable product objectives? But at the same time, those decisions that I make, I want to ensure they can still be productive in the longer term and not undermine productivity. That led me to an interest in technical debt and trying to understand how the tradeoffs of short term and long term played out.

Suzanne: That's really what this is going to be about is, How do we make those tradeoffs? Ipek, how did you come to be involved in this work?

Ipek Ozkaya: I come from a design background as well, but a little different. Before joining the SEI, I was at the Carnegie Mellon University side of things and working on my PhD in research in an area called computational design. What that is, is to develop software tools for typical design professionals. It's a very interdisciplinary field. So at any day, someone who needs to build a design product, be it a house, be it an industrial-design product, needs to work with someone who is in the software business to be able to actually bring it to life, to be able to draw about it, to be able to reason about it. So that was where my work was in terms of understanding how these different professionals talk to each other and what does it mean to develop computational tools, which are design tools. There is an overload of the design perspective.

A lot of the challenges that I realized are whether people are talking the same language and how do they make the tradeoffs? It's very similar but from a different perspective, design tradeoffs. Then, after I finished, I joined the SEI. And there software architecture takes a different form but with the same principles. And again, *How do we make these tradeoffs?* And people oftentimes overlook the long-term consequences, as well as the short-term importance. You cannot wait too long to make that prototype. Putting something out there is as valuable as making the right decisions. That's a very challenging tradeoff. And when the concept of technical debt fell in our lap, it was actually very eye-opening because it's very vivid from communicating that, and it's very powerful because it can communicate it to a product manager, to a project manager, but to a developer, as well as a domain expert who may not understand the technology that actually recognizes.

Suzanne: Technical debt is a very easy metaphor for people to understand because most of us at least…I can't speak for everyone, but most of us do have debt in our non-work lives. And so, that concept immediately resonates. *How am I going to pay it off*? Right? That's one of the questions. And that's the same question that we have to ask in the software world, is *If you're going to incur this, how do you pay it off?*

So, let's get into the meat of it. So you've been studying technical debt for several years. I've worked with you in some of the areas. I had some fun developing some ways of communicating about this. What is the most important thing for people to understand about technical debt today? What have we learned about technical debt and how it occurs, and how we do manage it, and how do we make those tradeoffs, and how do we answer that question of how are we going to pay it off? So, Rod?

Rod: From a practical experience, I think a developer would experience technical debt as a source of friction. So, *have they made decisions in the past that are now slowing them down so it's causing extra work and decreasing productivity?* And the other aspect is that even though technical debt is visible to the developer, it's largely invisible to the rest of the organization or to the users of the system who might be more focused on the features but not have such insight in terms of the extra work that the developer is incurring.

Suzanne: Let's talk about an example of intentional technical debt because we have both kinds: intentional technical debt and unintentional. We'll talk about the other one in a minute. But an example for me for intentional technical debt would be, as you said, *I have a minimum viable product that I need to demonstrate that we can execute some kind of an algorithm*. When I write the code for that algorithm, I don't write it in the most efficient way possible.

If we were to take the code that I wrote for that algorithm and actually put it in the field, it would have bad performance, in terms of what's expected. But I can at least demonstrate the feasibility of, *Yes, we can execute that algorithm*. And then the short term, that's, as you said, sometimes having the prototype demonstrating feasibility has a lot of value. So I have intentionally incurred some technical debt. As a developer, how should I communicate that to people, not just *Hey, great for me, I got this algorithm done* but, *oh yeah, but by the way, we're going to have to deal with this later*? How do people deal with that in today's world?

Rod: That's an excellent example. As we said, software is about tradeoffs. In this case, if you have a deadline, it may make sense to intentionally incur debt, just as you were mentioning with the metaphor. We might take out a loan, a mortgage for a house where we are getting some short-term benefit. The key idea, as you said, is to take that with intention, consciously making that decision but then bringing that awareness to your team and to the organization.

One way of doing that is to put that into your registry. If you have a backlog where you have user stories or features, then it makes sense at the same time as you are incurring that debt to create another issue for your backlog that this is a technical debt. It makes sense at this time to incur it, but once it's in the backlog, then you can start to monitor it. As you said, if it's kind of an expedient solution to make this deadline, it may start to slow you down, and that's a form of interest. So then you have to kind of monitor that interest to see, well, is it worthwhile paying a little bit longer to get that immediate benefit? But then as you start to plan future releases, you can start to anticipate what those interest payments might be and when you might pay that off.

