제25장

협상 및 리더십 기술

협상 및 리더십 기술은 오랜 학습, 연습, 그리고 시행착오를 통해서만 얻을 수 있는 하드 스킬이지만, 효과적인 소프트웨어 아키텍트가 되기 위해서는 필수적입니다. 이 책을 읽는다고 해서 누구나 하룻밤 사이에 협상과 리더십 전문가가 될 수는 없지만, 이 장에서 소개하는 기법들은 좋은 출발점이 될 것입니다.

이 주제에 대해 더 자세히 알아보려면 타냐 라일리의 저서 『엔지니어의 길: 개인 기여자를 위한 안내서』 (O'Reilly, 2022)와 로저 피셔, 윌리엄 L. 유리, 브루스 패튼이 공저한 협상 분야의 고전 서적인 『예스로 만드는 법: 굴복하지 않고 합의를 이끌어내는 협상법』(Penguin Books, 2011)을 추천합니다.

협상 및 중재

1 장 에서 우리는 소프트웨어 아키텍트에게 요구되는 핵심 사항들을 나열했습니다. 마지막으로 다룬 사항은 조직 내 정치적 역학 관계를 이해하고 헤쳐나가는 능력이었습니다. 이 부분이 중요한 이유는 소프트웨어 아키텍트가 내리는 거의 모든 결정이 도전을 받기 때문입니다. 자신들이 더 많이 안다고 생각하는 개발자들, 더 나은 아이디어나 접근 방식을 가진 다른 아키텍트들, 그리고 당신의 솔루션이 너무 비싸거나 시간이 너무 많이 걸릴 것이라고 생각하는 이해관계자들까지, 다양한 사람들이 이의를 제기합니다.

협상은 아키텍트의 가장 중요한 능력 중 하나입니다. 유능한 소프트웨어 아키텍트는 조직의 정치적 역학 관계를 이해하고, 뛰어난 협상 및 촉진 능력을 갖추고 있으며, 의견 차이를 극복하여 모든 이해관계자가 동의하는 솔루션을 만들어낼 수 있어야 합니다.

비즈니스 이해관계자와의 협상

수석 아키텍트인 당신이 주요 비즈니스 이해관계자와 협상해야 하는 다음 시나리오를 생각해 보세요.

시나리오 1

이 프로젝트의 수석 부사장 겸 제품 책임자인 파커는 새로운 글로벌 거래 시스템이 '파이브 나인(99.999%)'의 가용성을 지원해야 한다고 주장합니다. 하지만 글로벌 시장 간 거래가 이루어지지 않는 시간(2시간)을 고려했을 때, 당신은 '쓰리 나인(99.9%)'의 가용성으로도 프로젝트 요구 사항을 충족할 수 있다는 것을 알고 있습니다. 문제는 파커가 자신의 잘못을 인정하는 것을 싫어하고, 특히 자신의 의견이 무시당하는 것처럼 느껴질 경우 정정을 받아들이지 않는다는 점입니다. 파커는 기술적인 지식이 부족하면서도 스스로는 전문가라고 생각하기 때문에 프로젝트의 비기능적인 측면에까지 관여하는 경향이 있습니다. 아키텍트로서 당신의 목표는 협상을 통해 파커를 설득하여 '쓰리 나인(99.9%)'의 가용성으로도 충분하다는 것을 납득시키는 것입니다.

분석 과정에서 지나치게 자기중심적이거나 강압적인 태도를 보이지 않도록 주의해야 하지만, 협상 과정에서 자신의 주장이 틀렸음을 증명할 수 있는 중요한 정보를 놓치지 않도록 해야 합니다. 이러한 이해관계자 협상에 도움이 되는 몇 가지 핵심 협상 기법이 있습니다. 첫 번째는 다음과 같습니다.

사람들이 사용하는 유행어와 전문 용어에 주의를 기울이세요. 의미가 없어 보이더라도, 상황을 더 잘 이해하고 협상하는 데 도움이 되는 단서가 숨겨져 있는 경우가 많습니다.

기업에서 흔히 쓰는 전문 용어는 대개 의미가 없지만, 협상에 들어가기 전에는 귀중한 정보를 제공할 수 있습니다. 예를 들어 특정 기능이 언제 필요한지 물었을 때 파커가 "어제 필요했어요."라고 답했다고 가정해 봅시다. 문자 그대로 그 기능을 제공할 수는 없지만, 이 답변을 통해 해당 이해관계자에게 제품 출시 시간이 중요하다는 것을 알 수 있습니다. 마찬가지로 "이 시스템은 번개처럼 빨라야 합니다."라는 말은 성능이 매우 중요하다는 뜻이고, "다운타임 제로"는 문자 그대로의 의미가 아니라 해당 애플리케이션에서 가용성이 매우 중요하다는 것을 의미합니다. 유능한 소프트웨어 아키텍트는 이러한 과장된 표현의 행간을 읽어 이해관계자의 진정한 우려 사항을 파악합니다. 이는 두 번째 협상 기법으로 이어집니다.

협상에 들어가기 전에 가능한 한 많은 정보를 수집하십시오.

파커가 "파이브 나인(ive nines)"이라는 표현을 사용한 것은 시스템의 높은 가용성(구체적으로는 99.999%의 가동 시간)을 원한다는 의미입니다. 하지만 파커가 "파이브 나인"의 가용성이 실제로 무엇을 의미하는지 제대로 이해하지 못하고 있을 가능성도 있습니다.

연간 가동 중지 시간. 표 25-1은 가용성 9의 개수를 기준으로 한 가동 중지 시간을 보여줍니다.

표 25-1. 가용성의 9가지

연간 가동률(%)	연간 가동 중지 시간(일일)
(90.0%)	36일 12시간 (2.4시간)
99.0% (9가 두 개)	87시간 46분 (14분)
99.9% (세 개의 9)	8시간 46분 (86초)
99.99% (99.99%)	52분 33초 (7초)
99.999% (9의 5분의 1)	5분 35초 (1초)
99.9999% (99.9999%)	31.5초 (86밀리초)

'99.999% 가용성'은 연간 5분 35초의 다운타임, 즉 하루 평균 1초의 계획되지 않은 다운타임을 의미합니다. 이는 지나치게 높은 목표이며, 비용도 많이 들고, 특히 글로벌 거래가 이루어지지 않는 시간대를 고려한 시나리오 1에서는 불필요한 목표입니다. 따라서 '99.999%'라는 표현 대신 시간과 분(또는 이 경우에는 초) 단위로 목표를 설정하는 것이 훨씬 효과적입니다. 구체적인 수치와 측정 기준을 제시할 수 있기 때문입니다.

협상 시나리오 1에는 파커의 우려 사항을 확인하는 것("이 시스템에 있어 가용성이 매우 중요하다는 것을 이해합니다")이 포함되며, 그런 다음 협상을 99.9% 가용성이라는 극단적인 표현에서 벗어나 계획되지 않은 다운타임을 합리적인 시간(시간 및 분)으로 제한하는 방향으로 이끌어가는 것입니다.
(당신이 충분히 좋다고 생각하는) 99.9 ...

