제2장

건축적 사고

아키텍처적 사고란 건축가의 시각, 즉 건축적인 관점에서 사물을 바라보는 것을 의미합니다. 특정 변경 사항이 전체 확장성에 어떤 영향을 미칠지 이해하고, 시스템의 각 부분이 어떻게 상호 작용하는지 주의 깊게 살피며, 주어진 상황에 가장 적합한 서드파티 라이브러리와 프레임워크를 파악하는 것 모두 아키텍처적 사고의 예입니다.

건축가처럼 생각하려면 먼저 소프트웨어 아키텍처가 무엇인지, 그리고 아키텍처와 디자인의 차이점을 이해해야 합니다. 그다음으로는 다른 사람들이 보지 못하는 해결책과 가능성을 볼 수 있는 폭넓은 지식을 갖춰야 하고, 비즈니스 동인의 중요성과 그것이 아키텍처적 고려 사항으로 어떻게 이어지는지 이해해야 하며, 다양한 솔루션과 기술 간의 장단점을 이해하고 분석하고 조화시킬 수 있어야 합니다.

이 장에서는 건축가처럼 생각하는 이러한 측면들을 살펴봅니다.

건축과 디자인의 차이점

잠시 시간을 내어 꿈에 그리던 집을 마음속으로 그려보세요. 몇 층짜리 집인가요? 지붕은 평평한가요, 아니면 뾰족한가요? 넓고 쾌적한 단층 목조 주택인가요, 아니면 여러 층으로 된 현대적인 주택인가요? 침실은 몇 개나 있나요? 이 모든 것들이 집의 전체적인 구조, 즉 건축 양식을 결정합니다. 이제 집 내부를 상상해 보세요. 카펫이 깔려 있나요, 아니면 원목 마루인가요? 벽 색깔은 무엇인가요? 스탠드 조명이 있나요, 아니면 천장 조명이 있나요? 이 모든 것들이 집의 디자인과 관련이 있습니다.

마찬가지로 소프트웨어 아키텍처는 시스템의 외관보다는 구조에 더 중점을 두는 반면, 디자인은 시스템의 외관에 더 중점을 두고 구조보다는 외관에 더 중점을 둡니다. 예를 들어, 마이크로서비스를 사용하기로 한 결정은 시스템의 구조를 정의합니다.

시스템의 형태(아키텍처)가 시스템 디자인을 정의하는 반면, 사용자 인터페이스(UI) 화면의 모양과 느낌이 시스템 디자인을 정의합니다.

하지만 서비스를 더 작은 부분으로 분할할지 여부나 UI 프레임워크를 결정할 때와 같은 문제는 어떻게 해결해야 할까요? 안타깝게도 이러한 결정들은 대부분 아키텍처와 디자인의 경계선상에 놓여 있어 무엇을 아키텍처로 간주해야 할지 판단하기 어렵습니다.

다음 기준을 활용하면 어떤 것이 건축에 더 가까운지, 아니면 디자인에 더 가까운지 판단하는 데 도움이 될 수 있습니다.

- 전략적인 성격이 더 강한가, 아니면 전술적인 성격이 더 강한
  가? • 변경 또는 구축에 얼마나 많은 노력이 필요한가? • 절충점은 얼마
  나 중요한가?

그림 2-1 은 이러한 요소들을 보여주며 , 건축과 디자인 사이의 스펙트럼을 나타내어 의사 결정의 위치와 책임 소재를 판단하는 데 도움을 줍니다.

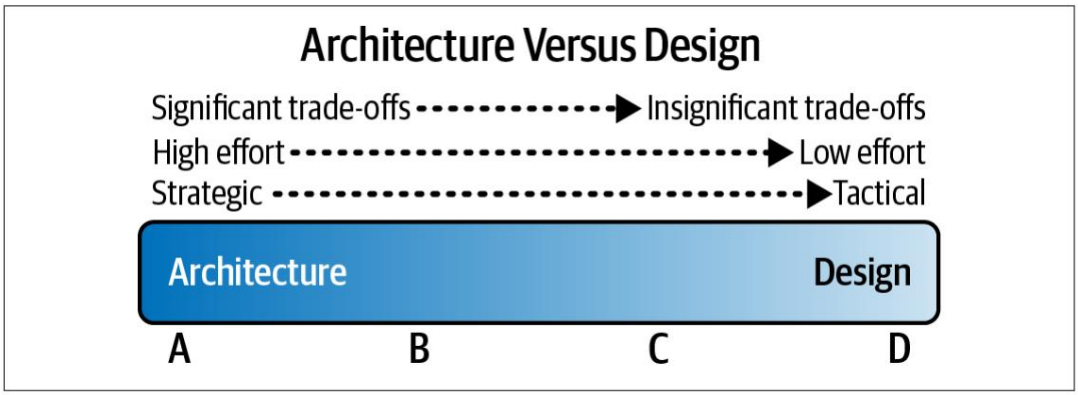

그림 2-1. 건축과 디자인 사이의 스펙트럼

전략적 결정과 전술적 결정

전략적 결정일수록 건축적인 성격이 강해진다. 반대로 전술적 결정일수록 디자인적인 측면이 더 두드러진다. 전략적 결정은 일반적으로 장기적인 반면, 전술적 결정은 일반적으로 단기적이며 다른 행동이나 결정과는 독립적인 경우가 많다.

어떤 결정이 전략적인지 전술적인지 판단하는 좋은 방법 중 하나는 다음과 같은 질문들을 고려하는 것입니다.

이 결정에는 얼마나 많은 생각과 계획이 필요합니까?
: 몇 분 안에 내려지는 결정은 전술적인 측면, 즉 디자인적인 측면이 더 강할 가능성이 높고, 몇 주간의 계획이 필요한 결정은 전략적인 측면, 즉 건축적인 측면이 더 강할 가능성이 높습니다.

그 결정에 몇 명이 참여했습니까?
: 혼자 또는 동료와 함께 내리는 결정은 전술적이고 디자인적인 측면에 더 가깝지만, 다양한 이해관계자들과 여러 차례 회의를 거쳐 내리는 결정은 전략적이고 아키텍처적인 측면에 더 가깝다.

이 결정은 장기적인 비전인가, 아니면 단기적인 행동인가?
: 곧 바뀔 가능성이 높은 결정은 대개 전술적인 성격을 띠며 디자인과 더 관련이 깊은 반면, 오랜 기간 지속될 결정은 일반적으로 더 전략적이고 구조적인 측면이 더 강합니다.

이러한 질문들은 다소 주관적이지만, 어떤 것이 전략적인지 전술적인지, 따라서 건축에 더 가까운지 디자인에 더 가까운지를 판단하는 데 도움이 됩니다.

노력 수준

소프트웨어 아키텍트인 마틴 파울러는 그의 유명한 논문 "누가 아키텍트를 필요로 하는가?" 에서 아키텍처를 "변경하기 어려운 것"이라고 정의했습니다. 일반적으로 변경하기 어려운 것은 더 많은 노력이 필요하므로, 그러한 결정이나 활동은 아키텍처 영역에 더 가깝다고 볼 수 있습니다. 반대로 구현이나 변경에 최소한의 노력만 필요한 것은 디자인 영역에 더 가깝습니다.

예를 들어, 단일 구조의 계층형 아키텍처에서 마이크로서비스로 전환하려면 상당한 노력이 필요하므로 아키텍처에 더 중점을 두어야 합니다. 반면 화면의 필드 배치를 바꾸는 것은 최소한의 노력만 필요하므로 디자인에 더 중점을 두어야 합니다.

트레이드오프의 중요성 특정 결정의 트레

이드오프를 분석하는 것은 해당 결정이 아키텍처에 더 가까운지 아니면 디자인에 더 가까운지를 판단하는 데 큰 도움이 됩니다. 트레이드오프가 클수록 해당 결정은 아키텍처적인 측면이 더 강합니다. 예를 들어, 마이크로서비스 아키텍처 스타일을 선택하면 확장성, 민첩성, 유연성 및 내결함성이 향상됩니다. 그러나 이 아키텍처는 매우 복잡하고 비용이 많이 들며 데이터 일관성이 떨어지고 서비스 간 결합도가 높아 성능이 저하됩니다. 이러한 점들은 상당히 큰 트레이드오프입니다. 따라서 이 결정은 디자인보다는 아키텍처적인 측면에 더 가깝다고 결론 내릴 수 있습니다.

