제27장

소프트웨어 아키텍처의 법칙

다시 방문하다

1 장 에서 우리는 소프트웨어 아키텍처의 세 가지 법칙을 설명했습니다.

- 소프트웨어 아키텍처의 모든 것은 절충의 문제입니다.
- '어떻게'보다 '왜'가 더 중요합니다.
- 대부분의 아키텍처 결정은 이분법적인 것이 아니라 스펙트럼 상에 존재합니다.
  과격한 수단.

이 책에 나오는 거의 모든 예시는 이러한 법칙들을 설명하고 있으며, 이는 법칙들의 기원을 보여줍니다. 초판을 집필할 당시, 우리는 소프트웨어 아키텍처에 대해 보편적으로 적용되는 수많은 사실들을 찾아내어 법칙으로 정리하고자 했습니다. 하지만 놀랍게도 초판에서는 단 두 가지 법칙만을 발견했고, 2판을 집필하면서 한 가지 법칙을 더 찾아냈습니다. 우리의 원래 의도대로, 이 세 가지 법칙은 매우 보편적이며 소프트웨어 아키텍트에게 중요한 여러 관점을 제시합니다.

이 짧은 장에서는 앞서 살펴본 예시들을 바탕으로 이러한 법칙들을 다시 살펴보고, 상충 분석에 대한 몇 가지 미묘한 차이점을 지적하고자 합니다.

제1법칙: 소프트웨어 아키텍처의 모든 것은 절충의 문제다!

우리의 첫 번째 원칙은 소프트웨어 아키텍처의 핵심 특징 중 하나입니다. 바로 모든 것에는 절충점이 있다는 것입니다. 많은 사람들이 소프트웨어 아키텍트의 역할은 까다로운 문제에 대한 만능 해결책을 찾아 영웅이 되는 것이라고 생각하지만, 그런 일은 거의 일어나지 않습니다.

(건축가들은 좋은 결정을 내려도 칭찬받는 경우가 드물지만, 나쁜 결정을 내리면 항상 비난받는다.)
아니요, 소프트웨어 아키텍처의 진정한 역할은 장단점을 분석하는 것입니다.

몇 가지 이유로 모든 건축가는 특정 접근 방식을 옹호하기보다는 절충안을 객관적으로 판단하는 중재자로서 명성을 쌓아야 한다고 생각합니다.

첫째, 아키텍처에서 특정 기술을 맹목적으로 옹호하는 것은 장기적으로 위험합니다. 어제의 모범 사례가 내일의 안티패턴이 되는 경향이 있기 때문입니다. 아키텍트는 불완전한 정보 속에서도 현재의 요소와 상황을 고려하여 최선의 선택을 합니다. 설령 당시에는 타당한 결정이었다 하더라도, 소프트웨어 개발 생태계는 끊임없이 진화하고 변화하기 때문에 상황은 서서히 변하고 결국에는 그 결정의 타당성을 약화시키거나 무효화할 가능성이 높습니다. 만약 아키텍트가 특정 솔루션을 옹호하는 데 사회적 자본을 투자했다면, 나중에 그 결정을 바꿔야 할 때 평판에 손상을 입을 수 있습니다. 따라서 항상 냉철하고 객관적인 시각으로 기술 선택을 해야 하며, 그렇지 않으면 시간이 지나면서 타당성을 잃은 결정에 자신의 신뢰도가 실추될 수 있습니다.

둘째, 조직의 의사 결정권자들은 열정적인 옹호보다는 냉철한 객관성을 더 중요하게 생각합니다. 객관적인 절충 분석을 제공하는 전문가로 명성을 쌓은 설계자는 조직에 매우 귀중한 자산이 됩니다. 중요한 결정을 내려야 할 때, 의사 결정권자들은 판단력을 신뢰할 수 있는 사람을 원합니다. 바로 당신이 그 사람이 될 수 있습니다.

소프트웨어 아키텍처 설계와 관련된 몇 가지 결정 사항에 대한 트레이드오프 분석 사례를 살펴보고, 흔히 발생하는 이해 부족 부분을 논의해 보겠습니다.

공유 라이브러리 vs. 공유 서비스

건축가들이 흔히 겪는 난제는 마이크로서비스나 EDA와 같은 분산 아키텍처에서 공유되는 동작과 관련된 것입니다. 즉, 빌드 시점에 각 서비스에 컴파일되는 공유 라이브러리를 사용해야 할까요, 아니면 그림 27-1에서처럼 다른 서비스가 런타임에 호출하는 공유 서비스를 사용해야 할까요?