다른 모든 방법이 실패하면, 실질적인 비용과 시간 측면에서 상황을 설명하십시오.

이러한 협상 전략은 최후의 수단으로 남겨두는 것이 좋습니다. "그건 비용이 많이 들 거예요" 또는 "우리는 그럴 시간이 없어요"와 같은 말로 협상이 엉망이 되는 경우를 너무 많이 봐왔습니다. 돈과 시간(노력 포함)은 분명 협상에서 중요한 요소이지만, 이를 언급하기 전에 더 중요한 다른 이유와 합리적인 근거를 먼저 찾아보세요. 돈과 시간은 최후의 수단이어야 합니다. 이해관계자와 합의에 도달한 후에야 비용과 시간을 고려할 수 있습니다.

이러한 상황에서 중요한 협상 기법 또 하나는 다음과 같습니다.

요구사항이나 조건을 구체화해야 할 때는 "분할 정복" 전략을 사용하세요.

고대 중국의 무술가 손자는 『손자병법』에서 상대방에 대해 "상대의 군대가 뭉쳐 있다면, 그들을 분산시켜라"라고 썼습니다. 협상에서도 이러한 분할 통치 전략을 활용할 수 있습니다. 시나리오 1에서 파커는 새로운 거래 시스템에 대해 99.999%의 가용성을 요구하고 있습니다. 하지만 시스템 전체에 99.999%의 가용성이 정말 필요할까요? 요구 조건을 구체화하여 실제로 99.999%의 가용성이 필요한 특정 영역으로 범위를 좁힐 수 있습니다. 이렇게 하면 까다롭고 비용이 많이 드는 요구 사항의 범위를 줄일 수 있을 뿐 아니라 협상의 범위도 좁힐 수 있습니다.

다른 건축가와의 협상은 흔한 일입니다. 건축가들

이 프로젝트를 진행하면서 다른 건축가들과 협상해야 하는 경우는 드물지 않습니다.
두 명의 설계자가 사용할 프로토콜에 대해 의견이 일치하지 않는 시나리오 2를 생각해 보겠습니다.

시나리오 2

> 당신은 프로젝트에서 두 명의 아키텍트 중 선임이며, 서비스 그룹 간의 통신에 비동기 메시징이 성능과 확장성을 모두 향상시키는 데 적합한 접근 방식이라고 생각합니다. 그러나 프로젝트의 다른 아키텍트인 애디슨은 강력하게 반대합니다. 그는 REST가 메시징보다 항상 빠르고 확장성도 동등하기 때문에 더 나은 선택이라고 주장합니다. 그의 주장은 구글 검색과 인기 있는 생성형 AI 도구에 입력한 결과로 구성된 연구 자료에 불과합니다. 애디슨과의 열띤 논쟁은 이번이 처음도 아니고 마지막도 아닐 것입니다. 당신은 메시징이 올바른 해결책이라는 것을 애디슨에게 설득하고 싶습니다.

수석 건축가로서 당신은 애디슨에게 그들의 의견은 중요하지 않다고 말하고 무시할 수도 있습니다. 하지만 그렇게 하면 두 사람 사이의 적대감만 더욱 심화될 뿐이며, 프로젝트를 맡은 두 건축가 사이의 비협조적인 관계는 개발팀 전체에 부정적인 영향을 미칠 가능성이 매우 높습니다.

시위는 토론을 무용지물로 만든다는 것을 항상 기억하십시오.

REST와 메시징 중 어떤 것이 더 나은지 논쟁하기보다는, 애디슨에게 왜 이 특정 환경에서 메시징이 더 나은 선택인지 보여줘야 합니다. 모든 환경은 다르기 때문에 단순히 구글 검색만으로는 정확한 답을 얻기 어렵습니다.
하지만 실제 운영 환경과 유사한 조건에서 두 가지 옵션을 비교하고 애디슨에게 결과를 보여준다면 논쟁을 완전히 피할 수 있을지도 모릅니다.

지나치게 논쟁적이거나 감정적으로 대응하지 마십시오. 차분한 리더십과 명확하고 간결한 논리가 결합되면 협상에서 거의 항상 승리할 수 있습니다.

이는 적대적인 관계를 다루는 데 매우 효과적인 기법입니다. 감정이 격해지거나 분위기가 험악해지면 협상을 중단하는 것이 최선입니다. 양측 모두 진정된 후에 다시 협상을 시도하십시오. 건축가들 사이에서도 의견 충돌은 흔히 발생하지만, 침착함을 유지하고 리더십을 보여준다면 상대방은 대개 양보할 것입니다.

개발자와의 협상: 유능한 소프트웨어 아

키텍트는 협업을 통해 팀의 존경을 얻습니다. 그렇게 하면 개발팀에 무언가를 요청할 때 논쟁이나 불만이 생길 가능성이 훨씬 줄어듭니다.

24장 에서 보셨듯이 개발팀과 협업하는 것은 때때로 어려울 수 있습니다.
개발팀이 아키텍처(또는 아키텍트)와 단절되었다고 느낄 때, 의사 결정 과정에서 소외감을 느끼는 경우가 많습니다. 이는 전형적인 상아탑식 아키텍처 안티패턴의 예입니다. 상아탑에 갇힌 아키텍트는 개발팀의 의견이나 우려를 고려하지 않고 위에서 지시만 내립니다. 이는 보통 팀원들이 아키텍트에 대한 존경심을 잃게 만들고, 결국 팀 전체의 역학 관계가 무너지는 결과를 초래할 수 있습니다. 이러한 상황을 해결하는 데 도움이 될 수 있는 협상 기법 중 하나는 항상 자신의 결정에 대한 타당성을 제시하는 것입니다.

개발자들이 아키텍처 설계를 채택하거나 특정 작업을 수행하도록 설득할 때는 "일방적으로 지시"하기보다는 타당한 이유를 제시해야 합니다.

어떤 작업을 해야 하는 이유를 제시하면 개발자들이 동의할 가능성이 더 높아집니다. 예를 들어, 기존의 n계층 아키텍처에서 간단한 쿼리를 구현하는 것에 대해 아키텍트와 개발자가 나눈 다음 대화를 생각해 보세요.

> 아키텍트: "그 결정을 내리려면 비즈니스 부서를 거쳐야 합니다."
> 개발자: "저는 동의하지 않습니다. 데이터베이스를 직접 호출하는 것이 훨씬 빠릅니다."

이 대화에는 몇 가지 문제가 있습니다. 첫째, 설계자가 "반드시 해야 한다"라는 표현을 사용한 것에 주목하세요. 이러한 명령조는 상대를 깎아내릴 뿐만 아니라 협상(또는 대화)을 시작하는 최악의 방법 중 하나입니다. 개발자는 비즈니스 계층을 거치는 것이 더 느리고 시간이 오래 걸린다는 이유를 제시합니다.