설계 결정에도 장단점이 있습니다. 예를 들어, 클래스 파일을 분리하면 유지 관리성과 가독성이 향상되지만, 관리해야 할 클래스 수가 늘어난다는 단점이 있습니다.

이러한 절충점은 (특히 마이크로서비스의 절충점과 비교했을 때) 그다지 중요하지 않으므로, 이 결정은 설계 측면에 더 가깝습니다.

기술적 폭

개발자는 업무 수행을 위해 상당한 기술적 깊이가 필요한 반면, 소프트웨어 아키텍트는 아키텍처적 관점에서 사물을 바라보기 위해 상당한 기술적 폭이 필요합니다. 기술적 깊이란 특정 프로그래밍 언어, 플랫폼, 프레임워크, 제품 등에 대한 깊이 있는 지식을 의미하는 반면, 기술적 폭은 다양한 분야에 대해 조금씩 아는 것을 의미합니다.

차이점을 더 잘 이해하기 위해 그림 2-2에 나타난 지식 피라미드를 생각해 보세요. 이 피라미드는 세상의 모든 기술 지식을 담고 있으며, 이를 세 가지 수준으로 나눌 수 있습니다. 즉, 아는 것, 모르는 것을 아는 것, 그리고 모르는 것을 모르는 것입니다.

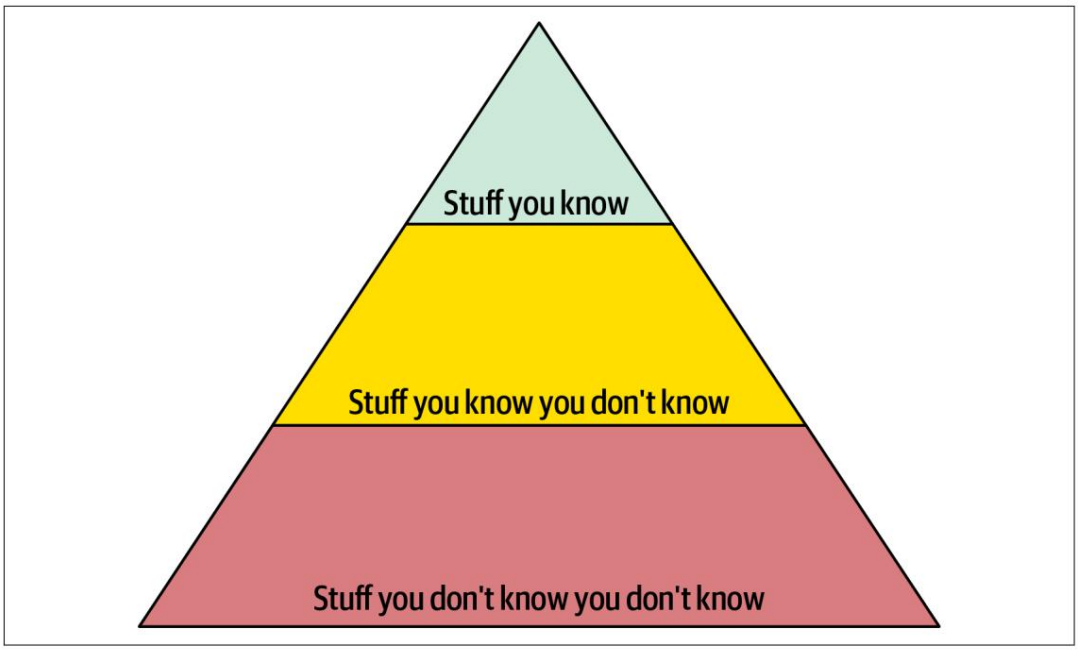

그림 2-2. 모든 지식을 나타내는 피라미드

Stu가 알고 있는 지식에는 기술자들이 업무를 수행하기 위해 매일 사용하는 기술, 프레임워크, 언어 및 도구가 포함됩니다(예: 자바 프로그래머가 자바를 아는 것). 그들은 이러한 모든 것에 능숙하거나 심지어 전문가 수준입니다. 피라미드의 최상단에 해당하는 이 수준의 지식은 가장 작고 포함하는 항목의 수가 가장 적다는 점에 주목하세요. 이는 대부분의 기술자들이 전문성을 개발할 분야를 선택해야 하기 때문입니다. 누구도 모든 것에 전문가가 될 수는 없습니다.

스튜, 당신이 모르는 것(피라미드의 중간 부분)은 기술자가 조금 알거나 들어본 적은 있지만 경험이나 전문 지식이 거의 또는 전혀 없는 것들을 포함합니다. 예를 들어, 대부분의 기술자는 Clojure에 대해 들어봤고 Lisp 기반 프로그래밍 언어라는 것을 알지만 Clojure로 소스 코드를 작성할 수는 없습니다. 이 수준의 지식은 최상위 수준보다 훨씬 더 광범위합니다. 이는 사람들이 전문성을 쌓을 수 있는 것보다 훨씬 더 많은 것에 익숙해질 수 있기 때문입니다.

'모른다는 사실조차 모르는 지식'은 지식 피라미드의 가장 큰 부분을 차지합니다. 여기에는 문제를 해결하려는 기술자가 존재조차 모르는 수많은 기술, 도구, 프레임워크, 언어가 포함됩니다. 개인의 커리어 목표는 '모른다는 사실조차 모르는 지식'에서 피라미드의 두 번째 영역인 '모른다는 사실조차 아는 지식'으로 옮기는 것이며, 전문성이 요구될 때는 피라미드의 중간 부분에서 꼭대기 부분인 '아는 지식'으로 옮겨가는 것입니다.

개발자 경력 초기에 피라미드 꼭대기 (그림 2-3) 를 확장한다는 것은 귀중한 전문 지식을 습득하는 것을 의미합니다. 하지만 알고 있는 지식은 끊임없이 변화하기 때문에 유지 관리해야 할 부분이기도 합니다. 예를 들어, 개발자가 Ruby on Rails 전문가가 되었다 하더라도 1~2년 동안 Ruby on Rails를 사용하지 않으면 그 전문 지식은 오래가지 못합니다. 피라미드 꼭대기의 지식을 유지하려면 지속적인 시간 투자가 필요합니다. 이 꼭대기 부분은 개인의 기술적 깊이, 즉 그들이 정말 잘 알고 있는 분야를 나타냅니다.

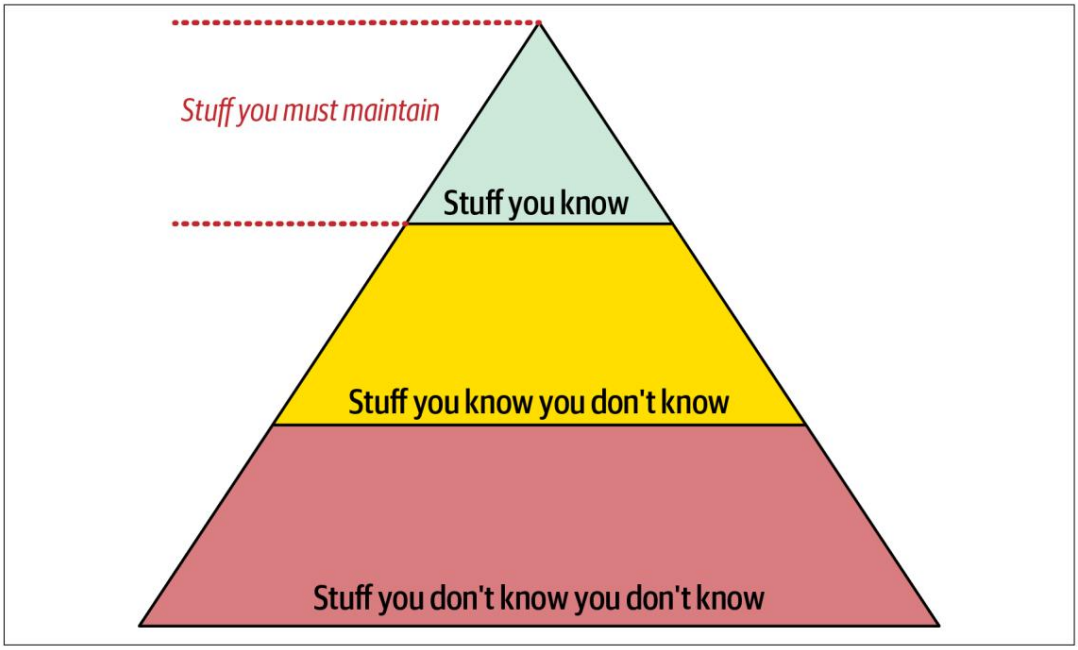

그림 2-3. 개발자는 전문성을 유지하기 위해 지속적으로 전문성을 점검해야 합니다.

