
# Technical Debt as a Core Software Engineering Practice

*featuring Ipek Ozkaya as Interviewed by Suzanne Miller*

---

**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 and operated by Carnegie Mellon University. Today's podcast will be available at the SEI website at sei.cmu.edu/podcasts.

My name is Suzanne Miller. I am a principal researcher here in the Software Solutions Division of the SEI. I am very pleased to introduce my colleague and friend, Ipek Ozkaya, also in the Software Solutions Division.

She is going to be here to talk to us about technical debt and what it means in software—because it is not the same in software as other places—and also to talk about technical debt in the context of some of the important things about educating our incoming software engineering professionals.

Ipek, if you would not mind giving a little bit of your background as to what kinds of experience you have that brought you to this issue of technical debt, and then we will talk a little bit more about what it is.

**Ipek Ozkaya:** Sure, Suzanne. Thanks a lot, first of all, for having me in your podcast series. As you mentioned, I am a principal researcher at the Software Solutions Division. At the Software Solutions Division, I work predominantly in the software architecture area.

Over the years, we worked with a lot of customers who have gone through issues in terms of legacy modernization, agile adoption, *how much architecture is enough?*, *how do you change technology as things change?*, and all that.

Across those different kinds of customers and different kinds of challenges, one thing remains constant: It is that inability to be able to articulate what actually accumulates as it stays on the system and then what actually could stay. For example, *Do I really need to upgrade the legacy? Do I really re-architect? And how do I really make that tradeoff decision?*

At the end of it, it is really about understanding both design decisions as well as your business decisions. These concepts have been around for a while about maintainability, evolution, and whatnot. But once you express them in terms of financial concepts and talk about them in terms of those tradeoffs, it starts resonating.

That is how my team and I started looking into technical debt as both a communication mechanism but also as a way to combine multiple research challenges together. That is how we embarked on the project. One of the things that we actually became very pleasantly surprised at, a lot of industry as well as researchers are quite engaged because it resonates with them. It expresses their problems in their perspective.

In 2010, we had a small gathering of researchers articulating what might be actually some of the issues. Since then, we have actually had a growing body of work as well as collaborative groups that have been working on the subject.

**Suzanne:** I really like the fact that this combines the technical and the economic. It also brings to heart, it brings home the role of architecture in making decisions. Not just creating a view of the architecture as we think it will be but continuing to understand the architecture as it evolves, and where it may have break points that are going to affect sustainability in the future or affect decisions that we have coming up.

**Ipek:** Correct. It is actually very interesting because, when you are talking about something like technical debt, there is a positive and a negative aspect of it. The positive aspect is it resonates with everybody. It resonates with the manager who may not be as competent in understanding the details of the particular technology they have been developing in.

It also resonates with the developer who has been opening the same bug over and over again and knows what is going on. That ends up being a positive, but it also has a negative "kitchen sink syndrome" because everything can be piled under technical debt, and you really want to separate this.

We did a study in terms of trying to figure out. *Does it really resonate consistently with different audiences*? It does resonate consistently, and the one consistent message that we have heard is the most painful technical debt that we need to deal with—and we do not do a good job—is around these architectural trade-offs.

If you step back and think about it logically, it makes sense because we are talking about these issues in the long-lived systems that have been around for decades. We are talking about software, which changes so quickly. Technology changes so quickly. There are a lot of disruptive technologies that come in, and businesses do not know how to deal with them. Those all really come down to those design trade-offs. That is actually why it is very exciting for us, but

it brings our expertise in software architecture in a very well-balanced way between business as well as technical stakeholders.

**Suzanne:** Excellent. We have seen this actually cited in the news. It has come far enough that it is made the papers, recently United Airlines and the New York Stock Exchange. What I am curious about is, is the way it was portrayed in the news consistent with what your research is talking about?

**Ipek:** Yes and no. Let me talk about the *no* aspect first. In those examples, the New York Stock Exchange and others, there are so many issues that are going on that are very grave. It is not fair to really just put it under the rug, *OK, it is technical debt and one minor error or architectural change*. There are quite a number of people as well as processes as well as decision-making aspects that go under it. So I feel very responsible in being very careful in terms of when we designate that.