시간이 더 필요해요.

이제 다른 접근 방식을 생각해 보겠습니다.

> 아키텍트: "변경 관리가 저희에게 가장 중요하기 때문에 폐쇄형 계층 아키텍처를 구축했습니다. 즉, 데이터베이스에 대한 모든 호출은 비즈니스 계층에서만 이루어져야 합니다."
>
> 개발자: "알겠습니다. 그런데 그렇다면 간단한 쿼리에서 발생하는 성능 문제는 어떻게 해결해야 할까요?"

여기서 설계자는 모든 호출이 비즈니스 계층을 거쳐야 한다는 요구 사항을 정당화하고 있습니다.
먼저 타당성을 제시하는 것이 항상 좋은 접근 방식입니다. 대부분의 사람들은 자신이 동의하지 않는 내용을 듣는 순간 듣기를 멈추는 경향이 있습니다. 요구사항을 제시하기 전에 이유를 설명하면 개발자는 건축가의 설명을 경청하게 될 것입니다.

건축가는 이러한 요구 사항을 표현할 때 개인적인 감정이 개입되지 않도록 했습니다. "반드시 이렇게 해야 한다"라고 말하는 대신 "이것은 ~을 의미한다"라고 표현함으로써 요구 사항을 단순한 사실 진술로 만들었습니다. 이제 개발자의 반응을 살펴보세요. 계층형 아키텍처의 제약 조건에 반대하는 대신, 개발자는 간단한 함수 호출의 성능 향상에 대한 질문을 던집니다. 이제 두 사람은 협력적인 대화를 통해 아키텍처의 폐쇄 계층 구조를 유지하면서도 간단한 쿼리를 더 빠르게 처리할 수 있는 방법을 찾을 수 있습니다.

또 다른 효과적인 협상 전략은 개발자가 스스로 해결책을 찾도록 하는 것입니다. 예를 들어, 아키텍트인 당신이 프레임워크 X와 프레임워크 Y 두 가지 중에서 선택해야 한다고 가정해 보겠습니다. 프레임워크 Y는 시스템의 보안 요구 사항을 충족하지 못하므로 당신은 당연히 프레임워크 X를 선택합니다. 하지만 팀의 한 개발자가 강력하게 반대하며 프레임워크 Y가 여전히 더 나은 선택이라고 주장합니다. 이때 논쟁을 벌이는 대신, 당신은 개발자에게 프레임워크 Y가 보안 문제를 어떻게 해결하는지 보여줄 수 있다면 프레임워크 Y를 사용하겠다고 제안합니다.

다음에는 두 가지 상황 중 하나가 발생할 것입니다. 첫 번째는 개발자가 프레임워크 Y가 보안 요구 사항을 충족한다는 것을 입증하려고 시도하지만 실패하는 것입니다. 실패를 통해 개발자는 왜 팀이 이 프레임워크를 사용할 수 없는지 직접적으로 이해하게 됩니다. 그리고 스스로 해결책을 찾아냈기 때문에 프레임워크 X를 사용하기로 한 결정에 대한 개발자의 동의를 자연스럽게 얻게 됩니다. 사실상 개발자의 결정에 맡기는 셈입니다.
이것은 긍정적인 결과입니다. 두 번째 옵션은 개발자가 프레임워크 Y를 사용하여 보안 요구 사항을 충족하는 방법을 찾아 당신에게 보여주는 것입니다. 이것 또한 긍정적인 결과입니다. 이 경우 당신은 프레임워크 Y에 대한 평가에서 무언가를 놓쳤고, 이제 문제에 대한 더 나은 해결책을 얻게 된 것입니다 (그리고 개발자는 여전히 의사 결정 과정에 참여했다고 느낄 것입니다).

개발자가 당신의 결정에 동의하지 않는다면, 그들이 스스로 해결책을 찾도록 하십시오.

개발자들은 유용한 지식을 가진 똑똑한 사람들입니다. 아키텍트는 개발팀과의 협업을 통해 그들의 존경을 얻고 더 나은 솔루션을 찾는 데 도움을 받을 수 있습니다. 개발자들이 아키텍트를 존중할수록 아키텍트는 그들과 협상하기가 더 쉬워집니다.

소프트웨어 아키텍트의 리더십

소프트웨어 아키텍트는 개발팀이 아키텍처를 구현하는 과정을 이끌어가는 리더이기도 합니다. 효과적인 소프트웨어 아키텍트가 되기 위한 핵심 역량 중 약 50%는 원활한 소통과 리더십을 포함한 뛰어난 대인관계 능력에 달려 있다고 생각합니다. 이 섹션에서는 소프트웨어 아키텍트를 위한 리더십 기법에 대해 논의합니다.

건축의 4C

변화는 끊임없는 것이라고 우리는 여러 번 강조해 왔습니다. 특히 비즈니스 프로세스, 기술, 심지어 아키텍처 자체에 이르기까지 점점 더 복잡해지는 변화의 경우 이 말은 더욱 진실입니다. 이 책 3부 에서 보셨듯이 , 일부 아키텍처는 매우 복잡합니다. 만약 아키텍처가 99.9999%의 가용성(six nines)을 지원해야 한다면, 이는 하루에 약 86밀리초, 또는 연간 31.5초의 계획되지 않은 다운타임을 허용하는 것과 같습니다. 이러한 복잡성을 본질적 복잡성(essential complexity)이라고 합니다. 다시 말해, "우리는 어려운 문제에 직면해 있습니다."

건축가(및 개발자)는 솔루션, 다이어그램 및 문서에 불필요한 복잡성을 추가하는 함정에 쉽게 빠질 수 있습니다. 닐의 말을 인용하자면 다음과 같습니다.

> 개발자들은 마치 불나방이 불길에 이끌리듯 복잡성에 매료되는데, 그 결과는 대개 같습니다.

그림 25-1은 매우 큰 규모의 글로벌 은행 백엔드 처리 시스템의 주요 정보 흐름을 나타낸 도표입니다. 이 시스템이 필연적으로 복잡해야 할까요? 다시 말해, 복잡해야만 할까요? 아무도 알 수 없습니다. 왜냐하면 설계자가 시스템을 복잡하게 만들었기 때문입니다. 이러한 복잡성을 '우발적 복잡성'이라고 합니다. 간단히 말해, "문제를 어렵게 만들었다"는 뜻입니다. 설계자는 때때로 너무 간단해 보이는 것에 자신의 능력을 증명하거나, 의사 결정 과정에서 항상 자신의 위치를 확보하거나, 심지어는 직업 안정성을 유지하기 위해 복잡성을 추가합니다. 이유가 무엇이든, 복잡할 필요가 없는 것에 우발적 복잡성을 도입하는 것은 존경을 잃고 비효율적인 리더이자 설계자가 되는 가장 좋은 방법 중 하나입니다.