하지만 개발자가 아키텍트 역할로 전환하면서 지식의 본질은 변화합니다. 아키텍트의 가치는 기술에 대한 폭넓은 이해와 이를 활용하여 특정 문제를 해결하는 방법에 대한 지식에서 크게 드러납니다. 예를 들어, 특정 문제에 대해 다섯 가지 해결책이 존재한다는 것을 아는 것이 한 가지 기술에만 전문성을 갖는 것보다 훨씬 효과적입니다. 아키텍트에게 가장 중요한 피라미드 부분은 최상단과 중간 부분이며, 그림 2-4에서 볼 수 있듯이 중간 부분이 최하단 부분까지 얼마나 깊이 스며들어 있는지가 아키텍트의 기술적 폭을 나타냅니다.

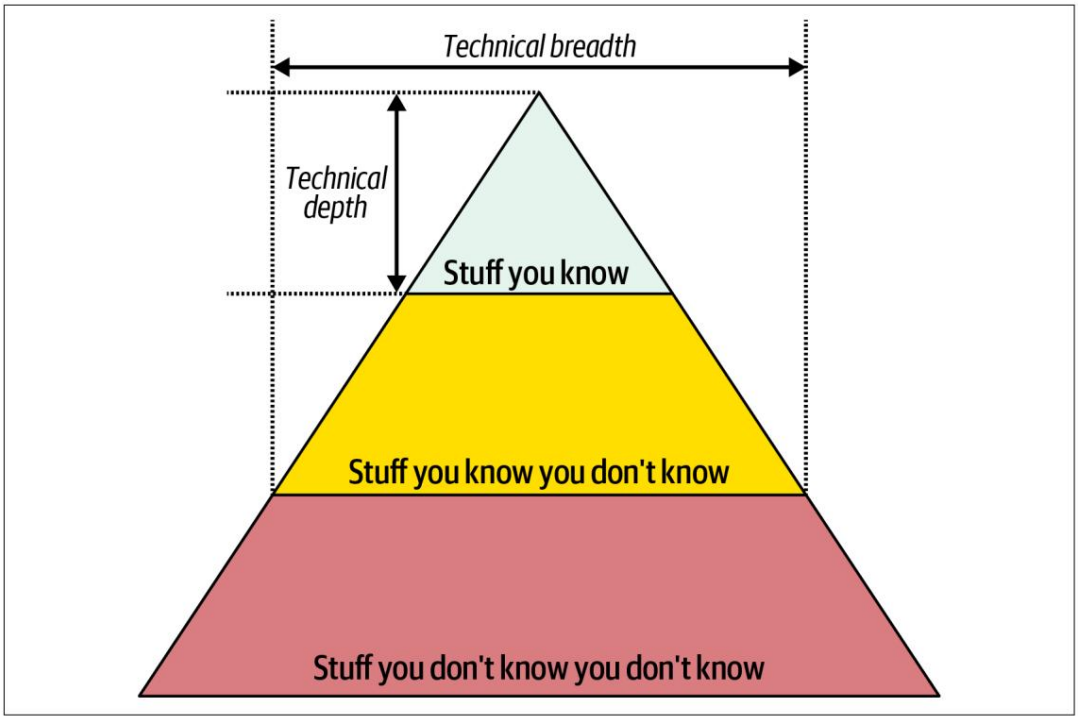

그림 2-4. 어떤 주제에 대해 얼마나 많이 알고 있느냐를 기술적 깊이라고 하고, 얼마나 많은 주제에 대해 알고 있느냐를 기술적 폭이라고 한다.

건축가에게는 깊이보다 폭이 더 중요합니다. 건축가는 기술적 제약 조건에 맞춰 역량을 조정하는 결정을 내려야 하므로, 다양한 솔루션에 대한 폭넓은 이해가 필수적입니다. 따라서 건축가에게 현명한 선택은 어렵게 쌓아온 전문 지식의 일부를 포기하고 그 시간을 활용하여 그림 2-5에서처럼 포트폴리오를 확장하는 것입니다. 특히 흥미로운 기술 분야처럼 특정 전문 분야는 유지되겠지만, 다른 분야는 유용하게 퇴보할 것입니다.

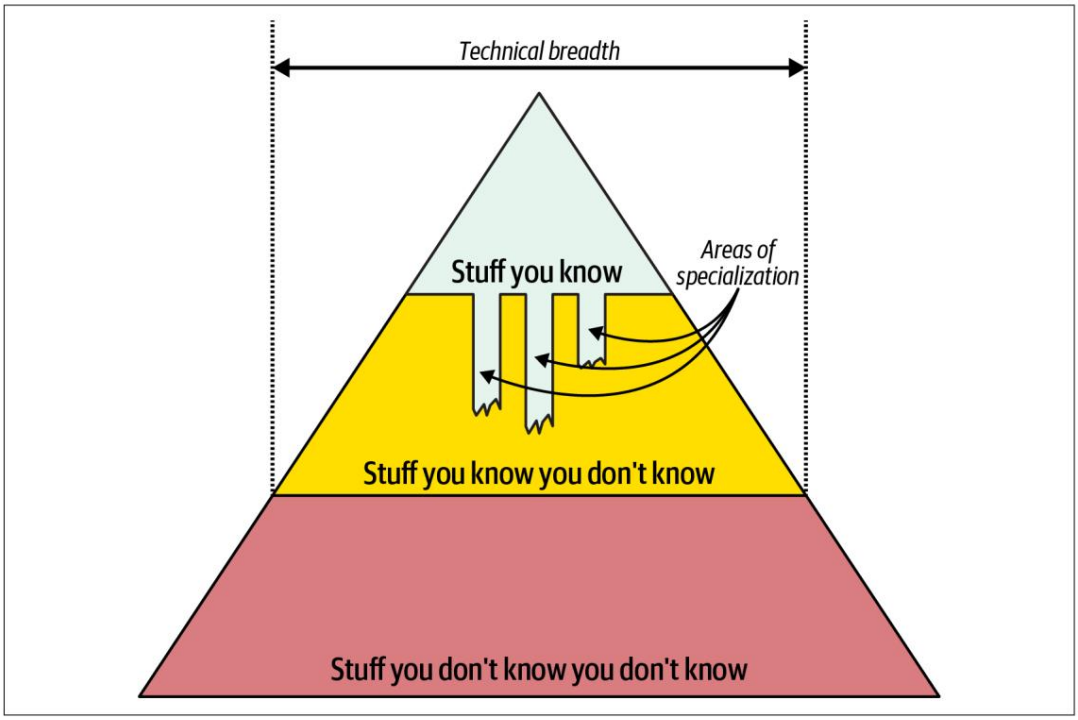

그림 2-5. 건축가 역할의 범위 확대 및 깊이 축소

우리의 지식 피라미드는 아키텍트와 개발자의 역할이 얼마나 근본적으로 다른지를 보여줍니다. 개발자는 평생 전문성을 갈고닦습니다. 아키텍트 역할로 전환한다는 것은 관점의 변화를 의미하며, 많은 사람들이 이를 어려워합니다. 이는 두 가지 일반적인 문제로 이어집니다. 첫째, 아키텍트는 매우 다양한 분야의 전문성을 유지하려 애쓰지만, 결국 어느 하나에도 성공하지 못하고 과로하게 됩니다. 둘째, 시대에 뒤떨어진 전문 지식, 즉 자신이 가진 구식 정보가 여전히 최첨단이라고 착각하는 현상이 나타납니다. 이는 특히 대기업에서 흔히 볼 수 있는데, 회사를 설립한 개발자들이 리더십 역할을 맡았음에도 불구하고 여전히 구시대적인 기준으로 기술 결정을 내리는 경우입니다(24 페이지의 "냉혹한 원시인 안티패턴" 참조).