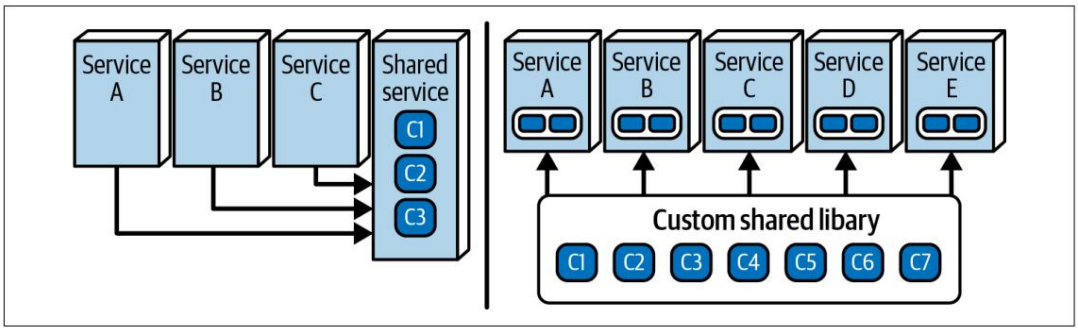

그림 27-1. 공통 기능을 위해 솔루션은 공유 라이브러리를 사용해야 할까요, 아니면 공유 서비스를 사용해야 할까요?

그림 27-1 에서 왼쪽 서비스는 공유 동작을 캡슐화하고, 다른 서비스는 이를 호출하여 접근합니다. 오른쪽에는 배포 전에 각 서비스에 컴파일되는 공유 라이브러리가 있습니다. 어느 쪽이 더 나을까요?

이제 독자 여러분은 이러한 모든 질문에 대한 답을 알고 계실 것입니다. 바로 "상황에 따라 다릅니다!"입니다. 하지만 필연적으로 따라오는 다음 질문에 대한 답을 찾아야 합니다. "무엇에 따라 다르다는 말인가요?" 소프트웨어 아키텍처에서 중요한 결정을 내릴 때처럼, 이 질문에 대한 답은 바로 명확하지 않습니다. 따라서 아키텍트인 여러분이 직접 장단점 분석을 수행해야 할 때입니다.

먼저, 이 솔루션에 영향을 미치는 모든 상황별 장단점을 파악해야 합니다. 이 목록은 조직과 솔루션 모두에 따라 매우 구체적이므로, 조직, 기술 환경, 팀 역량, 예산 및 기타 모든 요소를 고려하여 장단점을 도출해야 합니다. 이 솔루션의 경우, 다음과 같은 관련 요소 목록을 작성했습니다.

이기종 코드 환경에서 솔
: 루션을 여러 플랫폼으로 작성하면, 기술 스택에 관계없이 호출자가 네트워크를 통해 서비스에 접근하므로 구현 플랫폼이 중요하지 않아 서비스 사용이 훨씬 수월해집니다. 하지만 라이브러리의 경우, 각 기술 스택별로 코드 버전을 따로 작성하고 동기화 상태를 유지해야 하므로 프로젝트의 전반적인 복잡성이 크게 증가합니다.

코드 변동성이 높다는
: 것은 문제가 될 수 있습니다. 코드 변동성은 코드가 얼마나 빠르게 변경되는지(일반적으로 '변경 빈도'라고 함)를 측정하는 지표입니다. 서비스에서 호출자는 업데이트된 서비스를 배포하는 즉시 새로운 기능에 접근할 수 있습니다. 따라서 라이브러리를 변경해야 하는 경우, 새로운 기능을 활용하려면 각 서비스를 다시 컴파일하고 배포해야 합니다.

버전 관리 기능 측면에서 라이브러
: 리의 버전 관리는 서비스보다 훨씬 쉽습니다. 라이브러리를 업데이트해야 할 경우, 팀은 컴파일 시점에 버전 차이를 해결하고 필요한 부분만 빌드할 수 있습니다. 서비스의 경우, 버전 정보를 런타임에 확인해야 하므로 서비스와의 상호 작용이 복잡해집니다.

전반적인 변경 위험 측면에
: 서 라이브러리가 더 안전합니다. 라이브러리 코드를 변경하고 서비스에 성공적으로 컴파일하면 정상적으로 작동할 것이라는 확신을 가질 수 있습니다. 반면 서비스는 컴파일 시 검증 없이 변경될 수 있으므로 호출 시 런타임 오류가 발생할 가능성이 높아집니다.