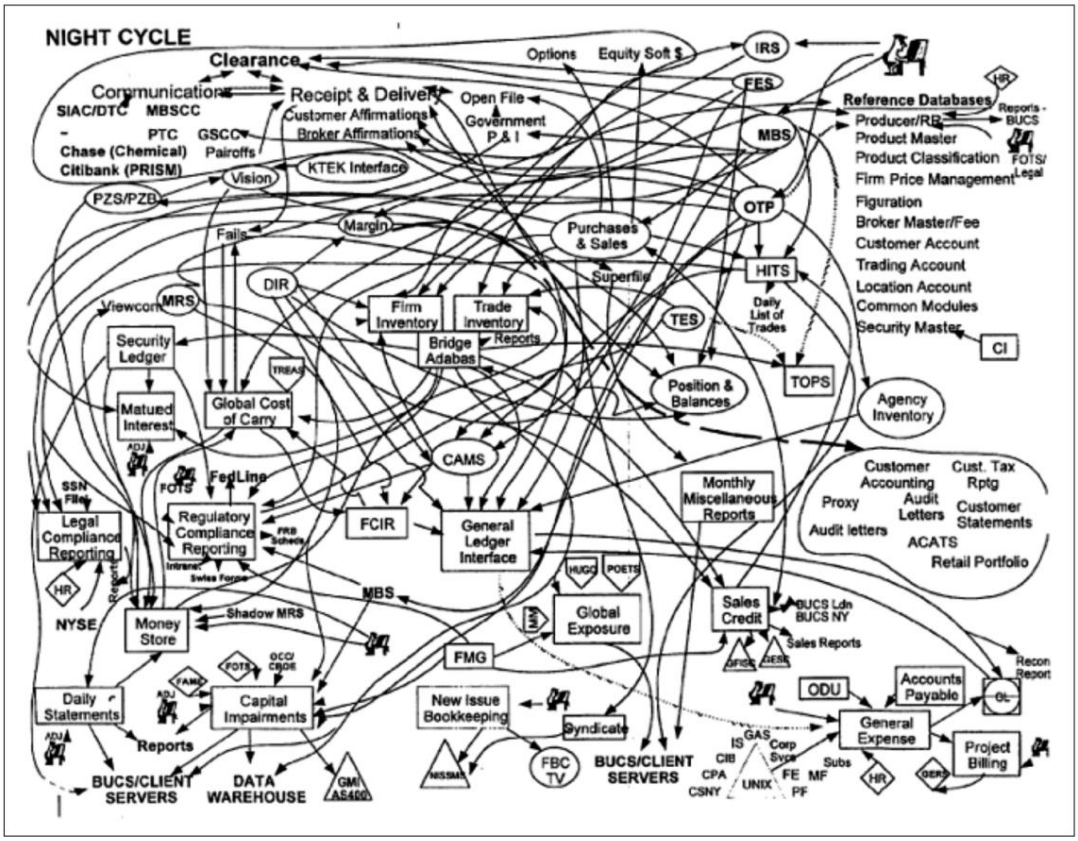

그림 25-1. 문제에 우발적인 복잡성을 도입하기

불필요한 복잡성을 피하기 위해 우리는 아키텍처 리더십의 4C, 즉 소통, 협업, 명확성, 간결성을 활용합니다. 그림 25-2 (다이어그램 작성의 C4 모델의 4C와 혼동하지 마십시오)에서 볼 수 있듯이, 이러한 요소들은 팀 내에서 효과적인 소통가이자 협업자를 만드는 데 기여합니다.

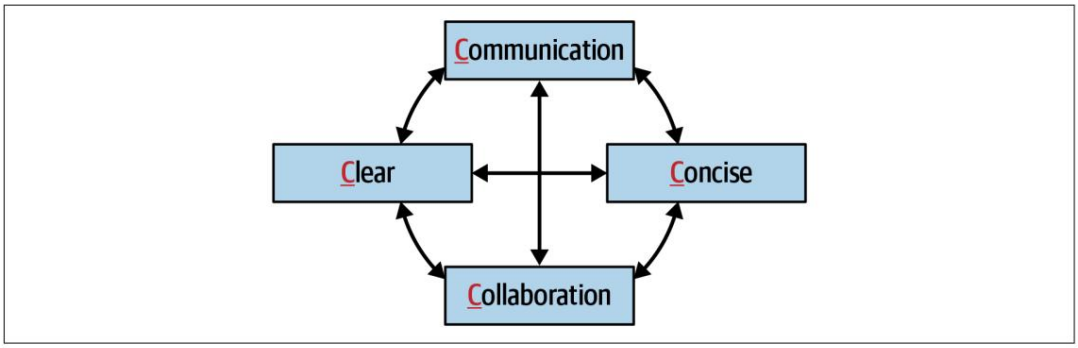

그림 25-2. 건축의 4가지 핵심 요소

리더, 조력자, 협상가로서 소프트웨어 아키텍트에게는 명확하고 간결하게 소통하는 능력이 필수적입니다. 또한, 다른 구성원들과 협업할 수 있는 능력 역시 매우 중요합니다.

개발자, 비즈니스 이해관계자 및 다른 아키텍트. 4C에 집중하면 아키텍트는 팀의 존경을 얻고 프로젝트에서 질문, 조언, 멘토링, 코칭 및 리더십을 위한 핵심 인물이 될 수 있습니다.

실용적인 동시에 비전 있는 사람이 되어

야 합니다 . 효과적인 소프트웨어 아키텍트는 실용적인 동시에 비전 있는 사람이 되어야 합니다. 이러한 균형을 이루는 것은 말처럼 쉽지 않으며, 상당한 수준의 성숙도와 풍부한 경험을 필요로 합니다.

선구자란 상상력이나 지혜로 미래를 생각하거나 계획하는 사람을 말합니다. 선구자라는 것은 문제에 전략적 사고를 적용하는 것을 의미하는데, 이는 바로 건축가들이 하는 일입니다. 건축가들은 미래를 계획하고 자신들이 건설하는 건축물이 오랫동안 유효하고 유용하게 유지되도록 합니다. 그러나 건축가들은 계획과 설계 과정에서 지나치게 이론적인 접근을 하여 실행하기 어렵거나 심지어 이해하기조차 어려운 해결책을 제시하는 경우가 종종 있습니다.

이제 동전의 다른 면, 즉 실용주의를 생각해 보겠습니다. 실용주의란 이론적인 고려보다는 실질적인 고려에 기반하여 사물을 합리적이고 현실적으로 다루는 것을 의미합니다. 건축가는 비전을 제시해야 하지만, 동시에 실용적이고 현실적인 해결책을 적용해야 합니다. 건축 솔루션을 구상할 때 실용주의적이라는 것은 다음 사항들을 고려하는 것을 의미합니다.