프로즌 케이브맨 안티패턴(Frozen

Caveman Antipattern)은 프로그래머 앤드류 코닉이 정의한 것으로, 처음에는 좋은 아이디어처럼 보이지만 결국 문제를 야기하는 것을 의미합니다. 흔히 볼 수 있는 행동적 안티패턴인 프로즌 케이브맨 안티패턴은 설계자가 모든 아키텍처 설계에서 자신이 가진 비합리적인 걱정에 집착하는 현상을 설명합니다. 예를 들어, 닐의 동료 중 한 명은 중앙 집중식 아키텍처를 특징으로 하는 시스템을 개발했습니다. 그들이 클라이언트 측 설계자에게 설계를 전달할 때마다 끊임없이 제기되는 질문은 "하지만 이탈리아와의 연결이 끊기면 어떡하죠?"였습니다. 몇 년 전, 예상치 못한 통신 문제로 클라이언트 본사와 이탈리아 지점 간의 연락이 두절되어 큰 불편을 초래했던 경험이 있었기 때문입니다. 이러한 상황이 다시 발생할 가능성은 극히 낮지만, 설계자들은 이 특정 아키텍처적 특징에 집착하게 된 것입니다.

일반적으로 이러한 안티패턴은 과거에 잘못된 결정이나 예상치 못한 사건으로 인해 쓴맛을 본 경험이 있는 설계자들이 관련된 모든 것에 대해 지나치게 조심스러워하는 경향에서 나타납니다. 위험 평가는 중요하지만 현실적이어야 합니다. 진정한 기술적 위험과 인식된 위험의 차이를 이해하는 것은 지속적인 학습 과정의 일부입니다. 설계자처럼 생각한다는 것은 이러한 고정관념과 경험을 극복하고, 다른 해결책을 모색하며, 더욱 관련성 있는 질문을 던지는 것을 의미합니다.

아키텍트는 더 넓은 기술 포트폴리오를 구축하여 다양한 분야에서 활용할 수 있도록 기술적 폭을 넓히는 데 집중해야 합니다. 개발자에서 아키텍트로 전향하는 사람들은 지식 습득 방식에 대한 관점을 바꿔야 할 수도 있습니다. 개발자라면 누구나 경력 전반에 걸쳐 지식 포트폴리오의 깊이와 폭을 균형 있게 유지하는 것을 고려해야 합니다. 그렇다면 아키텍트는 어떻게 기술적 폭을 넓힐 수 있을까요? 다음 섹션에서는 "자신이 모른다는 사실조차 모르는 것"을 발견하는 데 도움이 되는 몇 가지 기법을 소개합니다.

20분 규칙

그림 2-5 에서 볼 수 있듯이 , 건축가에게는 기술적 깊이보다 기술적 폭이 더 중요합니다. 하지만 풀타임으로 일하면서 경력을 쌓고, 친구들과 시간을 보내고, 가족을 돌보면서 어떻게 최신 트렌드와 유행어들을 모두 따라잡을 수 있을까요?

저희가 사용하는 기법 중 하나는 '20분 규칙'입니다. 하루에 최소 20분을 새로운 것을 배우거나 특정 주제를 더 깊이 탐구하는 데 투자하는 것입니다. 그림 2-6 은 InfoQ, DZone Refcardz, Thoughtworks Technology Radar 와 같이 20분을 활용하기 좋은 곳들을 보여줍니다 . 생소한 용어는 인터넷 검색을 통해 더 자세히 알아볼 수 있으며, 이를 통해 '모른다는 사실조차 몰랐던 것'을 '모른다는 것을 아는 것'으로 바꿀 수 있습니다. 이 책처럼 관련 서적을 읽는 데 시간을 투자하는 것도 좋은 방법입니다. 중요한 것은 시간을 조금씩이라도 내어 새로운 것을 배우는 것입니다.

바쁜 일상 속에서 시간을 내어 기술적 역량을 넓히고 궁극적으로 경력을 발전시키는 데 집중하세요.

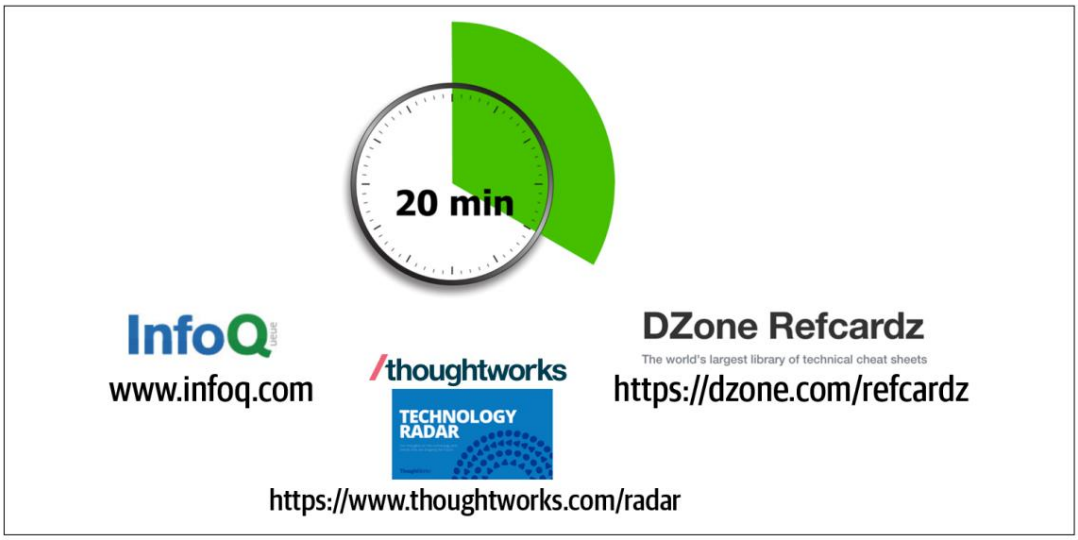

그림 2-6. 20분 규칙

많은 기술자들이 처음 이 개념을 받아들일 때 점심시간이나 퇴근 후에 20분을 확보하려고 계획하지만, 경험상 이러한 시간대는 제대로 활용하기 어렵습니다. 점심시간을 휴식보다는 밀린 업무를 처리하는 데 사용하기 쉽고, 저녁 시간은 하루 일과를 마친 후 친구들과의 약속, 가족과의 시간 등으로 더욱 어려워집니다.

대신, 아침에 일어나자마자 커피나 차 한 잔을 마시고, 무엇보다 이메일을 확인하기 전에 20분을 투자하는 것을 강력히 추천합니다. 이메일을 확인하는 순간 아침은 끝나고 하루가 시작되기 때문입니다. 정신이 맑고 집중력이 흐트러지기 전에, 그리고 온갖 방해 요소에 휘둘리지 않고 20분을 활용하세요.

20분 규칙을 따르면 기술적 폭이 넓어지고 효과적인 소프트웨어 아키텍트가 되기 위한 지식을 개발하고 유지하는 데 도움이 될 것입니다.

개인 레이더 개발하기

1990년대 대부분과 2000년대 초반까지, 필자 중 한 명은 소규모 교육 및 컨설팅 회사의 CTO였습니다. 그가 그 회사에 처음 왔을 때, 주력 플랫폼은 dBASE 파일 위에 DOS 애플리케이션을 구축하는 신속 애플리케이션 개발 도구인 Clipper였습니다. 그러던 어느 날, Clipper가 사라져 버렸습니다. 회사는 윈도우의 등장을 인지하고 있었지만, 비즈니스 시장은 여전히 DOS 시대였습니다… 그러다 갑자기 모든 것이 바뀌어 버린 것입니다.

한 동료는 자신들이 축적해 온 방대한 양의 클리퍼 관련 지식이 이제는 쓸모없어졌다는 사실에 한탄하며, 그 지식을 다른 것으로 대체할 수 없다는 점을 토로했습니다. 그는 역사상 어떤 집단이 이처럼 상세한 지식을 습득하고 버린 적이 있었을까 하는 의문을 제기했습니다.

소프트웨어 개발자로서 그들의 삶은 어떨까요? 그 경험은 그들에게 깊은 인상을 남겼습니다. 기술의 발전을 무시하는 것은 위험한 일이라는 것을요.

또한 이는 기술 거품에 대한 중요한 교훈을 주었습니다. 개발자, 아키텍트 및 기타 기술 전문가들이 특정 기술에 깊이 몰두하여 모든 노력과 생각을 쏟아붓다 보면, 마치 밈처럼 특정 기술에 대한 정보만 퍼져나가는 거품 속에 갇히게 됩니다. 이러한 거품은 일종의 확증 편향의 장으로 작용하여, 그 안에서는 모든 사람이 우리만큼이나 해당 기술에 대해 잘 알고 관심을 갖게 됩니다. 특히 그 거품이 특정 기술 공급업체에 의해 만들어진 경우라면, 외부의 객관적인 평가를 접하기조차 어려울 수 있습니다. 그리고 거품이 무너지기 시작할 때는 이미 너무 늦을 때까지 아무런 경고도 없습니다.

