:14

What does a career in software design and architecture look like? What is the difference between software design and software architecture? These are questions that we will explore in more detail throughout the specialization. However, let's take a quick look at them now. Like many roles in the software industry, the software designer or a software architect role can look very different from company to company. Characteristics like company size, the scope of the project, the experience of the development team, the organizational structure and the age of the company can all impact what these roles look like. In some companies, there may be a distinct role for a software designer or architect. In other companies, the design may be completed by a member or members of the development team. Typically, the software designer role would be responsible for outlining a software solution to a specific problem by designing the details of individual components and their responsibilities. A software architect role would be responsible for looking at the entire system and choosing appropriate frameworks, data storage, solutions and determining how components interact with each other. That brings us to the primary difference between software design and software architecture. In short, software design looks at the lower level aspects of a system, whereas software architecture tends to look at the bigger picture, the higher level aspects of a system. Think of this like designing a building. An architect focuses on the major structures and services, while an interior designer focuses on the smaller spaces within. Great software designers and architects are detail-oriented, forward thinkers. They need to be able to see the product at both the low and high levels. They need to be creative problem solvers in order to come up with a quality solution for the problem at hand. And they need to be able to express these ideas effectively with the product manager and the development team. Does that sound like something you'd be good at? Software design and architecture is essential to the software development process. Let's take a look at what people in the industry feel about this role. Software design is the process of turning the wishes and requirements of a customer into working code that is stable and maintainable in the long run, and can be evolved and can become part of a larger system. That's software design. I like the or because I don't make a distinction between software architecture and software design. I think they're just the same problem at a different scale. Way I like to think of it is that architecture is primarily, begins with understanding what's the business problem that the client needs to solve. Where business doesn't mean necessarily financial business, any business. Once you've realized that that's your primary task, which is to figure out what the client wants, then everything kind of falls in after that. Because If you understand the problem, then you can start to think about what, in your previous experience, as possible solutions, and then you start getting a idea of how your overall solution is going to look like. And that's where I kind of say, really architecture is the study of boxes and lines. Because your first description of what it is you're trying to do is simply a set of boxes with things inside them and lines expressing relationships. Design and architecture is important if you want to have a stable, long- lived system. Anybody can build a system that'll last a week or a month or a year, but if you want to build something that is the basis of other people's work and contribution over potentially a period of years or longer, in some cases, you need to put some thought into it. You need to have somebody whose job it is to look out for the long game and make sure that you are not making suboptimal short term decisions. Architecture is important because if you get it wrong, your project will fail. That's it. It's just that simple. We know it in the building world, and we know it in the software world. Where we're using the term architecture to be this understanding of the relationship between the requirements of the user and the ability to build a system that will deliver those requirements. I think you can trace back most major software failures to bad architecture, where architecture is used in this general sense. One of the key challenges in software architecture is the tendency to have to trade off between speed and quality, If I boil it down, right? I think there's a tendency for the customer in the business to want their, their software, to want their results as soon as possible. And there's a tendency for the engineering team to want to build, the most robust, thoroughly designed, thoroughly implemented system possible. And so we trading off between these things all the time. And I think, it's that tension in that trade off is where you get really good software, you get good designs out of that, but it's a process you have to go through to go through that. So our biggest issue that we face is understanding the client's problem. What is it they really want to do? And in many cases the client actually doesn't know what they want to do either. They come in with only a partial understanding, a vague kind of sense that they could be doing things better. But often, one of our first task is to actually help them understand, with more precision, what their business is. A software architect's job is to be the interface between the product and the customer and the engineering teams. And so for instance, customers will express a requirement or a need they have of the, of the software and it's the architect's job to then work with the customers and their representatives, product managers and such, to come up with the technical requirements of how we're going to solve the problem. And then they take those requirements to the engineering teams and worked with the engineers on how to realize that in a way that is meeting the customer's requirements and also aligned with the technical best practices and nonfunctional requirements that have to be adhered to in the product. The software architect is like a building architect. They're responsible for the overall conceptual integrity of the project. Their main goal is to serve the needs of the customer within the budget that the customer has. I would express software design or software architecture in a couple of different ways. For small things, for simple things, you'll sit with an engineer and you'll whiteboard something out and you'll come up with a design that way and you'll basically get them going. For larger initiatives, larger projects, you're typically writing fairly