Suzanne: So, for example, I may make an explicit decision that when we go to go back to this module where this algorithm occurs, *Later, we're going to have to do other things, other stories with it*. That's the time at which we're going to bring in performance tuning, I'll call it, to get rid of the technical debt on that.

Are people actually doing that? That's one of my questions. I know we recommend these things, but are you seeing evidence of changes in practice in our user communities that are actually saying, *Yes, I'm going to be explicit about this technical debt. I'm going to put it in the backlog. I'm going to decide when I'm going to deal with it.* Because I know five years ago, I wasn't seeing much evidence of that, and I'm just curious what you have seen since then.

Ipek: I think people are starting to recognize it. More importantly, I think they are using some of these mechanisms that are already available to them to communicate it. And so, we're looking to some of the code comments for example, or the issues that actually they do talk about technical debt, and they talk about it correctly. For example, developers find out an unintentional technical debt as, *Over time, we knew we had to change this interface. We didn't because other priorities kicked in. Now, if we don't do it, we're going to actually be in much, much more trouble in refactoring it.*

This vocabulary is in the developer's common vocabulary. Some organizations are more savvy and actually create the labels where they put the technical debt into their backlogs or manage it. Others make it more organic to the development environment. Other organizations actually fear technical debt because they think, *OK, it's maybe going to incentivize, but they try to have other mechanisms to help developers...*

Suzanne: They fear making the technical debt too big.

Ipek: It is definitely in the vocabulary of the developers. It's also much more common to understand. Like, for example, we also see examples where, *Oh, no, this is not technical debt, this is actually a bug that you have in there*. So there is this subtlety that developers also

appreciate and talk about. That is actually very encouraging because that means we're starting to manage it, we're starting to recognize it.

Suzanne: Although I will say, I still have to correct a lot of people more in the management side than the engineering side that they have this equivalence between technical debt as all the defects that have built up. That is not a true statement. Defects can incur and can generate technical debt the same way intentional decisions do, but the bugs themselves, the defects themselves are a whole thing that need to be managed on their own.

Ipek: The subtleties are not very easy to differentiate. We were very fortunate to interact with people who are in the trenches, as well as tool vendors, who recognize some of these because, over time, those defects become significant symptoms. But when do you recognize them as symptoms versus technical debt versus defects? That's where I think our work has actually established and helped some of it in terms of making the differentiation, recognizing that defects are actually symptoms, and some of those defects need to be treated differently. Hence, maybe you need to label it differently as well. I think that's where it's helpful.

Rod: That was part of the process of moving from the metaphor to start to think about the theoretical underpinnings and then the practice. So as Ipek was saying, rather than just talking about technical debt, we want to think about, what are the symptoms or the consequences and then trace that back to a concrete development artifact. What is the actual code or the architecture decision that was made that is the source of the technical debt, as well as differentiate that from the consequence. So perhaps like you're in a deadline-driven environment that provides an environment where maybe people are creating unintentional technical debt.

It's important to understand the environment to make changes, for the longer term, but then just changing the environment doesn't really change the actual artifacts. You need to think of it in terms of those three components: the symptom, the development artifact [and its environment], and the consequence.

Suzanne: I had a very interesting conversation with someone who is in an organization that has made the choice that every time you touch a piece of code as a developer, one of your duties is to simplify it as much as possible. So in many of our government settings, we have, *Only change what you've been told to change. Don't touch anything else.* We have a lot of legacy that is fragile, and there's a lot of reasons for that. But this group kind of went the other way. They said, *Simplify it and don't worry about long-term effects because we're going to turn over the software every three months*. I'd never heard that before. One of my immediate reactions to that was, *Well, that's one way to solve the technical debt problem*, but with an assumption that they are learning enough on each cycle to actually do better each time.

But have you run into that pattern? I haven't run into it in the government. This is in the commercial space.

Ipek: I'm not an expert in that space, but a very similar phenomenon is talked about in the gaming industry because the visualization capabilities, the processors, and everything changes as quickly as it does in both hardware as well as software. From their perspective, they worry about the version that is out there. Again, I'm not knowledgeable in that area, but a number of years ago, that example was given to me, not necessarily in the context of just technical debt, but from the context of long term, *You might actually be better off just starting with new capabilities that are out there, new frameworks, new hardware and thinking of that environment, rather than trying to think about reusability, evolvability, and whatnot.*

Of course, there is learning, and there are pieces of it that probably evolve. But the focus of it might be different. That's another example. There's definitely the organizational and the domain aspects that might change your perception or how sensitive you are to it. But to itself, that is technical debt management.