성능 면에서 이
: 라이브러리는 분명히 더 나은 성능을 보여줄 것입니다. 공유 기능에 대한 호출은 서비스에 필요한 네트워크 호출과 달리 프로세스 내 호출이기 때문입니다.
네트워크 지연 및 기타 요인으로 인해 이러한 호출은 처리 중인 호출보다 훨씬 느릴 것입니다.

내결함성

9 장 에서 논의했듯이 서비스에 대한 런타임 접근은 항상 잠재적 위험을 수반합니다.
네트워크 문제, 특히 여기서 문제되는 서비스와 관련된 문제입니다. 도서관에서는 다음과 같은 서비스를 제공합니다.
향상된 내결함성: 서비스를 컴파일, 테스트 및 배포한 후에는 다음과 같은 이점을 누릴 수 있습니다.
안정적이라고 확신합니다.

확장성

성능과 마찬가지로 서비스 간 호출도 지연 시간으로 인해 영향을 받으며, 이는 점점 줄어듭니다!
확장성을 향상시킵니다. 따라서 라이브러리는 접근성이 향상됨에 따라 더 나은 확장성을 제공할 것입니다.
공유 동작은 프로세스 내에서 매우 효율적인 호출입니다.

중요 요소 목록을 정했다면, 이제 매트릭스를 만들어 분석할 수 있습니다.
표 27-1 에 나와 있는 것처럼 각 기준에 대해 각 제품이 얼마나 잘 작동하는지 비교합니다 .

표 27-1. 공유 서비스와 공유 라이브러리 간의 장단점

요인	공유 라이브러리	공유 서비스
이종 코드	-	+
높은 코드 변동성	-	+
버전 변경 기능 +		-
전반적인 변화 위험	+	-
성능	+	-
내결함성	+	-
확장성	+	-

공유 도서관이 가진 여러 긍정적인 요소들을 종합해 보면, 적어도 당분간은 공유 도서관이 승자가 될 것입니다.
이러한 요인들과 이러한 맥락에서 볼 때, 그것이 문제에 대한 해결책이 될 수도 있고 아닐 수도 있습니다.
손—추가적인 가중치를 적용해야 할 수도 있습니다("두 번째 결론: 당신은 할 수 없습니다" 참조).
(자세한 내용은 494페이지의 "딱 한 번만 하세요"를 참조하세요 .) 하지만 이제 당신과 당신의 팀은 다음과 같은 기회를 갖게 되었습니다.
작용하는 여러 요인들을 잘 파악했네요.

동기식 메시징과 비동기식 메시징

다음과 같은 장단점 분석을 고려해 보세요. 여러분의 팀은 분산형 아키텍처를 구축하고 있습니다!
거래 정보를 알림 서비스 와 분석 서비스 모두에 전송하는 구조입니다 .
그리고 당신은 이 기능을 구현하기 위해 큐를 사용할지 토픽을 사용할지 고민하고 있는 겁니다.
행동.

첫 번째 옵션은 큐 또는 지점 간 통신 프로토콜입니다.
2 장 에서 배웠듯이 , 발행인은 누가 메시지를 받는지 알고 있습니다. 도달하려면
여러 소비자가 있는 경우, 게시자는 각 소비자마다 별도의 큐에 메시지를 보내야 합니다.
소비자. 거래 서비스가 분석 서비스 에 알리기 위해 큐를 사용하려는 경우 !
거래 관련 보고 서비스 및 기타 사항의 구현은 다음과 같이 나타납니다.
그림 27-2.