- 예산 제약 및 기타 비용 관련 요소
- 시간 제약 및 기타 시간 관련 요소
- 개발팀의 기술 역량 및 숙련도 • 각 아키텍처 결정의 장단점 및 영향 • 제안된 설계 또는 솔루션의 기술적 한계

훌륭한 소프트웨어 설계자는 문제를 해결할 때 실용주의와 상상력 및 지혜 사이의 균형을 맞추려고 노력합니다(그림 25-3 참조).

예를 들어, 원인을 알 수 없는 갑작스럽고 상당한 동시 접속 사용자 부하 증가에 직면한 아키텍트를 생각해 보세요. 선견지명이 있는 사람은 데이터베이스를 분리하고 복잡한 데이터 메시를 구축하여 시스템의 탄력성을 향상시키는 정교한 방법을 고안할 수 있습니다. 데이터 메시 는 분석 데이터와 트랜잭션 데이터를 분리하기 위해 도메인별로 분할된 분산 데이터베이스 모음입니다. 이론적으로는 이러한 접근 방식이 가능할 수 있지만, 현실적으로는 해결책에 이성과 실용성을 적용해야 합니다. 예를 들어, 회사가 이전에 데이터 메시를 사용한 적이 있는지, 데이터 메시를 사용할 때의 장단점은 무엇인지, 그리고 이 해결책이 정말로 문제를 해결할 수 있는지 등을 고려해야 합니다.

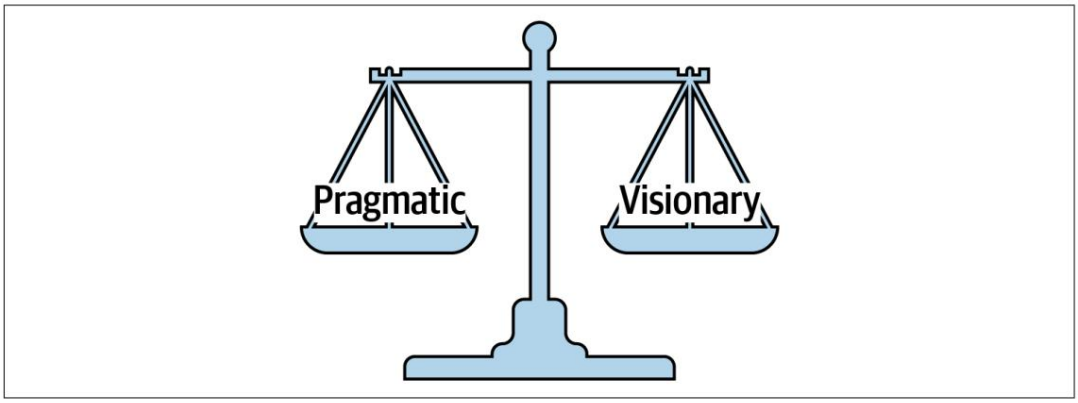

그림 25-3. 훌륭한 건축가는 실용성과 비전 사이의 균형을 유지한다.

실용성과 비전 사이에서 균형을 잘 유지하는 것은 아키텍트로서 존경을 얻는 훌륭한 방법입니다. 비즈니스 이해관계자들은 제약 조건 내에서 실현 가능한 비전 있는 솔루션을 높이 평가하고, 개발자들은 이론적인 솔루션보다는 실용적인 솔루션을 구현할 수 있다는 점을 높이 평가합니다.

실용적인 아키텍트라면 먼저 시스템의 탄력성을 제한하는 요소를 살펴볼 것입니다. 병목 현상을 찾아 분리하는 것이 이 문제에 대한 실질적인 첫 번째 접근 방식입니다. 데이터베이스가 호출되는 서비스나 필요한 외부 소스와 관련하여 병목 현상을 일으키고 있습니까? 그렇다면 데이터베이스 호출 횟수를 줄이기 위해 일부 데이터를 캐싱할 수 있을까요?

솔선수범하여 팀을 이끌어라 . 형편없는 소프

트웨어 아키텍트는 자신의 직위를 이용해 사람들에게 일을 시키는 경향이 있다.
유능한 소프트웨어 아키텍트는 솔선수범하여 사람들을 이끌어 일을 하게 만든다.
다시 말하지만, 이 모든 것은 개발팀, 비즈니스 이해관계자, 그리고 조직 내 다른 사람들(예: 운영 책임자, 개발 관리자, 제품 소유자)의 존경을 얻는 것에 관한 것입니다.

"직책이 아니라 솔선수범하라"는 고전적인 이야기는 군사 전투에서 하사관을 지휘하는 대위와 관련된 것입니다. 병사들과 거의 떨어져 있는 대위는 모든 병사들을 이끌고 특히 험준한 언덕을 점령하라고 명령합니다.

의심으로 가득 찬 병사들은 하급 장교에게 확인을 구한다.
상황을 파악한 하사는 고개를 살짝 끄덕였고, 병사들은 곧바로 자신감 넘치는 모습으로 언덕을 점령하기 위해 앞으로 나아갔다.

이 이야기의 교훈은 사람들을 이끄는 데 있어서 직급과 직함은 그다지 중요하지 않다는 것입니다. 컴퓨터 과학자 제럴드 와인버그는 "어떤 문제든 결국 사람과 관련된 문제다"라는 유명한 말을 남겼습니다. 대부분의 사람들은 기술적인 문제를 해결하는 것은 사람과의 소통 능력과는 전혀 상관없고 오로지 기술력에만 달려 있다고 생각합니다.

지식. 기술적 지식은 분명히 필요하지만, 문제 해결의 일부분일 뿐입니다.

예를 들어, 아키텍트가 운영 환경에서 발생한 문제를 해결하기 위해 개발자 팀과 회의를 한다고 가정해 봅시다. 개발자 중 한 명이 제안을 하자, 아키텍트는 "그건 정말 어리석은 생각이야."라고 말합니다. 그러면 그 개발자는 더 이상 제안을 하지 않을 뿐만 아니라, 다른 개발자들도 감히 아무 말도 하지 못합니다. 아키텍트가 팀 전체의 협업을 단절시켜 버린 것입니다.

건축가와 개발자 간의 또 다른 대화 내용을 살펴보겠습니다.

> 개발자: "그럼 이 성능 문제를 어떻게 해결할까요?"
> 건축가: "캐시를 사용하면 문제가 해결될 겁니다."
> 개발자: "나한테 이래라저래라 하지 마."
> 건축가: "제가 말씀드리는 건, 그게 문제를 해결해 줄 거라는 겁니다."

이는 협업 없이 소통하는 좋은 예입니다. "당신이 해야 할 일은 이것입니다"라는 말을 사용함으로써 설계자는 협업을 차단하고 있습니다. 이제 수정된 접근 방식을 생각해 보세요.