Suzanne: It is not our normally thought-of strategy, but it was an interesting contrast to many of the systems that we work with.

Rod: That is one of the focuses of our research—to think more about the implications for managing technical debt or looking at it from a strategic point of view because a lot of the tools that are coming up to try to support technical debt focus on the code and all the hundreds or thousands of violations and then associating a cost to fix them—the principal, if you will—and then rolling that up into one large number, saying, *You have all this debt that you have to pay it off.* But it's not always black and white. As we talked about, maybe you want to understand the principal as well as the interest, and whether it's worthwhile to pay the interest for a while or as you were just saying...

Suzanne: Until it's replaced; it's going to be replaced soon enough.

Rod: Yes, replace or declare bankruptcy or amnesty if you're not going to be changing this for a while, then you can...

Suzanne: You have some more degrees of freedom in terms of how you manage it.

Ipek: Also, it's like what to prioritize. Knowing the debt and prioritizing the new functionality or adding on to it is completely different than not knowing about it and adding on to it, even if you decide to leave it there. The strategies, the design choices, or the implementation choices you make will definitely change when you're aware of it, even if you decide not to do anything

about it. You might decide to maybe decouple it from the rest of the software. The approaches you take still protect you going forward from some of these aspects.

Suzanne: In contrast, my father has a pacemaker. I really don't want people just replacing the software in the pacemaker every three months and saying, *Well, that's the old software, and we've learned more.* I want a little more on top of that to make sure that there isn't anything that goes wrong in the field. We definitely have systems that have long-term lifecycles. We are looking at technical debt from that viewpoint of that longer term tradeoff analysis that is absolutely necessary. It's not just appropriate; it's necessary.

So what's the latest in this area? We are talking about some of the basics. But what is the new thing that you've been working with, struggling with, discovered, in the last couple of years that our viewers haven't really heard about yet?

Ipek: I don't know whether this is the latest, but I think tool support to make these abstractions easier is going to be the newest and the latest for a couple of more years because automating these processes, automating these analyses is not trivial. Also, they have these niches based on the language of choice, based on the framework of choice, or the domain of choice. The good news is there are more researchers and more tool vendors who are actually within this domain and looking into it and also integrating some of these tools into the continuous-integration, continuous-deployment pipeline; [these] are some of the newer approaches, which will hopefully, in the short term, eliminate unintentional technical debt because you'll be able to catch these obvious ones at least earlier.

Suzanne: Through the automation, you are going to have enough repetition of the testing and the integration cycles that you have confidence that you've actually caught the things that it's intended to catch.

Ipek: Correct. But from a tooling perspective, being able to develop analysis tools that are able to abstract the design choices, the framework choices, that's still an ongoing work area, which makes it exciting as well as also an opportunity.

Suzanne: If you think about humans making choices, one of my favorite phrases is, *Context is everything*. The context in which a developer is working: how well they know the language, how well they know the domain, how well they understand the as-is architecture, the intended architecture, how well they understand the context of where their software is going to fit; all of those things, that context influences design choices. And many of those things are fixed—language, domain has a boundary, but not all of them. And so, I would see that as a challenge, and as a tool builder, understanding where is the boundary of things that I can reason about from

a computational viewpoint versus where I really have to get out and look at the comments, talk to the coders, and ask the question, *Why, why did you do this?*

Rod: In addition to automation, there are also advances in what we talk about the coverage in the technical debt landscape. Originally, it got started as technical debt with a focus on code. But as we've been talking, our work has been kind of lifting that and looking at design and architecture. So there's some promise there. Another area of that landscape is the production. There's the build, test, deployment infrastructure. If we think of infrastructure as code, there could be technical debt in that infrastructure as well as in that delivery process in terms of the alignment of these different stages of testing and internal environments as well as deploying it to the field.

Suzanne: So when I don't have what we call environment parity where I can't establish the development environment and the deployment and test environments being equivalent, I've got...

Rod: Yes, there could be misalignment there.

Suzanne: I've got some technical debt in my infrastructure. Oh, that's cool. Okay, that's new, that's new. I'm excited.

Ipek: And also, we tended to think initially in terms of the source line of code. Even if we say, *OK, it's the source line of code, when I change it, I need to test it, I need to put it into the overall structure, and I need to push it to the end users, to the development pipeline*. So it's not just that, *Oh, I have code debt.* That's not necessarily the piece. You have to really think about it from an internal perspective. So that's actually important.