But, on the positive side, the layman out there is recognizing that there is something going on. Part of it is actually not making the right technical decisions and not measuring them, not monitoring them, not using the correct tool sets and whatnot that results in that perspective.

From that regard, it is definitely a correct way to articulate them. I caution because not every failure that happens can actually benefit from calling it technical debt. There are cases where it actually could be a disservice, because it might legitimize—because there is a positive aspect of technical debt as well.

**Suzanne:** When I see people talking about technical debt, frequently I see them really focusing on technical debt as, *Well these are all the defects that have accumulated in the system*, and that is really all they talk about—talk about how technical debt is a really larger concept than that.

**Ipek:** Maybe it is better to step back and look into how it originated. This has been about a couple decades now. It was mentioned in the context of developers take a shortcut, but they know they go back and refactor it. Ward Cunningham was trying to articulate that aspect. When you think about that shortcut, that is actually a very conscious development choice you are making and you know you want to go back to it.

That is really the essence of technical debt that you are making a trade-off. You know you may actually go back and monitor and change that trade-off, but at that moment that buys you some value. From that perspective, the debt metaphor is actually very powerful.

**Suzanne:** I've got to pay it back at some point.

**Ipek:** Exactly. You have to pay it back, but I am getting value. Like your mortgage or some other debt that you might…anything you borrow. That is our goal through our research. We want

to really articulate it such that developers and teams start using the practices that are already available for them, for example, defect management. There are lots of practices that they can definitely use, and come to a point where they talk about those strategic tradeoffs very specifically in terms of *I am borrowing time. I am borrowing resources by making this particular design choice or this particular implementation choice. I will monitor it. These are my terms. What are the conditions I monitor to decide whether I need to go back and change something, or is it OK*? Maybe at some time it just pays it off itself, and you may not need to change it, which also makes it quite a fun research problem for us because it brings these economic tradeoffs: what you really monitor and measure and put value around to be able to make that decision as well as what do you really measure in the system. *Is it defects, or is it really those architectural decisions, design decisions, that are very hard to visualize anyways with the current tools?* And how do you bring that back to a concrete perspective for the developers as well as project managers and other decision makers?

**Suzanne:** The effects of architectural decisions are often not seen until much later.

**Ipek:** Correct.

**Suzanne:** That idea of *I am projecting some level of debt, but I really do not understand how it is going to get paid off until I get much later*, that has got to be one of the challenges. So that monitoring, *What do I monitor so I can see if I am either accumulating more debt or am I paying off the debt?* becomes important. So that has got to be quite the challenge.

**Ipek:** I give the example of we do a lot of emprical work. We are very blessed to be working with a lot of industry as well as government collaborators and researchers, both in academia and industry labs.

One of the examples we use is, for example, a recent one that we come across is the system has--observes—continuous crashes. It comes from integer overflow. So they look, find it, and patch it in the place where it is found. It repeats itself, and it is until a developer recognizes, *You know what? This keeps coming up. There is something else going on.* Well, it turns out that it is actually an external API that it is relying on that keeps injecting this particular problem. In that bottom line, yes, it is a defect when you keep finding it and patching it and all whatnot. But at bottom line, it is an architectural choice by using this external API, and how do you locate it and patch it at the resource? How do you negotiate a service agreement with this external party?

Because of that it has a lot of these complexities, but also the potential because it helps the developers talk about it with a different vocabulary. Rather than talking about it as a defect over defect, maybe if I talk about it, *Well this keeps coming up. It accumulates. We keep patching it. We have to have a root cause, and I need to talk about it differently.*

**Suzanne:** How do software engineers get educated currently about technical debt? How do they make this leap from *I am going to patch, patch, patch, patch, patch* to *There is something else here and there is a way of expressing this that has meaning technically and for the business?*

**Ipek:** Currently, they do not get educated by this because, first of all, it is a new concept. It is a concept that in some cases does not resonate as well with the developers because they feel like they might already be doing some of the things with their current management practices. What starts making it obvious to them is through these examples where they might have actually dealt with it for a long time.