> 개발자: "그럼 이 성능 문제를 어떻게 해결할까요?"
> 건축가: "캐시를 사용하는 것을 고려해 보셨나요? 그러면 문제가 해결될 수도 있습니다."
> 개발자: "음, 아니요, 그 부분은 생각 못 했네요. 어떻게 생각하세요?"
> 건축가: "음, 여기에 캐시를 설치하면…"

"혹시 고려해 보셨나요?"라는 표현은 명령을 질문으로 바꾸어 개발자에게 통제권을 되돌려주고, 아키텍트와 협력적인 대화를 나눌 수 있도록 합니다. 언어를 어떻게 사용하는지는 협업 환경을 구축하는 데 매우 중요합니다.

건축가로서 협업을 이끌어간다는 것은 단순히 개인적으로 다른 사람들과 어떻게 협업하는지에 대한 것만이 아닙니다. 팀 구성원 간의 협업을 촉진하는 것 또한 중요합니다. 팀의 역학 관계를 관찰하고 이 대화에서처럼 특정 상황에 주목하세요. 만약 팀원이 요구적이거나 거만한 언어를 사용하는 것을 목격한다면, 그 사람을 따로 불러 협력적인 언어를 사용하는 방법을 지도하세요. 이는 팀 분위기를 개선할 뿐만 아니라 팀원들이 서로를 존중하는 데에도 도움이 될 것입니다.

누군가에게 하기 싫어하는 일을 시켜야 할 때, 요청을 부탁으로 바꾸는 것이 때로는 가장 좋은 방법입니다. 일반적으로 사람들은 지시받는 것을 싫어하지만, 남을 돕고 싶어 합니다. 바쁜 스프린트 기간 동안 아키텍처 리팩토링 작업을 진행하는 아키텍트와 개발자 간의 다음 대화를 생각해 보세요!

> 아키텍트: "결제 서비스를 다섯 개의 개별 서비스로 분리해 주셔야 합니다. 각 서비스에는 스토어 크레딧, 신용 카드, 페이팔, 기프트 카드, 포인트 적립 등 저희가 허용하는 각 결제 유형에 대한 기능이 포함되어야 합니다. 이렇게 하면 웹사이트의 장애 복구 및 확장성이 향상될 것입니다. 작업 시간은 그리 오래 걸리지 않을 겁니다."
> 개발자: "죄송하지만 이번 버전 작업은 너무 바빠서 그럴 시간이 없어요. 정말 안 될 것 같습니다."

건축가: "이건 중요한 일이고 이번 프로젝트에서 꼭 해야 해요."
개발자: "죄송하지만, 제가 할 수는 없습니다. 다른 개발자분들이 하실 수 있을지도 모르겠네요. 저는 지금 너무 바빠서요."

아키텍트가 더 나은 내결함성과 확장성을 제공한다고 정당화했음에도 불구하고, 개발자는 즉시 그 작업을 거절합니다. 아키텍트는 개발자에게 자신이 너무 바빠서 할 수 없는 일을 시키고 있는 것이며, 심지어 그 요구에는 개발자의 이름조차 언급되지 않았습니다! 이제 요청을 호의로 바꾸는 기법을 생각해 보세요.

아키텍트: "안녕하세요, 스리다르. 제가 정말 곤란한 상황에 처했어요. 결제 서비스를 결제 유형별로 분리해서 장애 복구 및 확장성을 높여야 하는데, 너무 오래 미뤄왔네요. 이번 스프린트에 이 작업을 포함시킬 수 있을까요? 정말 큰 도움이 될 거예요."

개발자 (잠시 멈춤): "이번 반복 작업은 정말 바쁘지만, 제가 할 수 있는 일이 있는지 알아보겠습니다."
건축가: "고마워요, 스리다르. 정말 고맙습니다. 신세를 졌네요."
개발자: "걱정 마세요. 이번 개발 주기 안에 완료되도록 하겠습니다."

첫째, 상대방의 이름을 부르는 것은 딱딱하고 비인간적인 업무적 요구가 아닌, 더욱 개인적이고 친근한 대화를 만들어냅니다. 대화나 협상 중에 상대방의 이름과 적절한 대명사를 사용하는 것은 존중과 건강한 관계를 구축하는 데 도움이 됩니다. 사람들은 자신의 이름을 듣는 것을 좋아할 뿐만 아니라, 이는 친밀감을 형성하는 데에도 효과적입니다. 상대방의 이름을 자주 사용함으로써 이름을 기억하는 연습을 해보세요.
이름 발음이 어렵다면 정확한 발음을 찾아보고 완벽해질 때까지 연습하세요. 누군가 자신의 이름을 말하면 우리는 그 이름을 따라 말하고 제대로 발음하고 있는지 확인하는 습관이 있습니다. 만약 잘못 발음했다면 제대로 발음할 때까지 이 과정을 반복합니다.

둘째, 이전 대화에서 건축가는 자신들이 "정말 곤란한 상황"에 처해 있으며 서비스를 분리하는 것이 "큰 도움이 될 것"이라고 인정했습니다. 이 방법이 항상 통하는 것은 아니지만, 타인을 돕고자 하는 인간의 기본적인 욕구를 자극하는 것은 첫 번째 대화보다 성공 확률이 높습니다. 다음에 비슷한 상황에 직면했을 때 시도해 보세요.

처음 만나는 사람이나 가끔씩만 보는 사람에게 인사할 때 효과적인 리더십 기법 중 하나는 악수를 하고 눈을 마주치는 것입니다. 악수는 중세 시대부터 이어져 온 중요한 대인 관계 기술입니다. 악수를 통해 두 사람은 서로 친구 사이임을 확인하고 유대감을 형성할 수 있습니다. 다만 악수에는 문화적 차이가 있을 수 있다는 점을 유념해야 합니다. 예를 들어 미국, 영국, 유럽, 호주에서는 악수가 인사로 통용되는 반면, 다른 문화권에서는 다른 인사 방식을 사용합니다(일본에서는 절을 하고 절하는 타이밍이 중요합니다). 그럼에도 불구하고, 간단한 악수조차 제대로 하기 어려울 수 있습니다.

악수는 단단해야 하지만 지나치게 세게 해서는 안 됩니다. 상대방의 눈을 바라보세요. 악수하는 동안 시선을 피하는 것은 무례한 행동으로 여겨지며, 대부분의 사람들은 이를 알아차립니다. 또한 악수를 너무 오래 하지 마세요. 2~3초면 충분합니다.

필요한 건 그게 전부입니다. 악수를 너무 과하게 할 필요도 없습니다. 매일 아침 사무실에 와서 모든 사람과 악수를 한다면, 사람들을 불편하게 만들기에 충분합니다. 하지만 운영 책임자와의 월례 회의는 자리에서 일어나 "안녕하세요, 루스 씨, 다시 뵙게 되어 반갑습니다."라고 인사하고 짧고 단단한 악수를 나누기에 완벽한 기회입니다. 언제 악수를 해야 하고 언제 하지 말아야 하는지를 아는 것은 복잡한 대인 관계 기술의 일부입니다.