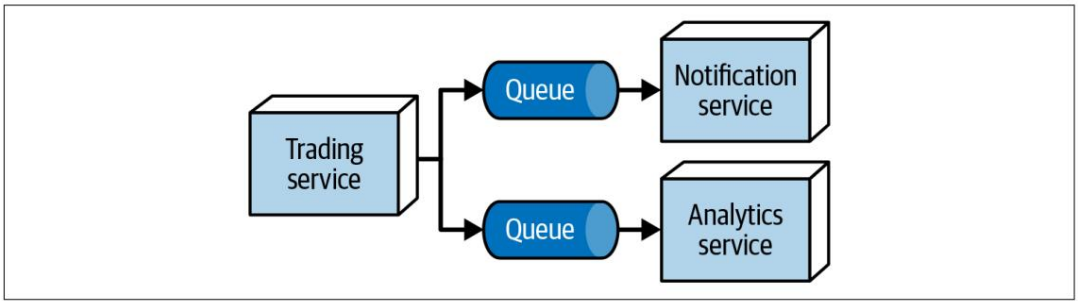

그림 27-2. 큐를 사용하여 거래 정보를 알림 및 분석 기능 으로 전달하는 방법

그림 27-2 에서 거래 서비스 는 큐를 통해 각 소비자에게 메시지를 보냅니다.
각 소비자는 자체 대기열을 가지고 있으며, 발신자는 각 소비자에게 메시지를 게시합니다.

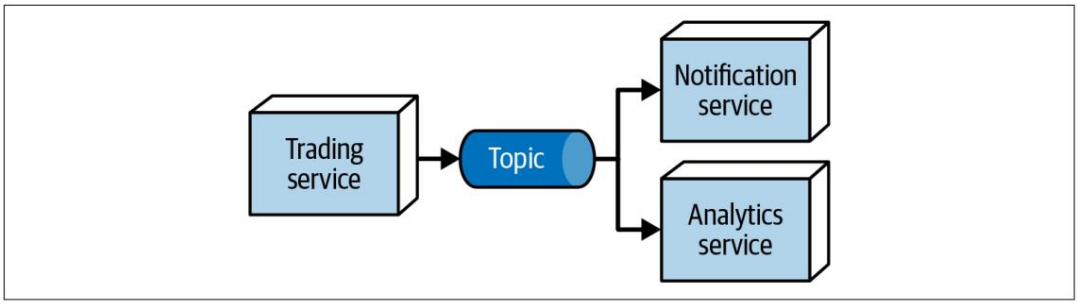

그림 27-3. 토픽을 사용하여 거래 정보를 알림 및 분석 기능 으로 전달하는 방법

대안적인 토폴로지는 토픽을 사용하는데, 이는 지점 간 통신이 아닌 브로드캐스트 방식을 구현합니다. 그림 27-3 에서 거래 서비스 는 하나의 메시지를 토픽에 게시합니다. 각 소비자는 해당 토픽을 구독하고, 메시지가 게시되면 모두 알림을 받습니다. 이 경우 메시지 게시자는 소비자가 누구인지 알 필요가 없으며, 실제로 팀은 다른 소비자나 메시지 발행자를 변경하지 않고도 언제든지 새로운 소비자를 추가할 수 있습니다.

두 가지 선택지 모두 실행 가능하니, 어떻게 결정해야 할까요? 당연히 장단점 분석을 통해 결정하면 됩니다!

큐를 사용하면 트레이딩 서비스가 알림을 보내야 하는 모든 서비스는 각각 별도의 큐를 갖게 됩니다. 알림 서비스와 분석 서비스가 서로 다른 정보를 필요로 하는 경우, 각 큐에 다른 메시지를 보낼 수 있으므로 매우 유용합니다. 트레이딩 서비스는 통신하는 모든 시스템을 파악하고 있으므로, 다른 (악의적인) 서비스가 "도청"하기가 더 어려워집니다. 이는 보안이 최우선 순위인 경우에 특히 효과적입니다. 각 큐는 독립적이므로 개별적으로 모니터링하고 필요에 따라 독립적으로 확장할 수도 있습니다. 트레이딩 서비스는 소비자와 긴밀하게 연결되어 있어 소비자의 수를 정확히 파악할 수 있습니다.

하지만 규정 준수 서비스 로 메시지를 보내야 하는 경우 , 거래 서비스를 수정하여 세 번째 큐로 메시지를 보내도록 해야 합니다 .

표 27-2는 이 프로젝트에서 큐를 사용할 때의 장단점을 요약한 것입니다.