The number one suggestion we have to teams is to make these more grift problems that keep opening up, or they keep consuming resources because they were not done correctly in the first place, or they need to be revisited. Track them as technical debt as opposed to some other mechanism in this system.

This has the benefit to help the teams to recognize what is accumulating in terms of rework, other defects, other side effects, as well as helping them communicate it more clearly to different audiences. The number one education mechanism is to help them use the resources that they are already using.

Teams are already using some form of sprint management, iteration management tools. They are already using some way of defect management mechanisms. It is really not a big leap to be able to use these tools to that purpose.

The second one is to understand the trade-offs that they are making, and what are some of the measurable aspects of it. *Is it the re-opening of the bugs? Is it the time it takes, that it is longer, or is it some other side effect in the system that needs to be actually made obvious?*

Again these are things that all the developers are already dealing with day-to-day. It is just making that more explicit so that the communication happens in the right place rather than in the periphery, which are the symptoms that we usually observe.

**Suzanne:** You and I have talked a couple of times about the idea of this becoming a core concept in software engineering curriculum, even at the undergraduate level, and getting the awareness of this as something that a professional software engineer is going to have to deal with. Get that to them early. Talk a little bit about what you are doing to promote that and to make that happen.

**Ipek:** We do see technical debt management as a core software engineering practice in the near future, similar to how we have requirements engineering and management, software architecture and so on. I think we need to treat it at different levels.

At the undergraduate level, it is not necessarily a course that is called "Managing Technical Debt." It is really, first of all, making sure that when you are talking about some of the techniques that you are introducing to the students, that they could be used in large-scale systems or where they have larger development, larger teams for these purposes. For example, a very simple one is static analysis and code quality.

Today, a lot of these tools can help you recognize some of these problems that might actually be systematically happening in your system, like complexity or modifability violations. At the undergraduate level, these kinds of things are introduced to them. When you are introducing these subjects, maybe give us homework in terms of *OK, now if you were to locate some of these that might have an accumulating rework, how would you do it?* So there might be different ways at the undergraduate level.

At the graduate level, there are a lot of project-based courses. Today, the reality of software development is rarely from scratch. It is mostly…

**Suzanne:** Evolution.

**Ipek:** When we are talking about evolution, we are already talking about some of those tradeoffs that are, you have to assume and recognize. Then, through those projects there are a lot of opportunities to help the students both understand as well as develop competency in terms of how they could actually manage those tradeoffs.

**Suzanne:** Have you had any contact with any of the accreditation boards, the sort of people that…?

**Ipek:** No, this is not at that point yet.

**Suzanne:** So you are still in the research phase of this?

**Ipek:** There is a lot of research, and I think the industry has embraced it. Although a lot of organizations do not necessarily have specific technical debt management practices, quite a number of them are looking at it in terms of, especially empowering their teams to communicate these at the right time. Because the industry-based systems are large systems. They also need to be able to communicate it with their external customers and owners. So at that level they are really looking into practices.

It is more at the industry, collaborating, and empirical research level. Then eventually it is going to, of course, make its way to education as well. There is a significant increase in terms of the PhD students who are actually embracing some of these topics as they research. Eventually, we will also see it, I think, more in the courses as well.

**Suzanne:** What is the focus of your research right now in relationship to this, and where do you think this is going from the SEI viewpoint?

**Ipek:** The bottom line of all of this discussion comes to how do you recognize data in your system from multiple artifacts? Because we are talking about code, we are talking about defects in systems. We are talking about architecture, and none of these are uniform artifacts that you could do analysis.

There are two aspects of our research. One of them is, *How do we actually combine data that comes from these heterogeneous places and intelligently and automatically—or as automatically as possible, semi-automatically—collect that information and present it to the user in a meaningful way?* An example I could give is, we have recognized that issue trackers contain quite a lot of hints about where your technical debt might reside.

So, you might actually run some of these tools in your historical issue trackers and recognize some of the areas that might be connected to them through some of these tools. So, that is one area that we are looking into.