리더로서 모든 직급의 사람들 사이의 개인적인 경계를 존중하고 유지하는 데 주의를 기울여야 합니다. 악수는 전문적인 방식으로 신체적 유대감을 형성하는 방법입니다. 이것이 좋은 점이라고 생각해서, 어떤 사람들은 포옹이 더 좋고 유대감을 강화한다고 생각합니다. 하지만 그렇지 않습니다. 어떤 환경이든 전문적인 상황에서 포옹은 상대방을 불편하게 만들 수 있으며, 심지어 직장 내 괴롭힘의 한 형태로 이어질 수도 있습니다. 이는 직장 내 일반적인 행동이나 동료와 함께 출장을 갈 때도 마찬가지입니다. 포옹은 삼가고, 상식적인 전문적인 행동 수칙을 준수하며, 악수를 하는 것이 좋습니다.

훌륭한 리더는 팀에서 개발자들이 질문이나 문제가 있을 때 가장 먼저 찾는 사람이 되어야 합니다. 효과적인 소프트웨어 아키텍트는 직책이나 팀 내 역할에 관계없이 기회를 포착하고 솔선수범하여 팀을 이끌어야 합니다. 누군가 기술적인 문제로 어려움을 겪고 있다면, 적극적으로 나서서 도움이나 지침을 제공하세요. 비기술적인 상황에서도 마찬가지입니다. 팀원이 뭔가 문제가 있는 듯 우울하고 힘들어 보이는 모습으로 출근한다면, 효과적인 소프트웨어 아키텍트는 먼저 말을 걸어볼 수 있습니다. "안토니오, 나 커피 마시러 갈 건데, 같이 갈까?"

산책하는 동안 괜찮은지 물어볼 수 있습니다. 이는 보다 개인적인 대화로 이어질 수 있는 계기가 되며, 최선의 경우 멘토링이나 코칭의 기회로까지 발전할 수 있습니다. 하지만 효과적인 리더는 언어적, 비언어적 신호(얼굴 표정이나 몸짓 등)에도 주의를 기울이고, 언제 물러나야 할지 판단할 줄 알아야 합니다.

팀에서 핵심 인물로 자리매김하는 또 다른 방법은 특정 기술이나 기법에 대한 간단한 "점심 식사 겸 학습" 세션을 정기적으로 주최하는 것입니다. 디자인 패턴이나 최신 프로그래밍 언어 릴리스의 새로운 기능 등을 다룰 수 있습니다. 이 책을 읽고 계신다면 분명 다른 사람들이 갖지 못한 특별한 기술이나 지식을 가지고 계실 것입니다. 이러한 세션을 주최하는 것은 개발자들에게 유용한 정보를 제공하거나 기술적 역량을 보여주는 것뿐만 아니라, 멘토링 및 발표 능력을 연습하고 리더이자 멘토로서의 이미지를 구축할 수 있는 기회이기도 합니다.

개발팀과의 통합

건축가의 일정표는 그림 25-4에서처럼 회의가 겹치는 경우가 많습니다 . 그렇다면 건축가는 어떻게 팀에 합류하여 팀원들을 지도하고 멘토링하며, 질문이나 우려 사항에 답변할 시간을 확보할 수 있을까요?

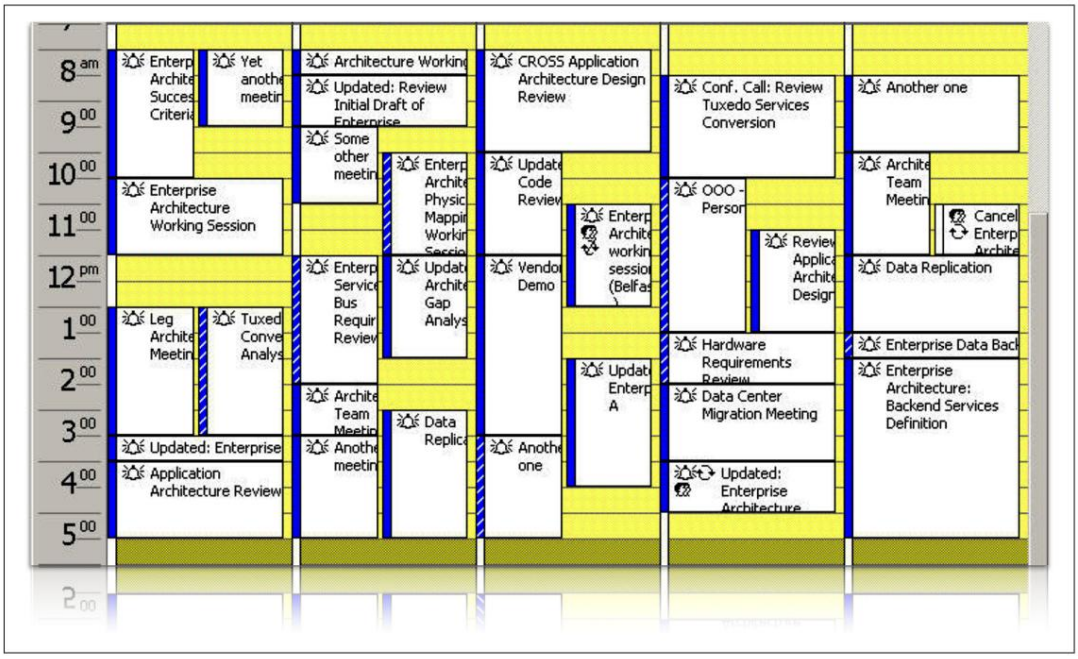

그림 25-4. 소프트웨어 건축가의 일반적인 일정표

불행히도 잦은 회의는 불가피한 악입니다. 중요한 것은 회의를 통제하여 팀을 위한 시간을 확보하는 것입니다. 그림 25-5 에서 볼 수 있듯이 회의에는 두 가지 유형이 있습니다. 하나는 상대방이 초대하는 회의이고, 다른 하나는 자신이 소집하는 회의입니다.