substantial design specification documents, where you're exploring all the different possible use cases, all the different possible flow variations and things of this kind, in addition to all of those critical functional and nonfunctional requirements, stability, maintainability, these kinds of things. So, in general, I would say that we communicate software architecture through the written word, through wikis, through white papers, these kinds of things, in addition to fairly detailed engineering design schematics, class diagrams, if necessary, big box diagrams, if it's just a simple high level architecture design. I've been programming for 45 years and one thing I've learned is that the only thing that's really there is the code and everything else that you talk about is views on the code. So, I like to express architecture or describe architecture as saying that, all the things that I'm going to do, the boxes and lines, the prose, the fancy diagrams of the diagrams and napkins, there are simply indexes into the code. That's how you find your way to the actual artifact that's actually doing what you want to do. I tend to apply simplicity first as my main principle, if I'm looking at how I'm approaching the problem. That's the filter that I try to use on it. And I often will find myself, there's a tendency in people to complicate things, to inject complexity because it's interesting. As an engineer, as a technical person, complexity is fun and interesting. And it's only when you stripped away all the unnecessary complexity that you realize you've got the core of a great solution to a problem. And so, I really try to do that and when I'm working with product teams and engineers alike, that simplicity principle really helps to cut through a lot of the confusion. What's the most important principle? Simplicity. That's true and it's the engineering maxim to keep it simple. The reason for that is twofold. One is that if it's simple, you probably have a pretty good chance of getting it right or almost right. That's one part. The other thing is if it's simple, then you can explain it to someone simply, that communication of architecture is important because you're not going to be around forever. And you need to transfer your knowledge over to someone else. And if it's not simple, the knowledge transfer cost is higher because it's more complicated to explain and the chances of misunderstanding are much higher. So, I became interested in software design by working as an engineer and being exposed to larger scale code bases, you know, progressively over the years. When I first started, of course, I worked on mostly very, very small things. And I was very interested in how software was put together and the design at, a micro design level. And then, as I proceeded in my career and I started to be exposed to some fairly large pieces of software that served millions of people, I got really interested in how those things are put together and what is it that makes that successful. And how do you make sure that you're not having to re-implement this thing over and over and over again. And I found that to be a really interesting side of the business that I really hadn't explored before. And so it turns out I really enjoyed that. I think, if you start writing code to do things, to play around, you start asking yourself questions about: well, why is it that this code is really nice to work with and this code here is horrible? And you start asking yourself questions about design and the difference between design and architecture in the software business is, there really isn't any because our business is all self-similar. Issues that you ask about programs are the same issues you can ask about big systems. And I think in my case, I just became interested in this fundamental understanding of the issue of building software artifacts. And then it just naturally scales up, at some point you're building systems with a hundred thousand lines of code. And then, you suddenly realize on your project team that you don't have a million lines of code. And you're now a software architect because you have a million lines of code whereas before you were just a programmer you only had 10,000 lines of code. Most architects that I know started as software engineers. Usually as an intern or a new grad and they work basically overtime. They work on progressively larger and larger pieces of the software that they're responsible for. And what happens is, you start to see those engineers get to a level of comfort where they start to push outside of the code base that they're indirectly responsible for. And they start involving themselves in discussions around the larger impacts to the system of the work that's being done. And that just generally continues until all of a sudden they're actually working at a much higher level of abstraction. And then contributing at a very different level and so that's how you know you've got an architect on your hands. Yeah, there's not a career path into software architecture. What it really is is, if you think of architects as having more responsibility than programmers, what it really is as a career path where you get more and more responsibility, that you do by demonstrating that you're actually good at building things. My experience has been that I didn't think I was a expert programmer until I had been out in the world for 10 years and I think that's consistent with many of my colleagues. Over that period, you start working on bigger and bigger systems and eventually someone trusts you with being the point person to put together a design for a much larger system than you'd ever done before. And then once you've done one of those and it hasn't been a total disaster, you get an even bigger system. So I guess, it's gradual building of your reputation is what makes you into a software architect. I would say the most exciting thing about being a software architect is the satisfaction of seeing the final product put together and out there and being used by real people to good effect, right. Because you spend all this time early on in a project and you have to fight for your nonfunctional requirements and you have to fight for how this is all going to be put together. And then when it all comes together and you've done all that negotiation and it's out there in the hands of a customer and it's valuable, there's a real sense of pride with that. I think, additionally, you also get a lot of satisfaction and pride from making the right call in terms of long term viability of a code base and of a project. And so seeing somebody be able to come to your product, maybe years later, and make some very business critical contribution extension of something that you designed, without having to redesign it, is very satisfying. It tells you that you hit it on the right mark. What's exciting in architecture? Well in general, you don't want too much excitement because that's usually associated with some sort of looming disaster. But what's interesting about software architecture, and that continues