우리가 사는 세상에는 기술 레이더가 부족합니다. 기존 기술과 신기술의 위험과 이점을 평가하는 데 도움이 되는 살아있는 문서가 필요한 것이죠. 이 레이더 개념은 닐이 이사 겸 소프트웨어 아키텍트로 재직 중인 Thoughtworks에서 나왔습니다 . 이 섹션에서는 이 개념이 어떻게 탄생했는지 설명하고, 자신만의 레이더를 만드는 방법을 알려드리겠습니다.

Thoughtworks Technology Radar는

Thoughtworks의 기술 자문 위원회(TAB)에서 발행하는 보고서입니다. TAB는 CTO가 회사와 고객을 위한 기술 방향 및 전략에 대한 결정을 내리는 데 도움을 주는 고위 기술 리더들로 구성된 그룹입니다. 최신 기술 동향을 파악하기 위해 이 그룹은 현재 연 2회 발행되는 Technology Radar를 제작하기 시작했습니다.

이것은 예상치 못한 부작용을 가져왔습니다. 닐이 컨퍼런스에서 강연을 할 때, 참석자들은 그를 찾아와 레이더 개발에 도움을 준 것에 대해 감사를 표했고, 종종 자신들의 회사에서도 자체 버전을 개발하기 시작했다고 덧붙였습니다. 닐은 또한 이것이 컨퍼런스 발표 패널에서 끊임없이 제기되는 질문, 즉 "어떻게 기술 발전에 발맞춰 나가십니까? 다음에 무엇을 추구해야 할지 어떻게 파악하십니까?"에 대한 해답이라는 것을 깨달았습니다. 물론 그 해답은 발표자 모두에게 일종의 내면의 레이더가 있다는 것입니다.

구성 요소. Thoughtworks Radar는 소프트웨어 개발 환경의 대부분을 포괄하도록 설계된 네 가지 사분면으로 구성됩니다.

도구

: IDE와 같은 개발 도구부터 기업용 통합 도구까지 모든 것이 포함됩니다.

언어 및 프레임워크

: 일반적으로 오픈 소스인 컴퓨터 언어, 라이브러리 및 프레임워크

기술: 프로세

: 스, 엔지니어링 관행 및 조언을 포함하여 소프트웨어 개발 전반을 지원하는 모든 관행

플랫폼

데이터베이스, 클라우드 공급업체 및 운영 체제를 포함한 기술 플랫폼

링. 레이더에는 바깥쪽에서 안쪽 순으로 다음과 같은 4개의 링이 있습니다.

홀드

(Hold) 링의 원래 의미는 "지금은 보류"였습니다. 이는 아직 충분히 평가하기에는 너무 새로운 기술, 즉 많은 관심을 받고 있지만 아직 검증되지 않은 기술을 나타내는 것이었습니다. 홀드 링은 시간이 지나면서 의미가 바뀌었고, 이제는 "이 기술로 새로운 시도를 하지 마세요"와 같은 의미를 지닙니다.

기존 프로젝트에 사용하는 것은 괜찮지만, 새로운 개발에 사용할 때는 신중하게 생각해야 합니다.

평가 단계

는 해당 기술이 조직에 미치는 영향을 파악하기 위해 (개발 스파이크, 연구 프로젝트 또는 컨퍼런스 세션 등을 통해) 탐색해 볼 가치가 있음을 나타냅니다. 예를 들어, 모바일 브라우저가 주목받기 시작했을 때 많은 대기업들이 모바일 전략을 수립하는 과정에서 이 단계를 거쳤습니다.

시험 단

계는 개발 가치가 있는 기술을 위한 단계입니다. 이 단계에 있는 기능은 구축 방법을 이해하는 것이 중요합니다. 지금이 바로 위험 부담이 적은 프로젝트를 시범 운영해 볼 적기입니다.

Adopt

Thoughtworks는 업계가 Adopt 목록에 있는 항목들을 채택해야 한다고 강력하게 주장합니다.

그림 2-7 의 레이더 예시에서 각 "점"은 서로 다른 기술 또는 기법을 나타냅니다. Thoughtworks는 레이더를 통해 소프트웨어 업계에 대한 집단적인 의견을 전달하지만, 많은 개발자와 아키텍트 또한 이를 기술 평가 프로세스를 구조화하고 시간 투자 방향을 정하는 데 활용합니다. 개인적인 용도로는 사분면의 의미를 다음과 같이 변경하는 것을 권장합니다.

여기에

는 피해야 할 기술과 기법뿐만 아니라 고치려고 노력하는 습관도 포함될 수 있습니다. 예를 들어, .NET 분야의 아키텍트라면 팀 내부 사정에 대한 최신 뉴스나 가십을 포럼에서 읽는 데 익숙할 수 있습니다. 재미있을 수는 있지만, 이러한 정보는 가치가 낮을 수 있습니다.

홀드 링에 넣어두면 피하고 싶은 것을 상기시키는 역할을 합니다.

평가하기

링을 사용하여 아직 직접 평가해 볼 시간이 없었지만 좋은 평을 들었던 유망한 기술을 평가해 보세요. 이 링은 향후 더 심층적인 연구를 위한 준비 단계 역할을 합니다.

시험 단

계(Trial) 링은 대규모 코드베이스 내에서 스파이크 실험을 수행하는 것과 같은 활발한 연구 개발을 나타냅니다. 이러한 기술은 시간을 투자하여 더 깊이 이해하고, 이를 통해 트레이드오프 분석에 효과적으로 포함할 가치가 있습니다.

나만의

'Adopt' 반지는 당신이 가장 기대하는 새로운 것들과 특정 문제를 해결하기 위한 최고의 방법들을 상징합니다.

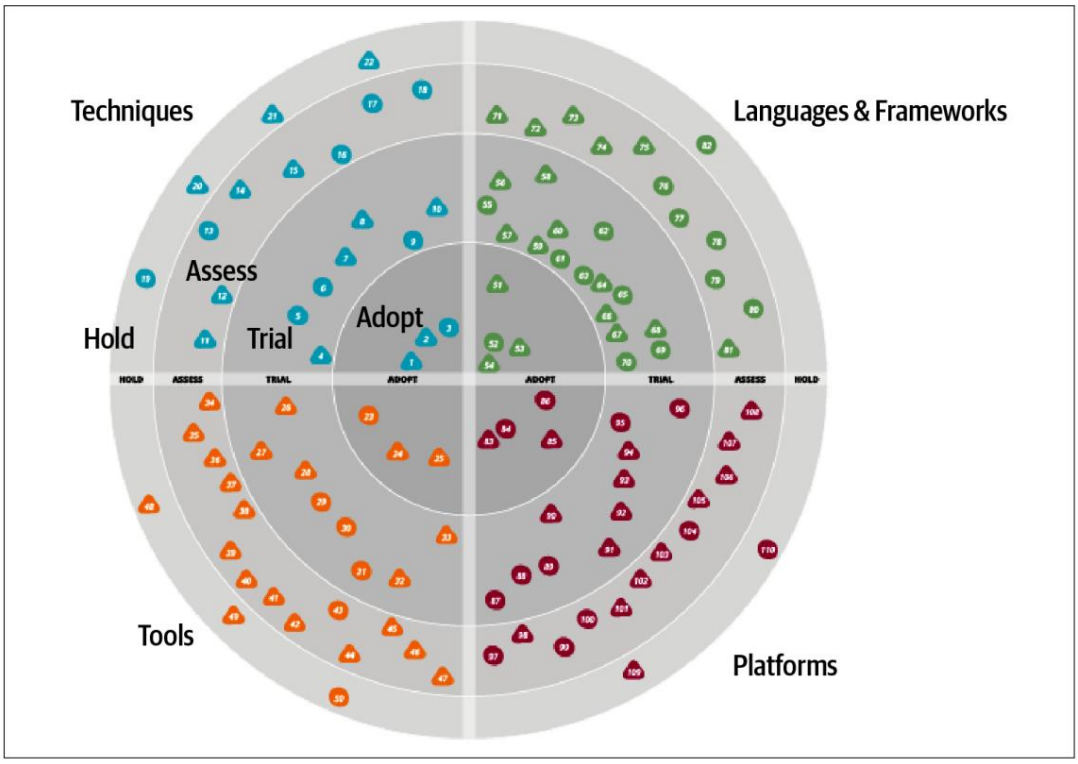

그림 2-7. Outworks 기술 레이더 샘플

대부분의 기술 전문가들은 유행하는 기술이나 고용주가 사용하는 기술을 기준으로 즉흥적으로 기술을 선택하는 경향이 있지만, 기술 포트폴리오 관리에 있어 방임적인 태도를 취하는 것은 경력에 위험할 수 있습니다. 기술 레이더를 구축하면 기술에 대한 사고방식을 체계화하고 상반되는 의사결정 기준 사이의 균형을 맞추는 데 도움이 됩니다. (예를 들어, "더 유행하는" 기술 분야의 새로운 직장을 구하기는 어려울 수 있지만, 이미 확립된 기술은 일자리 시장은 크지만 업무의 흥미도는 떨어질 수 있습니다.)