Suzanne: How easily are software architects and possibly system architects accepting the idea that they may be intentionally incurring technical debt in the products that they are creating, because that's a tough job? I mean, the software architect, if you go too detailed, now you are imposing constraints that aren't necessary. If you're not detailed enough, then nobody knows what it is that we're doing. If you do it too long, then the world changes, and it is overcome by events anyway. How do you start communicating about that in a productive way with those higher level kinds of work products and communication mechanisms?

Ipek: In our experience, and maybe we've been fortunate, someone who has been in the field and who is experienced knows the fact that they are making tradeoffs, and those tradeoffs have to be communicated. From a technical-stakeholder perspective, architects, developers, we don't get any pushback. What I think might happen at times is, we haven't seen this situation where they get nervous that they would be penalized for technical debt or not.

On the other hand, when we talk to people, they say, *Could you please have this conversation with my manager as well?* I think it's really them communicating. Even if the managers or the non-technical stakeholders accept the fact that there is technical debt, they may not always appreciate the timelines it takes and how it might need to be resolved and what it takes to resolve them. So I think that's where our work becomes kind of a bridge between these different stakeholders and giving each of them different tools to make that communication more on the same page, rather than talking above or below.

Suzanne: And so, yes, one of the things that I know from dealing with a lot of different sides of software and system acquisition is the time from when a decision is made to the time when you actually see the effect of that decision, depending on the character of it, can be quite long. So when you have these delayed effects, it can be very difficult to get management to sign up to a decision that isn't going to have an effect for months, possibly years. It's easier to get them to sign up for a decision that they can see the effect of very quickly. So there are some dynamics there, system dynamics actually, that kind of play into when managers don't understand the sort of long-term causal loops versus the short-term loops they're more likely to make some of the short-term decisions, even if that's not the best technical solution in the long term.

I can see where the technical debt discussion can get complicated when you start trying to make them understand some of these tradeoffs. *If you go with the short-term decision, then no, you're not going to see it in three months, but you or your successor's going to see it in 12 months or 18 months*. OK, I'm going back to the debt metaphor. It's like those balloon mortgages, right? *You only have to pay a little interest now, but then you're going to have the big balloon payment at the end*. Some of us are more willing to accept that than others, and none of us like it when the balloon payment comes. So that's a true thing.

Rod: That is an active challenge. But if we think of technical debt in context, we could think of three layers. One is when we're looking at the software, we're looking at the structure of the system and all those metrics that go along with understanding the architectural health of the system. So that's the structure.

But then on top of that, we can think of technical debt in terms of the principal and interest and then, pointing to that structure or the development artifact that is creating those symptoms. That principal/interest tends to be more within the development team, and I think that's a contribution now in terms of how they could manage it. But then on top of that, is the business side. So the business risk, is there a certain liability based on that technical debt or is there an opportunity cost.

Some experiences that I've seen with system, software architects and enterprise domain, saying that their management at least is very much attuned to this risk liability. So technical debt feeds into risk liability, along with other factors.

So by having that conversation to show how this growing amount of interest—not just paying off the principal, which they might be willing to keep the debt [associated with the principal]—but if you can make that argument that this is how the interest could play out and the liability.

Suzanne: And this is the liability you're accepting from a business viewpoint.

Rod: That resonated more, on the business side, at least for his work that he's working in.

Ipek: Also, relying just on the maintainability argument is not the right thing to do as well. Obviously, it's about long-term maintainability. But the reason you have the long-term maintainability is that it has effect in terms of the user observable or more of the performance-related or unrelated aspects as well. So I think sometimes we get lazy and don't do that mapping and assume that different kinds of stakeholders would be able to do that mapping. Helping them make that connection actually also helps make the conversation a lot easier.

Suzanne: We haven't talked about this before, so I apologize if this is a surprise discussion topic, but have you interacted at all with the model-based engineering community in terms of how using models as the representation artifact of architecture and functionality can help people to reason about technical debt as one of the things that the models reason about? I can't think of an instance right off the top of my head of a modeling activity that I know of that is actually trying to reason about technical debt. But have you guys ran into that yet? Because that seems that might be a promising area of research.

Rod: Yes. So there are models in the broad view. We would include dependency-structure matrices as a form of model and then some of the source-code analysis tools building on dependency-structure matrices certainly address that model of the architecture that is important for understanding the structure that we talked about and the principal and the interest. In terms of model based, some of the other languages, those that tend to focus more on operational qualities, like performance or reliability, I would say have to be complemented with some of these other models that focus more on the module, on the architecture so that we can understand the structure and how we can support evolving the system over time in addition to supporting the operational qualities.