to make it interesting, is that someone always has a problem that's slightly different than all the problems you've seen before, which means that your previous solutions aren't necessarily going to work and you get to do something new. So, it's the novelty that makes up architecture interesting. An architect has to have a number of important skills, obviously, deep technical expertise is table stakes. You have to be a technical guru, I think, at a certain level. In addition to that, you need to be able to communicate with people at the level that they want to be communicated with. So if you're talking to a business person, they don't want to hear about your code. They want to hear about their business problems. And they want to hear how you're solving their business problems. If you're talking to an engineer, they want to know the business context but they need you to talk to them about code. And so, it's really important to have that ability to understand how the person you're talking to wants to be communicated with. So empathic communication, I would say, is really important. Additionally, some basic functional skills like a little bit of project planning and organizational skills, being able to keep a backlog of work so that you don't forget about things. Be able to juggle a lot of different competing concerns at the same time is also a very important skill. The most important ones are what I would call, the soft people skills that you need in order to get people to tell you what their requirements are. This is very hard actually, especially in situations of uncertainty. Clients are very reluctant typically, to tell you the things that they're really bad at. They like to tell you all the things that they know how to do but they're reluctant to express where their understanding of a problem is incomplete or where their business processes just don't work right. And, if you don't identify those areas, you've actually encountered a big risk in your project, because those are the areas that are the problem. The well understood parts of a client's needs are not an issue. It's the parts that are fuzzy and not well understood. But of all the technical skills that you've got, you need this meta skill, which is to look at various technologies and ideas and decide, is that going to be useful to me or not in my particular problem I'm trying to solve? So as an architect, you have to know a lot about what's out there. But not in a tremendous amount of detail because a lot of the stuff that's out there, isn't going to be useful to you, at least immediately. By the time you might need it, it's probably gone through 10 releases anyway and isn't the same thing. So you have to have the skill of being able to quickly assess various technologies and fit them into your understanding of the discipline. So new language comes out, you so say, "Oh yeah, this is yet another procedural language with nothing much different than all these other ones." Or you might see something else and says, "Oh, that's interesting. I wonder if this particular style of approaching the problem, perhaps, aspect oriented programming, just to pull something of the air, will actually help me solve my problem in a better way or express my problem in a better way." Well, staying up to date is a bit of a trick. It's about exposing yourself to as much as you can in the outside world and inside your own company, as well. But in particular, you know, look at what the big companies are doing. What's Apple doing? What's Google doing? What's Amazon doing? And you read their blogs. You play with their software. You get an account on whichever tool you want to use and you start using those things. And you use that for inspiration. And just to see how others are approaching architecture in their systems, right? So there's a number of levels of inspiration there, I think. Additionally, read a lot of just the general tech press and find out what's going on out there in the world. Read academic journals for the appropriate areas and see what's coming a little farther down the line. What are the academics thinking about? So there's lots of those things to go after. So the advice I would give to a new software architect is to get as comfortable talking to people as you can and meet as many people as you as humanly possible. Expose yourself to as many ideas as you can. And share your own perspective as well. And I think it's by by leveraging the community, leveraging those around you, that you're going to be inspired to be creative in your architecture and you're going to get a better understanding of the context in which you're operating, both within your business as well as the broader technology landscape out there. And it'll help you to make better choices, ultimately. The advice you give to new software architect is the same advice you give to a musician. Try and play with people who are much better than you are because that's how you become a better architect. And that means working with people who are better than you are. If you have the opportunity, at the very least, try to read as much of the foundational literature in the field and there's not that much to read. There's a maybe 20 key resources you should go to. Some of them dating back to the original papers in the 70s about coupling and cohesion. And then, of course, writing code. So you need to work with people that are better than you are. Read a lot of code and read a lot of code and that's how you become a software architect. Oh and of course, learning from your mistakes is also quite valuable.