기술 포트폴리오를 금융 포트폴리오처럼 관리하세요. 분산 투자가 중요합니다! 수요가 높은 기술이나 스킬을 선택하고 그 수요를 꾸준히 추적하세요. 하지만 생성형 AI나 임베디드 IoT 기기처럼 과감한 기술 혁신에도 도전해 볼 만합니다. 밤늦게까지 오픈 소스 프로젝트에 매달리다가 인기를 얻고 결국 유료화된 프로젝트를 통해 사무실 생활에서 벗어난 개발자들의 이야기는 흔합니다. 이처럼 특정 분야에 깊이 파고들기보다는 폭넓은 지식을 쌓는 것이 중요합니다.

개인 기술 감지 능력을 키우는 것은 기술 포트폴리오를 확장하는 데 좋은 기반을 제공하지만, 궁극적으로는 결과보다 과정 자체가 더 중요합니다.
레이더 시각화를 만들면 바쁜 일정 속에서도 시간을 내어 이러한 것들을 생각해 볼 수 있는 계기가 되는데, 이러한 종류의 사고를 실현하는 데에는 종종 그 방법밖에 없습니다.

개인 레이더를 구축하는 데 있어 가장 중요한 부분은 그 과정에서 생성되는 대화이지만, 매우 유용한 시각화 자료를 만들어내는 것 또한 중요한 역할을 합니다. 기술 전문가들이 자신만의 레이더 시각화를 구축해 달라는 요청이 많았던 만큼, Thoughtworks는 ' 나만의 레이더 만들기(Build Your Own Radar)'라는 도구를 출시했습니다. Google 스프레드시트를 입력하면 개인 레이더를 시각화하여 보여줍니다. 모든 기술 전문가 여러분께서 이 도구를 적극적으로 활용해 보시길 권장합니다.

Trade-O! 분석

건축가처럼 생각한다는 것은 기술적이든 아니든 모든 해결책에서 장단점을 파악하고, 그 장단점을 분석하여 최적의 해결책을 찾는 것입니다. 이것이 건축가의 핵심 활동 중 하나(따라서 건축적 사고의 일부)인 이유는 (저자 중 한 명인) 마크의 다음 인용문에서 잘 드러납니다.

> 건축이란 구글 검색이나 법학 석사 과정 학생에게 물어볼 수 있는 것이 아니다.
> —마크 리처즈

아키텍처의 모든 것은 절충의 연속입니다. 그래서 세상의 모든 아키텍처 질문에 대한 유명한 대답은 "상황에 따라 다릅니다"입니다. 이 대답이 다소 거슬릴지 모르지만, 안타깝게도 사실입니다. REST API가 시스템에 더 적합한지, 메시징이 더 적합한지, 마이크로서비스 아키텍처가 신제품에 적합한지 구글 검색이나 인공지능 엔진, 대규모 언어 모델(LLM)에 물어볼 수는 없습니다. 왜냐하면 답은 정말로 상황에 따라 다르기 때문입니다. 배포 환경, 비즈니스 목표, 기업 문화, 예산, 시간 제약, 개발자 역량 등 수많은 요소에 따라 달라집니다. 모든 사람의 환경, 상황, 그리고 해결해야 할 문제는 다를 것입니다.

그래서 건축이 어려운 겁니다. 다른 저자인 닐의 말을 인용하자면:

> 건축에는 정답이나 오답이 없고, 단지 절충안만 있을 뿐입니다.
> —닐 포드

예를 들어, 온라인 입찰자들이 경매에 나온 품목에 입찰하는 품목 경매 시스템 (그림 2-8) 을 생각해 보겠습니다. 입찰 생성기 서비스는 입찰자로부터 입찰 금액을 생성한 다음, 해당 입찰 금액을 입찰 캡처, 입찰 추적 및 입찰 분석 서비스로 전송합니다.

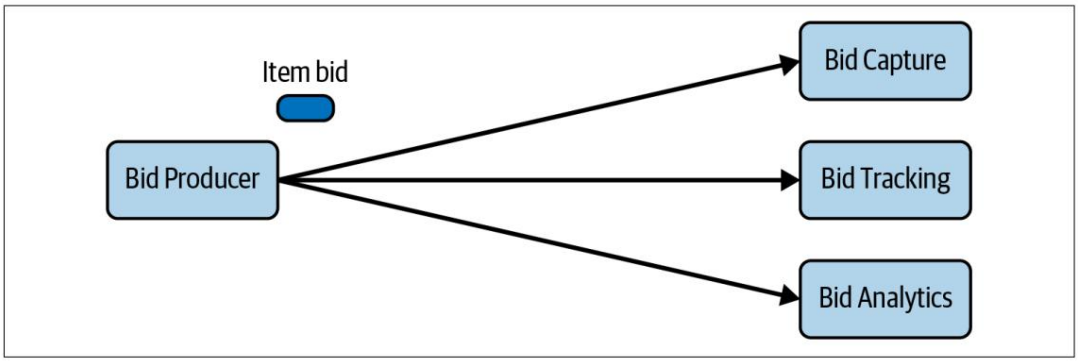

그림 2-8. 경매 시스템의 예시 - 거래 대기열 또는 주제?

이 시스템의 비동기 동작을 구현하기 위해 아키텍트는 지점 간 메시징 방식으로 큐를 사용하거나, 발행-구독 메시징 방식으로 토픽을 사용할 수 있습니다. 어떤 방식을 선택해야 할까요? 구글 검색으로는 답을 찾을 수 없습니다. 아키텍트의 사고방식은 각 옵션의 장단점을 분석하고 특정 상황에 맞는 최적의 (또는 차선책) 옵션을 선택하도록 요구합니다.

품목 경매 시스템의 두 가지 메시징 옵션은 그림 2-9에 나와 있는데, 이는 발행-구독 메시징 모델에서 토픽을 사용하는 방법을 보여주고, 그림 2-10은 지점 간 메시징 모델에서 큐를 사용하는 방법을 보여줍니다.