표 27-2. 큐를 사용할 때의 Trade-os

이점	불리
다양한 소비자를 위한 이기종 메시지 지원, 높은 수준의 결합도	
대기열 깊이를 독립적으로 모니터링할 수 있습니다.	거래 서비스는 여러 대기열에 연결되어야 합니다.
더욱 안전함	추가적인 인프라가 필요합니다
확장성이 떨어짐 (소비자 수를 늘리려면 대기열을 추가해야 함)	확장성과 진화 가능성에 대한 훌륭한 지원

토픽에 대해서도 같은 방식으로 살펴보겠습니다. 토픽의 장점은 분명합니다. 토픽은 확장성이 매우 뛰어납니다. 예를 들어 규정 준수에 관심 있는 새로운 소비자는 기존 동작을 전혀 건드리지 않고 해당 토픽을 구독할 수 있습니다. 하지만 이러한 장점에는 단점도 있습니다. 각 소비자는 토픽에서 동일한 메시지를 소비해야 하므로 스탬프 결합(그림 9-9 참조)이 발생할 수 있습니다. 또한 모든 소비자가 전체 메시지를 읽을 수 있기 때문에 보안 문제가 발생할 수 있습니다. 모든 사람이 전체 메시지를 읽을 수 있도록 허용하는 것이 과연 옳은 것일까요?

이러한 관점을 바탕으로 표 27-3 에 나와 있는 것처럼 절충표를 만들 수 있습니다 .

표 27-3. 토픽 사용 시 Trade-os

이점	불리
낮은 결합	각 소비자에게 동일한 메시지
거래 서비스는 메시지를 하나만 생성합니다. 개별 소비자를 모니터링하거나 확장할 수 없습니다.	
더 확장 가능하고 진화 가능성이 높음	보안 수준이 낮음
	확장성 옵션이 적습니다

이제 장단점 분석을 마쳤으니, 이 솔루션에 대한 조직의 목표를 다시 살펴보고 어떤 옵션이 더 적합한지 판단해야 합니다. 보안이 더 중요하다면 큐 방식을 선택하는 것이 적절할 것입니다. 조직이 빠르게 성장하고 있고 거래에 관심 있는 다른 서비스가 있다면 확장성이 우선시될 수 있으며, 이 경우 토픽 방식을 사용하는 것이 더 적합할 것입니다.

첫 번째 결론: 누락된 거래-O!

우리의 첫 번째 법칙에는 두 가지 결과가 따른다. 첫 번째는 다음과 같다.

> 만약 당신이 어떤 것을 발견했는데 그것이 상충 관계가 아니라고 생각한다면, 아마도 당신은 아직 상충 관계를 파악하지 못했을 가능성이 더 큽니다.
>
> —1차 정리

소프트웨어 아키텍처 결정의 핵심은 절충 분석이지만, 절충점이 전혀 없어 보이는 결정을 마주했을 때는 어떻게 해야 할까요? 저희의 조언은 다음과 같습니다. 계속해서 다른 방법을 찾아보세요!

코드 재사용을 생각해 보세요. 분명 이는 순전히 이로운 관행이죠? 조직이 재사용할 수 있는 코드가 많을수록 작성해야 하는 코드가 줄어들어 시간과 중복을 절약할 수 있습니다.

코드 재사용의 효율성을 결정하는 두 가지 요소가 있습니다. 아키텍트는 종종 첫 번째 요소는 파악하지만 두 번째 요소는 간과합니다. 첫 번째 요소는 추상화입니다. 코드를 추상화하여 여러 호출 지점에서 사용할 수 있다면 재사용하기에 좋은 후보입니다. 두 번째 요소는 낮은 변동성입니다. 팀이 항상 변경되는 코드 모듈을 재사용하면 전체 시스템에 혼란이 발생합니다. 공유 코드가 변경될 때마다 해당 코드를 호출하는 모든 부분이 변경 사항에 맞춰 조정해야 합니다. 호환성을 깨뜨리는 변경 사항이 아니더라도 팀은 변경으로 인해 문제가 발생하지 않았는지 확인해야 합니다. 아키텍처에서 코드를 부적절하게 재사용하면 팀은 아키텍처 전체에서 호환성을 깨뜨리는 변경 사항을 추적하는 데 시간을 허비하게 됩니다. 이는 아키텍트 분야가 오케스트레이션 기반 SOA 아키텍처 스타일(17 장 에서 다룸)에서 얻은 중요한 교훈 중 하나이며, 해당 스타일의 기본 철학 중 하나는 가능한 한 많은 코드를 재사용하는 것이었습니다.

실질적인 관점에서 볼 때, 그러한 아키텍처에서 작업하는 팀은 마치 늪에 빠진 것처럼 어려움을 겪었습니다. 모든 변경 사항은 시스템 전체에 예측할 수 없는 부작용을 일으킬 가능성이 있었기 때문입니다.