The other one is, it is really, *How do you navigate the level of abstraction of the systems?* Static analysis tools run on the codes. Maybe you are looking into the classes, lines of code. Whereas architecture is a whole different way of organizing the information…

**Suzanne:** A very different viewpoint. Yes.

**Ipek:** And the tools tend to lag behind in architecture analysis. So, our second line of research aims to bridge that gap in terms of how do we extract the information from the systems and navigate that level of abstraction from the lines of code to the architectural perspective in terms of helping the developers, as well as project managers, recognize those tradeoffs.

Because one of the questions that we unfortunately repeatedly get asked—and this is unfortunate because it is too late by the time you get there is—*Here is a piece of software I have, tell me how much technical debt it has.* That is a whole different way of looking at the problem than when a developer makes that decision and the tradeoff.

**Suzanne:** We have not talked about this before, but I am wondering if there are some nice connections between where you are going and some of the direction in model-based sys-software engineering and model-based systems engineering taking advantage of some of the modeling tools that are becoming available.

And maybe using some of the constructs that are--give you hints. And so, if you have a list of things that if you see these kinds of things showing up in the model you may be unintentionally adding technical debt. Have you explored any of those things yet?

**Ipek:** Our team has not, although we have a lot of colleagues at SEI who do work on model-based engineering. Some of our other colleagues out there are looking into, actually, models as a way of recognizing them.

Of course, especially when you are talking about unintentional issues that you might be creeping in, conformance might be an opportunity. *Well if I am not conforming to it based on the model I have developed, I am already deviating from what I want to do*. Or it could be based on, *Well, is this really the kind of performance or availability I am expecting, and does my model conform to it?*

Those questions can be easily asked to the models. Similarly, if I am doing it intentionally, again, *What am I violating, and what are some of the areas of the model?* So, it is really based on the analysis questions that you ask and the artifact would be the model, the code, the commit history or combination of them.

**Suzanne:** So, it sounds like there are many directions for you to be exploring, so you will be busy for a while. I know you like to be busy.

**Ipek:** We are busy for a while, and we also keep the Managing Technical Debt Workshop series. It is been going on healthy for a couple years now, and the next one hopefully in 2017 will be somewhere in Europe. The industry as well as academia keeps engaged and driving those. Those are opportunities for some of our audience to test their ideas, learn more about. We make all of those resources available online.

**Suzanne:** You have a very active page online. I have seen the areas--the software architecture page has got links to the technical debt. You have published in *IEEE Software*. There are a lot of different sources, and we will mention several of those sources when we publish the transcript.

**Ipek:** One of our recent papers in *IEEE Software,* in the issue in "Future of Software Engineering," actually brings some of these things that we talked about together as positioning managing technical debt as one of the core areas that organizations need to pay more attention to going forward. So that is something available already…

**Suzanne:** We look forward to this showing up in the software engineering body of knowledge at some point. I very much agree with you. In my experience with lots of complex systems, the surprises late in the game that were really predictable when you think this way.

When you think about these tradeoffs from both an economical viewpoint and a technical viewpoint, I think that there is a real good case for this to be much more prominent in the thinking of architects, the thinking of managers, and the thinking of developers.

**Ipek:** It is always the short-term versus long-term tradeoff. We have talked about this. We have known this for decades. But I think it keeps coming back because we still lack the tools to communicate it properly. And that is where the research challenges still exist and they are still exciting to a number of people. So from that perspective, yes, we will be busy for the next couple of years definitely…

**Suzanne:** And I know that the folks that work with you are very dedicated to this so you have a, you have got…

**Ipek:** Yes we have…

**Suzanne:** We will be seeing a lot of good things from you. All right. Well, I want to thank you so much for coming over and joining us today and talking about this. We have been trying to get this together for a while so…

**Ipek:** Thanks for having me.

**Suzanne:** It was wonderful. I do want our readers and our listeners to go to sei.cmu.edu/podcasts as a place to get the transcript for this, which is where all the links will be. So that is going to be a resource for you.

The other place for her work, in particular, is sei.cmu.edu/architecture.

This podcast is available on the SEI website at sei.cmu.edu/podcasts and on Carnegie Mellon University's iTunes U site. As always, if you have any questions, please don't hesitate to email us at info@sei.cmu.edu. Thank you.