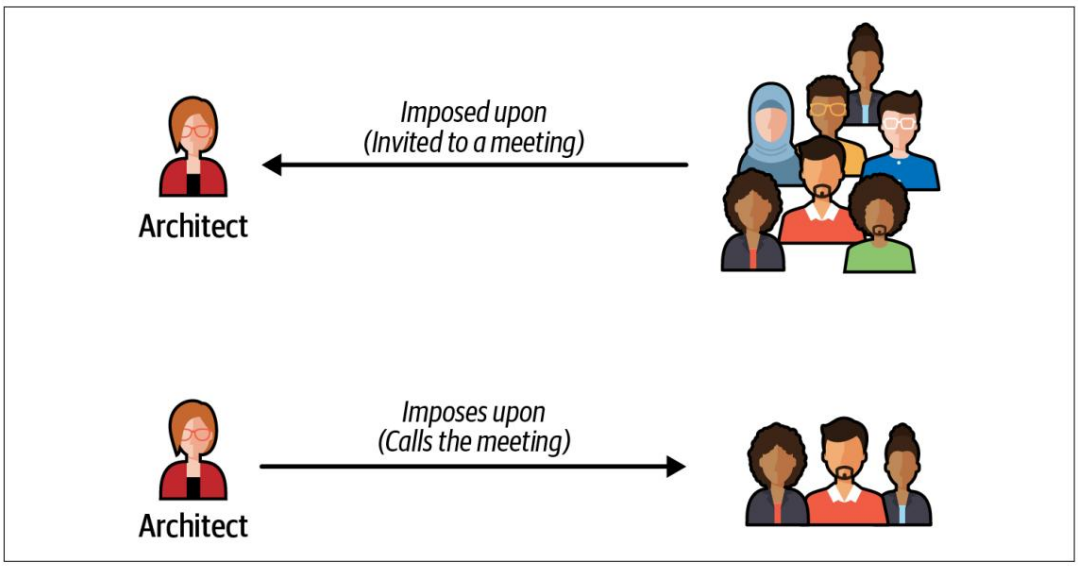

그림 25-5. 회의 유형

다른 사람이 소집하는 회의는 통제하기가 가장 어렵습니다. 소프트웨어 아키텍트는 다양한 이해관계자들과 소통하고 협업해야 하기 때문에, 실제로 참석이 필요하지 않은 회의까지 포함하여 거의 모든 회의에 초대받습니다. 회의에 초대받았을 때는 주최자에게 왜 참석해야 하는지 물어보세요. 단순히 진행 상황을 알려주기 위한 것이라면 회의록이 있으니 걱정하지 마세요. 회의에 참석해야 하는 이유를 물어보면 어떤 회의에 참석하고 어떤 회의는 건너뛸지 결정하는 데 도움이 됩니다.

회의 안건을 살펴보는 것도 도움이 됩니다. 또한, 회의 전체에 참석해야 하는지, 아니면 특정 주제를 논의하는 부분에만 참석해야 하는지 확인하세요. 해당 안건이 끝나면 자리를 뜰 수 있나요? 개발팀의 문제 해결을 도울 수 있는 시간을 회의에서 낭비하지 마세요.

회의에 참석해야 할 필요가 있는지 여부를 판단하는 데 도움이 되도록 회의 안건을 미리 요청하세요.

개발자나 기술 리더가 회의에 초대되었을 때, 특히 당신과 기술 리더 모두 초대되었다면, 그들을 대신하여 참석하는 것을 고려해 보세요. 이렇게 하면 팀이 당면한 업무에 집중할 수 있습니다. 유용한 팀원들의 회의 참석을 미루는 것은 당신이 회의에 참여하는 시간을 늘릴 수는 있지만, 개발팀의 생산성을 높이고 당신에 대한 존경심을 강화하는 데 도움이 됩니다.

건축가로서 때때로 다른 사람들에게 회의를 요청해야 할 때가 있지만, 이는 당신이 통제할 수 있는 부분이므로 회의 횟수를 최소한으로 유지해야 합니다. 회의 안건을 정하고 이를 철저히 지키세요. 모든 사람에게 중요하지 않은 다른 문제로 회의가 중단되지 않도록 하세요. 회의를 소집하는 것이 팀의 업무에 지장을 주는 것보다 더 중요한지 스스로에게 질문해 보세요. 예를 들어, 중요한 정보를 전달하는 것이라면 회의를 소집하는 대신 이메일을 보내는 것으로 충분하지 않을까요? 하지만 개발팀과 회의를 해야 한다면, 개발자들의 업무 시간에 방해가 되지 않도록 아침 일찍, 점심 식사 직후 또는 퇴근 시간에 회의를 잡는 것이 좋습니다.

개발자의 몰입 상태(Flow State) 란 개발자들이 특정 문제에 완전히 몰두하여 집중력과 창의력을 극대화하는 상태를 말합니다. 예를 들어, 개발자가 특히 어려운 알고리즘이나 코드를 작업할 때 몇 시간이 몇 분처럼 느껴질 수 있습니다. 팀의 생산성을 나타내는 몰입 상태에 세심하게 주의를 기울이고 이를 방해하지 않도록 하세요. 몰입 상태에 대한 자세한 내용은 미하이 칙센트미하이의 저서 『몰입: 최적 경험의 심리학』(Harper Perennial, 2008)에서 확인할 수 있습니다.

개발팀과 친분을 쌓는 또 다른 좋은 방법은 현장에서 근무할 때 함께 앉아 있는 것입니다. 팀과 떨어져 칸막이 책상에 앉아 있는 것은 "나는 특별한 존재이고 방해받고 싶지 않다"는 메시지를 전달하는 것과 같습니다. 반면 팀과 함께 앉아 있는 것은 "나는 팀의 중요한 일원이며 질문이나 우려 사항에 대해 언제든 도움을 줄 준비가 되어 있다"는 메시지를 전달합니다. 현장에 있지만 개발팀과 함께 앉을 수 없는 경우에는, 사무실을 돌아다니며 가능한 한 자주 모습을 드러내는 것이 좋습니다. 항상 다른 층에 있거나 사무실에만 틀어박혀 있는 아키텍트는 팀을 제대로 이끌 수 없습니다. 아침, 점심 식사 후, 또는 퇴근 후 시간을 내어 대화를 나누고, 문제를 해결하고, 질문에 답하고, 기본적인 코칭과 멘토링을 제공하세요. 개발자들은 이러한 소통 방식을 높이 평가하고, 업무 시간 중에 시간을 내어준 것에 대해 고마움을 느낄 것입니다. 다른 이해관계자에게도 마찬가지입니다. 커피를 사러 가는 길에 운영 책임자에게 인사하는 것만으로도 원활한 소통을 유지하는 데 큰 도움이 됩니다.

원격 근무 환경과 같이, 개발팀과 직접 만나거나 돌아다니며 소통하기 어려운 경우가 있습니다. 이러한 상황은 협업을 훨씬 어렵게 만듭니다. 원격 팀 관리 방법에 대한 자세한 내용은 재키 리드의 저서 『커뮤니케이션 패턴(Communication Patterns)』(O'Reilly, 2023)을 참고하시기를 강력히 추천합니다. 특히 4부는 원격 팀 관리에 대한 내용으로 구성되어 있습니다.

요약

이 장에서 소개하는 협상 및 리더십 팁은 소프트웨어 아키텍트가 개발팀 및 기타 이해관계자와 더 나은 협력 관계를 구축하는 데 도움이 되도록 고안되었습니다. 이러한 기술은 필수적입니다. 하지만 저희 말만 믿지 마시고 , 미국의 제26대 대통령 시어도어 루스벨트 의 말도 들어보십시오.

> 성공 공식에서 가장 중요한 단일 요소는 사람들과 잘 지내는 방법을 아는 것입니다.
>
> —시어도어 루스벨트