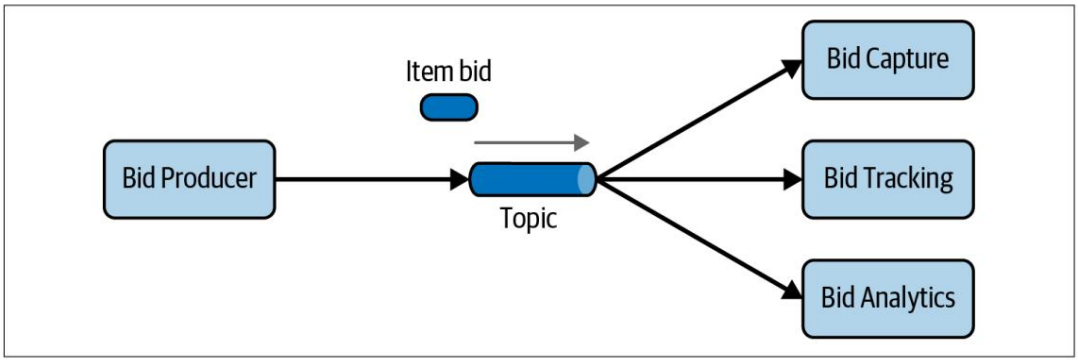

그림 2-9. 서비스 간 통신을 위한 토픽 사용

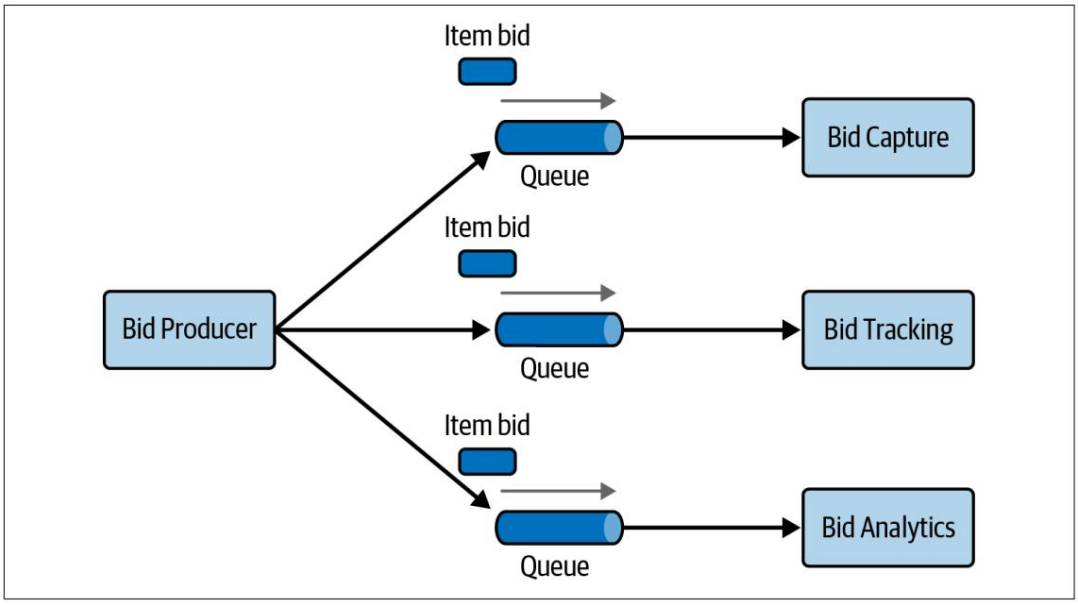

그림 2-10. 서비스 간 통신을 위한 큐 사용

그림 2-9 에서 이 문제에 대한 명확한 장점(그리고 명백해 보이는 해결책) 은 아키텍처 확장성입니다. 입찰 생성자 서비스는 토픽에 대한 단 하나의 연결만 필요합니다. 그림 2-10의 큐 방식과 비교해 보면, 입찰 생성자는 세 개의 서로 다른 큐에 연결해야 합니다. 만약 입찰 내역 이라는 새로운 서비스 (각 입찰자에게 모든 입찰 내역을 제공하는 서비스)를 이 시스템에 추가하더라도 기존 서비스나 인프라를 변경할 필요가 없습니다. 새로운 입찰 내역 서비스는 이미 입찰 정보가 포함된 토픽을 구독하기만 하면 됩니다.

하지만 그림 2-10 에 나타난 큐 방식을 사용할 경우 , 입찰 내역 서비스에는 새로운 큐가 필요하며, 입찰 생성자 (Bid Producer) 도 새 큐에 대한 연결을 추가하도록 수정해야 합니다. 여기서 중요한 점은 큐 방식을 사용하면 새로운 입찰 기능을 추가할 때 서비스와 인프라에 상당한 변경이 필요하다는 것입니다. 반면 토픽 방식을 사용하면 기존 인프라를 변경할 필요가 없습니다. 또한 토픽 방식에서는 입찰 생성자 가 입찰 정보가 어떻게 사용될지, 어떤 서비스에서 사용될지 알 필요가 없으므로 시스템과의 연결성이 낮습니다 . 반면 큐 방식에서는 입찰 생성자가 입찰 정보가 어떻게 사용될지(그리고 누가 사용할지) 정확히 알고 있으므로 시스템과의 연결성이 더 높습니다.

지금까지의 비교 분석 결과, 발행-구독 메시징 모델을 사용하는 토픽 접근 방식이 명백하고 최선의 선택인 것처럼 보입니다. 하지만 클로저 프로그래밍 언어의 창시자인 리치 히키의 말을 인용하자면,

> 프로그래머는 모든 것의 장점은 알지만 단점은 전혀 모릅니다. 반면 아키텍트는 장점과 단점 모두를 이해해야 합니다.
>
> —리치 히키

아키텍처적 사고란 주어진 솔루션의 장점뿐만 아니라 그에 따른 단점이나 절충점까지 분석하는 것을 의미합니다. 경매 시스템 예시를 다시 살펴보면, 소프트웨어 아키텍트는 토픽 솔루션의 장점뿐 아니라 단점도 분석해야 합니다. 그림 2-9 에서 볼 수 있듯이, 토픽을 사용하면 누구나 입찰 데이터에 접근할 수 있으므로 데이터 접근 및 데이터 보안에 문제가 발생할 수 있습니다.

하지만 그림 2-10 에 나타난 큐 모델에서는 큐로 전송된 데이터는 해당 메시지를 수신하는 특정 소비자만 접근할 수 있습니다. 만약 악의적인 서비스가 큐를 도청하더라도, 해당 서비스는 요청을 수신하지 못하게 되고 데이터 손실(및 잠재적인 보안 침해)에 대한 알림이 즉시 전송됩니다. 다시 말해, 토픽을 도청하는 것은 매우 쉽지만, 큐를 도청하는 것은 그렇지 않습니다.

보안 문제 외에도 그림 2-9 의 토픽 솔루션은 동질적인 계약만 지원합니다. 입찰 데이터를 수신하는 모든 서비스는 동일한 데이터 계약과 입찰 데이터 세트를 수락해야 합니다. 큐 방식에서는 각 소비자가 필요한 데이터에 특화된 자체 계약을 가질 수 있습니다. 예를 들어, 새로운 입찰 내역 서비스가 입찰가와 함께 현재 호가를 필요로 하지만 다른 서비스는 해당 정보를 필요로 하지 않는다고 가정해 보겠습니다. 이 경우 계약을 수정해야 하므로 해당 데이터를 사용하는 다른 모든 서비스에 영향을 미칩니다. 큐 모델에서는 별도의 채널이 존재하므로 다른 서비스에 영향을 미치지 않는 별도의 계약이 됩니다.

토픽 모델의 또 다른 단점은 토픽 내 메시지 수를 모니터링하는 기능을 지원하지 않아 자동 확장 기능을 사용할 수 없다는 점입니다.
하지만 큐 옵션을 사용하면 각 큐를 개별적으로 모니터링할 수 있고, 각 입찰 소비자에게 프로그래밍 방식의 로드 밸런싱을 적용하여 각각을 자동으로 독립적으로 확장할 수 있습니다. 단, 이러한 장단점은 기술에 따라 다를 수 있습니다.

AMQP(고급 메시지 큐 프로토콜)는 프로듀서가 메시지를 보내는 익스체인지와 컨슈머가 메시지를 수신하는 큐를 분리함으로써 프로그래밍 방식의 로드 밸런싱 및 모니터링을 지원할 수 있습니다.

이러한 포괄적인 장단점 분석을 고려했을 때, 어느 쪽이 더 나은 선택일까요?

상황에 따라 다릅니다! 표 2-1은 이러한 장단점을 요약한 것입니다.

표 2-1. 주제별 거래

주제의 장점	주제의 단점
아키텍처 확장성, 데이터	접근성 및 데이터 보안 문제
서비스 분리	이질적인 계약 없음
	모니터링 및 프로그램적 확장성

소프트웨어 아키텍처의 모든 것에는 장단점이 존재합니다. 아키텍트처럼 생각한다는 것은 이러한 장단점을 분석하고 "확장성과 보안 중 어느 것이 더 중요한가?"와 같은 질문을 던지는 것을 의미합니다. 아키텍트의 선택은 항상 비즈니스 목표, 환경, 그리고 여러 다른 요소에 따라 달라집니다.

비즈니스 동인 이해하기

아키텍트처럼 생각한다는 것은 시스템의 성공에 필요한 비즈니스 동인을 이해하고, 이러한 요구 사항을 확장성, 성능, 가용성과 같은 아키텍처 특성으로 변환하는 것을 의미합니다. 이는 아키텍트가 비즈니스 영역에 대한 지식을 갖추고 주요 비즈니스 이해관계자들과 긴밀한 협력 관계를 유지해야 하는 어려운 과제입니다. 이 책에서는 이 주제에 네 개의 장을 할애했습니다. 4 장 에서는 다양한 아키텍처 특성을 정의하고, 5 장 에서는 아키텍처 특성을 식별하고 검증하는 방법을 설명합니다. 6 장 에서는 시스템의 비즈니스 요구 사항이 충족되는지 확인하기 위해 각 특성을 측정하는 방법을 설명하고, 마지막 7장 에서는 아키텍처 특성의 범위와 결합도와의 관계를 논의합니다.