숨겨진 상충 관계는 바로 이것입니다. 코드를 효과적으로 재사용하려면 뛰어난 추상화와 낮은 변동성이 필요합니다. 이것이 바로 아키텍처에서 가장 성공적인 재사용 대상이 기술 프레임워크, 라이브러리, 플랫폼 등과 같은 "기반 구조"인 이유입니다. 대부분의 애플리케이션에서 가장 빠르게 변화하는 부분은 도메인(소프트웨어를 개발하게 된 근본적인 동기)이므로, 도메인 개념은 재사용하기에 적합하지 않습니다.
(이는 DDD의 경계 컨텍스트 원칙의 근간을 이루는 내용입니다. 즉, 어떤 경계 컨텍스트도 다른 경계 컨텍스트의 구현 세부 사항을 재사용할 수 없습니다.)

우리가 좋은 것을 가질 수 없는 이유—거래!

전문 컨설턴트로서 저희는 고객들로부터 다음과 같은 요청을 자주 받습니다. "마이크로서비스와 높은 수준의 결합도를 가진 분산 아키텍처는 민첩성과 빠른 배포를 가능하게 해주기 때문에 저희는 이러한 아키텍처를 선호합니다. 하지만 동시에 팀들이 코드를 끊임없이 다시 작성하지 않도록 높은 수준의 재사용성도 원합니다."

안타깝게도 좋지 않은 소식을 전해드려야 할 것 같습니다. 두 가지를 모두 갖는 것은 불가능합니다. 시스템에서 재사용성을 구현하는 방식은 결합도를 통해서만 가능하기 때문입니다. 어떤 조직도 결합도 낮추면서 높은 수준의 재사용성을 동시에 달성할 수는 없습니다. 이 두 가지는 근본적으로 양립할 수 없습니다. 이는 조직이 핵심적인 상충 관계를 오해할 수 있는 대표적인 사례입니다.

두 번째 결론: 한 번에 끝낼 수 없다. 설계자가 단 한 번의 트레이

드오프 분석으로 모든 워크플로에 안무를 적용하기로 결정할 수 있다면 좋겠지만, 트레이드오프 분석에는 두 가지 문제가 있습니다. 첫째, 의사결정에 영향을 미치는 변수(기술적 변수 및 기타 변수)가 수십, 심지어 수백 개에 달하는 경우가 많습니다. 복잡성, 팀 경험, 예산, 팀 구성, 일정 압박 등 그 목록은 끝이 없습니다. 이러한 변수의 미묘한 차이로 인해 분석 방향이 한쪽으로 쏠릴 수 있으므로, 트레이드오프 분석은 지속적인 작업이 됩니다. 설계자가 향후 이 솔루션을 적용할 때 유효하지 않을 수 있는 가정에 기반하여 포괄적이고 반영구적인 결정을 내리는 것은 위험합니다.

이러한 결론을 건축가의 직업 안정성으로 생각해 보세요. 우리는 겉보기에 비슷한 상황에서도 끊임없이 절충 분석을 해야 합니다. 이는 건축가의 진정한 업무는 영구적이고 완벽한 결정을 내리는 것이 아니라 절충 분석을 하는 데 있다는 생각을 더욱 강화시켜 줍니다.

제2법칙: '왜'가 '어떻게'보다 더 중요하다

두 번째 원칙은 '어떻게'보다 '왜'의 중요성을 강조합니다. 경험 많은 건축가로서 우리는 기존 시스템을 살펴보고 그 작동 방식을 설명할 수 있습니다. 하지만 이전 건축가가 최종 해결책을 결정할 때 모든 기준을 기록해 두지 않았기 때문에 왜 다른 선택지 대신 이 옵션을 선택했는지 명확하지 않은 경우가 있습니다.

이것이 바로 아키텍처 다이어그램과 ADR(21 장 에서 다룸)을 모두 활용하는 것이 중요한 이유입니다 . 모든 트레이드오프 분석은 솔루션에 나타나지 않는 엄청난 양의 맥락 정보를 생성합니다. 미래의 아키텍트(바로 당신일 수도 있습니다)가 이유를 이해하기 위해 분석을 다시 수행해야 하는 상황을 방지하려면 해당 분석(및 솔루션의 알려진 타협점과 한계점)을 문서화하는 것이 매우 중요합니다.

맥락을 벗어난 안티패턴은 트레이드오프

분석에서 흔히 나타나는 안티패턴 중 하나입니다.
이러한 안티패턴은 아키텍트가 장단점을 이해하지만 현재 상황에 따라 모든 장단점에 가중치를 부여하는 방법을 모를 때 발생합니다.