Ipek: Our approach is to, at the end of the day, anchor the technical debt we're talking about on a software-development artifact. From that perspective, it's either code or tests or builds. These are all software artifacts. A model, in that perspective, as long as it's a software artifact, system

artifact, can actually be part of that abstract analysis. We haven't done or seen analysis that runs on the models, but models generate code, right? That's part of that. Some of our analyses, we did apply into some of those codes. And some of them are able to...

Suzanne: Codes that were generated.

Ipek: That were generated. Some of it can provide insights. But, of course, that needs to be tested not only necessarily on the model. From a software-artifact perspective, it's definitely relevant. But what does it mean to generate analysis on the actual model itself from a runtime perspective? That could be of interest to some of the communities, definitely. That could definitely be part of our framework. That's not excluded at all.

Suzanne: We have talked about infrastructure as code is one of the things I also heard in the hallways at one of the places, is models as code, which they are. They are codified views of things, but there is also the digital-engineering strategy from the U.S. federal government, trying to be more cognizant of the fact that these models depend on codifications of ideas. And if we're not going to treat them as code, we take some risks in the models themselves.

Ipek: At the end of the day, they become part of the executing functioning system, so how is that part of the analysis? I think we sometimes fail to think of it that way. We think of it only as a representation. Some of them are going towards the aspect that they're beyond the representation. Then you can run the code analysis and all of that or other kinds of analyses on them.

Suzanne: For viewers that haven't been involved in this work so far, what are a couple of the key resources? All these will be mentioned in the transcript, and the links will be in the transcript. What are some of the key resources where if I'm a software developer, and I'm just now becoming aware of technical debt as a concept, where do I go? How do get smart about this?

Ipek: There are some resources that are already available like from our website. All of our published papers, as well as some of our tooling, will be available, like scripts of our analysis techniques are out there. We've been running the Technical Debt Workshop. Now, it's actually going to be in its second year as a conference. The Technical Debt Conference is collocated with the International Conference of Software Engineering in 2019 in May in Montreal.

That's definitely an opportunity to both learn, to submit work, to interact with us. We have different tracks, and that's also available. Myself, Rod, and Philippe Kruchten have been working on what does it mean to actually put these things into practice. So a practitioner-oriented

book will be coming up later in 2019 in May as well through Addison-Wesley as part of the SEI Series in Software Engineering.

Suzanne: Oh, May of 2019, the conference. You're going to be dealing with some technical debt as you try and get towards that deadline!

Ipek: We hope we are ahead of it, but we will see.

Suzanne: That's definitely something to look forward to. What is the new research that you're going forward with as you continue on this journey? You've got sort of that practitioner view going, but what's the more advanced stuff that makes the researcher's heart happy?

Ipek: I think we are starting to explore more in terms of the continuous-integration, continuous-deployment pipeline. What does it mean to integrate these techniques into that pipeline? What is the correctness of some of these analysis techniques from the receptiveness of the developers, and how do you bring some of these different code artifacts together, not just the code analysis, but bridging the gap between the code and the architecture and the infrastructure. That's what's on our radar going forward. That's one piece of it.

Suzanne: Anything else you wanted to mention, Rod?

Rod: This past year, we've been starting to partner with our data-science team at the SEI as well to see if we can apply some machine-learning techniques to some of these development artifacts. And that's something that we're continuing to work on and will be reporting in the future.

Suzanne: So there may be a blog post in the future on that. Excellent. Excellent. And both of these folks are frequent bloggers. So if you look for either Ipek Ozkaya or Robert Nord, you should be able to find some interesting things to read.

I want to thank you both for joining us and updating me on the stuff I haven't been able to work with in the last couple years. I am hearing a lot more use of the term and apparent understanding of it even, in the areas that we work. I think you're making an impact on the community out there. And that's what we're after. So, thank you very much.

For our viewers, you'll see all of these resources we've talked about and other blog posts that are relative to this in the transcript for this. I want to thank all of you for viewing.

Thank you for joining us. Links to resources mentioned in this podcast are available in our transcript. This podcast is available on the SEI website at sei.cmu.edu/podcasts and on Carnegie Mellon University's iTunes U site and the SEI's YouTube channel. As always, if you have any questions, please don't hesitate to email us at info@sei.cmu.edu. Thank you.