아키텍처 설계와 실무 코딩의 균형 유지

아키텍트가 직면하는 어려운 과제 중 하나는 직접적인 코딩과 소프트웨어 아키텍처 설계 사이의 균형을 맞추는 것입니다. 우리는 모든 아키텍트가 코딩을 할 수 있어야 하며, 일정 수준의 기술적 깊이를 유지해야 한다고 굳게 믿습니다(20 페이지의 "기술적 폭" 참조).
이는 쉬운 일처럼 보일 수 있지만, 실제로 달성하기는 상당히 어려울 수 있습니다.

직접 코딩을 하면서 소프트웨어 아키텍트 역할도 병행하려는 사람들을 위한 첫 번째 조언은 '병목 현상 함정'을 피하는 것입니다. 병목 현상 함정은 아키텍트가 시스템의 핵심 경로(주로 기본 프레임워크 코드 또는 복잡한 부분)에 있는 코드에 대한 책임을 맡게 되면서 팀의 병목 현상을 일으키는 안티패턴입니다. 이는 아키텍트가 전업 개발자가 아니기 때문에 개발 업무(소스 코드 작성 및 테스트 등)와 아키텍트 역할(다이어그램 작성, 회의 참석 등) 사이에서 균형을 맞춰야 하기 때문에 발생합니다.

병목 현상 함정을 피하는 한 가지 방법은 아키텍트가 시스템의 핵심 부분을 개발팀의 다른 구성원에게 위임하고, 이후 1~3번의 반복 개발 주기 동안 서비스나 UI 화면과 같은 사소한 비즈니스 기능 구현에 집중하는 것입니다. 이러한 접근 방식에는 세 가지 긍정적인 효과가 있습니다. 첫째, 아키텍트는 팀의 병목 현상을 일으키지 않으면서 실제 운영 환경에서 사용할 코드를 작성하며 실무 경험을 쌓을 수 있습니다. 둘째, 핵심 경로 및 프레임워크 코드가 개발팀에 배분되어(원래 담당해야 할 부분) 팀이 시스템의 어려운 부분을 더 잘 이해하고 책임감을 갖게 됩니다. 셋째, 그리고 아마도 가장 중요한 점은 아키텍트가 개발팀과 동일한 비즈니스 관련 소스 코드를 작성한다는 것입니다. 이를 통해 아키텍트는 개발팀이 프로세스, 절차 및 개발 환경과 관련하여 겪는 어려움을 더 잘 파악하고 개선하는 데 기여할 수 있습니다.

하지만 만약 건축가가 개발팀과 함께 코드를 개발할 수 없다고 가정해 봅시다.
어떻게 하면 실무에 직접 참여하면서도 일정 수준의 기술적 깊이를 유지할 수 있을까요?
다음은 기술적 능력을 더욱 심화시키고자 하는 건축가들을 위한 몇 가지 팁과 기법입니다.

빈번한 개념 증명(POC)을 수

행하면 아키텍트는 소스 코드를 작성하게 되며, 구현 세부 사항을 고려하여 아키텍처 결정을 검증하는 데 도움이 됩니다. 예를 들어, 아키텍트가 두 가지 캐싱 솔루션 중에서 하나를 선택해야 하는 상황에 처했다고 가정해 보겠습니다. 이 경우 각 캐싱 제품으로 작동하는 예제를 개발하고 결과를 비교하는 것이 효과적인 결정에 도움이 될 수 있습니다. 이를 통해 아키텍트는 구현 세부 사항과 전체 솔루션 개발에 필요한 노력의 양을 직접 확인할 수 있습니다. 또한 확장성, 성능 및 전반적인 내결함성과 같은 다양한 캐싱 솔루션 간의 아키텍처 특성을 더 잘 비교할 수 있습니다.

가능한 한 아키텍트는 최고 수준의 프로덕션 품질 코드를 작성해야 합니다. 이러한 관행을 권장하는 데에는 두 가지 이유가 있습니다. 첫째, 종종 "일회용" 개념 증명 코드가 소스 코드 저장소에 들어가 참조 아키텍처 또는 다른 사람들이 따라야 할 가이드 예제가 되는 경우가 있습니다. 어떤 아키텍트도 자신이 작성한 허술한 일회용 코드가 자신의 일반적인 업무 품질을 대표하는 것으로 여겨지는 것을 원하지 않을 것입니다. 둘째, 프로덕션 품질 코드를 작성하는 것은 여러 가지 이점이 있습니다.

개념 증명 코드란, 빠르고 허술한 개념 증명 코드를 만들어 나쁜 코딩 습관을 계속 들이는 대신, 품질 좋고 구조가 잘 잡힌 코드를 작성하는 연습을 하는 것을 의미합니다.

기술적 부채를 해결하세요

아키텍트가 실무에 직접 참여할 수 있는 또 다른 방법은 기술 부채를 해결하여 개발팀이 핵심 기능 사용자 스토리에 집중할 수 있도록 하는 것입니다. 기술 부채는 일반적으로 우선순위가 낮기 때문에 아키텍트가 특정 스프린트 내에서 기술 부채 해결 작업을 완료하지 못하더라도 큰 문제가 되지 않으며, 일반적으로 스프린트의 성공에 영향을 미치지 않습니다.

버그 수정 또

한 마찬가지로, 스프린트 내에서 버그를 수정하는 것은 개발팀을 지원하면서 실무 코딩 능력을 유지하는 또 다른 방법입니다. 화려한 작업은 아니지만, 이 방법을 통해 아키텍트는 코드베이스와 아키텍처 내의 문제점과 약점을 파악할 수 있습니다.

자동화를 활용

하세요. 간단한 명령줄 도구와 분석기를 만들어 개발팀의 일상적인 작업을 지원하는 것은 개발팀의 효율성을 높이면서 직접 코딩 기술을 유지하는 좋은 방법입니다. 개발팀이 반복적으로 수행하는 작업을 찾아 자동화하세요. 팀원들은 자동화에 감사할 것입니다. 예를 들어, 다른 린트 테스트에서 찾을 수 없는 특정 코딩 표준을 확인하는 자동화된 소스 코드 유효성 검사기, 자동화된 체크리스트, 반복적인 수동 코드 리팩토링 작업 등이 있습니다.

아키텍처 분석 및 적합성 함수 형태의 자동화는 아키텍처의 안정성과 규정 준수를 보장하는 데 유용하며, 실무 경험을 쌓으면서 동시에 실무에 참여할 수 있는 좋은 방법입니다. 예를 들어, 아키텍트는 Java 플랫폼의 ArchUnit 에서 Java 코드를 작성하여 아키텍처 규정 준수를 자동화하거나, 사용자 정의 적합성 함수를 만들어 아키텍처 규정 준수를 보장하면서 실무 경험을 쌓을 수 있습니다. 이러한 기법에 대해서는 6 장 에서 자세히 다룹니다 .

아키텍트로서 실무에 직

접 참여하는 마지막 방법은 빈번한 코드 리뷰를 수행하는 것입니다. 아키텍트가 직접 코드를 작성하지는 않더라도, 이를 통해 소스 코드에 대한 관심을 유지할 수 있습니다. 코드 리뷰의 추가적인 이점으로는 아키텍처 준수 여부를 확인하고 팀 내 멘토링 및 코칭 기회를 파악하는 것이 있습니다.

건축적 사고에는 그 이상의 것이 있다

이 장에서는 건축가처럼 생각하기 시작하는 데 필요한 기초적인 측면들을 다룹니다.
하지만 건축가처럼 생각한다는 것은 이 장에서 설명하는 것보다 훨씬 더 많은 것을 포함합니다.
건축가처럼 생각한다는 것은 시스템의 전체 구조를 이해하는 것(이 주제는 3 장 에서 다룹니다),
비즈니스적 고려 사항을 파악하고 이를 아키텍처적 특징으로 변환하는 것(이 주제에 대해서는 네
장에 걸쳐 다룹니다), 그리고 마지막으로 시스템을 구성하는 논리적 요소, 즉 시스템의 빌딩 블록
을 통해 시스템을 바라보는 것을 의미합니다(이 주제는 8 장 에서 다룹니다).