488페이지의 "공유 도서관 대 공유 서비스" 에서 수행한 절충 분석을 생각해 보십시오. 객관적으로 보면 공유 도서관이 긍정적인 평가가 부정적인 평가보다 더 많았으므로 선호되는 해결책으로 보입니다. 그러나 이 절충 분석에는 잠재적인 결함이 있습니다. 모든 기준에 동일한 가중치가 부여되었을까요?

한 팀이 여러 플랫폼에서 코드를 작성한다고 가정해 보세요. 성능이나 확장성은 크게 신경 쓰지 않지만, 공통된 동작을 깔끔하게 관리하고 싶어 합니다. 이 경우, 앞서 언급한 두 가지 기준이 훨씬 더 높은 우선순위를 가지게 되므로, 공통된 동작을 관리하는 방법을 먼저 고려해야 합니다.

공유 서비스를 선택하기로 했습니다. 게다가, 팀은 이미 어떤 문제들을 해결해야 할지 파악해 놓았습니다.

건축가는 경험을 바탕으로 절충 기준을 세우지만, 동시에 이러한 기준들을 고려하여 최적의 해결책을 찾아야 합니다. 일반적인 절충 분석은 그다지 유용하지 않으며, 특정 맥락에 적용될 때 비로소 가치를 지닙니다.

극단 사이의 스펙트럼

초판에서는 두 가지 법칙만을 제시했지만, 점차 세 번째 법칙이 존재한다는 사실을 깨달았습니다.

> 대부분의 아키텍처 결정은 이분법적인 것이 아니라 스펙트럼 상에 존재합니다.
> 과격한 수단.
>
> —소프트웨어 아키텍처의 세 번째 법칙

명확하고 깔끔한 이분법적 결정이 가능한 세상에서 살면 좋겠지만, 소프트웨어 아키텍처는 그런 세상에 존재하지 않습니다. 사람들은 소프트웨어 아키텍처의 중요한 개념들, 예를 들어 아키텍처와 디자인, 오케스트레이션과 안무, 토픽과 큐 등에 대한 포괄적인 정의를 내리는 것이 얼마나 어려운지 종종 이야기합니다. 근본적인 이유는 결정 기준이 이분법적이지 않고, 오히려 복잡한 스펙트럼 상에 존재하기 때문입니다.

17페이지의 "아키텍처 대 디자인" 에서 우리는 아키텍처와 디자인 사이의 스펙트럼과 특정 결정이 그 스펙트럼 내에서 어디에 위치하는지 판단하는 방법에 대해 논의했습니다. 실제로 이 법칙은 소프트웨어 아키텍처 쪽에 더 가깝고 디자인 쪽에 덜 가까운 결정을 생각하는 데 유용한 방법을 제공합니다.

> 소프트웨어 아키텍처를 결정할 때는 각 옵션마다 상당한 장단점이 있다는 점을 고려해야 합니다.

소프트웨어 아키텍처의 모든 것이 절충의 문제라면, 아키텍처 관련 결정을 내릴 때에도 각 옵션에 대한 절충안을 고려해야 합니다.

아키텍트로서 모든 결정을 이분법적으로 생각하려고 하지 마세요. 세상에는 그런 이분법적인 결정은 거의 없습니다. 소프트웨어 아키텍처에서 모든 질문에 대한 답이 "상황에 따라 다릅니다"인 이유 중 하나는, 해당 기준이 가능한 해결책의 스펙트럼 상에서 어디에 위치하는지에 따라 답이 달라지기 때문입니다.

건축가로서 우리는 불확실성의 늪에서 결정을 내려야 합니다. 이러한 상황에 적응하는 것은 번거롭지만 필수적입니다. 중요한 결정을 불완전한 정보에 기반해야 할 때도 있지만, 모든 정보를 다 알고 있다 하더라도 명확한 결론에 도달하기 어려운 경우가 많습니다. 왜냐하면 그 결정은 불확실성의 스펙트럼 어딘가에 위치하기 때문입니다.

두 가지 극단적인 경우.

소프트웨어 아키텍처에 오신 것을 환영합니다!

마지막으로 드리는 조언

> 어떻게 하면 훌륭한 디자이너를 얻을 수 있을까요? 훌륭한 디자이너는 당연히 디자인을 합니다.
>
> —프레드 브룩스

> 건축가들이 평생 동안 건축 설계를 할 기회가 여섯 번도 채 되지 않는다면, 어떻게 훌륭한 건축가를 배출할 수 있겠습니까?
>
> —테드 뉴어드

연습은 기술을 연마하고 실력을 향상시키는 가장 확실한 방법이며, 건축 또한 예외는 아닙니다. 저희는 신진 및 기존 건축가들이 건축 설계 기술을 연마하는 것은 물론, 각자의 기술적 역량을 넓혀나가도록 장려합니다. 이를 위해 본서에 수록된 예시들을 바탕으로 건축 카타(kata)를 제작 하여 웹사이트에 게시했습니다. 이 카타를 활용하여 건축 기술을 연습하고 발전시켜 나가시기를 바랍니다.

저희 카타를 사용하시는 분들이 자주 묻는 질문이 있습니다. "혹시 정답 해설 같은 게 있나요?" 안타깝게도 없습니다. 닐의 말을 인용하자면,

> 건축에는 옳고 그른 답이 없습니다. 오직 절충안만 있을 뿐입니다.

저자들이 실시간 교육 수업에서 아키텍처 카타를 처음 활용했을 때, 학생들의 도면을 모아 정답 저장소로 만들려는 목적으로 보관했었습니다. 하지만 곧 그 계획은 포기했습니다. 도면이 불완전한 자료라는 것을 깨달았기 때문입니다. 학생들은 토폴로지 도면을 통해 솔루션 구현 방식을 보여주었지만, 그 이면에 숨겨진 '이유'는 훨씬 더 중요했습니다. 수업 시간에는 학생들이 고려했던 장단점을 설명했지만, 아키텍처 결정 과정을 기록할 시간은 부족했습니다. '방법'만 기록하는 것은 전체 이야기의 절반만을 얻는 것과 마찬가지였습니다.

자, 마지막으로 드리고 싶은 조언은 다음과 같습니다. 항상 배우고, 항상 연습하고, 건축에 뛰어드세요!

토론 질문

제 1장: 서론

1. 소프트웨어 아키텍처를 정의하는 네 가지 차원은 무엇입니까?
2. 아키텍처 결정과 디자인 원칙의 차이점은 무엇인가요?
3. 소프트웨어 아키텍트에게 기대되는 8가지 핵심 사항을 나열하십시오.
4. 소프트웨어 아키텍처의 첫 번째 법칙은 무엇입니까?

제 2장: 건축적 사고

1. 어떤 결정이 건축에 관한 것인지, 디자인에 관한 것인지를 판단하는 세 가지 기준을 제시하시오.
2. 지식 삼각형의 세 가지 지식 수준을 나열하고 각 수준에 대한 예를 제시하십시오.
3. 건축가가 기술적 폭에 집중하는 것이 특정 기술 분야에 집중하는 것보다 더 중요한 이유는 무엇일까요?
   기술적 깊이보다 더 깊은가요?
4. 아키텍트로서 기술적 전문성을 유지하고 실무에 직접 참여하는 방법에는 어떤 것들이 있을까요?

제 3장: 모듈성

1. 모듈성과 세분성의 차이점을 설명하고, 각각의 예를 제시하십시오.
2. 결합도와 응집도의 차이점은 무엇인가요?
3. 'connascence'라는 용어는 무슨 뜻인가요?

4. 정적 연결과 동적 연결의 차이점은 무엇입니까?
5. 가장 강력한 형태의 연결은 무엇입니까?
6. 결합의 가장 약한 형태는 무엇입니까?
7. 코드베이스 내에서 정적 연결과 동적 연결 중 어느 것이 더 바람직합니까?

제4 장 : 건축적 특징의 정의

1. 속성이 아키텍처로 간주되려면 어떤 세 가지 기준을 충족해야 합니까?
   특성?
2. 암묵적 특징과 명시적 특징의 차이점은 무엇입니까?
   각각의 예를 들어 설명하십시오.
3. 운영상의 특징에 대한 예를 제시하십시오.
4. 구조적 특징의 예를 제시하시오.
5. 모든 분야에 공통적으로 나타나는 특성의 예를 제시하시오.
6. 업계 전반에 걸쳐 표준 아키텍처 목록을 만드는 것이 불가능한 이유는 무엇입니까?
   형질?

제5 장 : 건축적 특징 파악하기

1. 특성 개수를 제한하는 것이 바람직한 이유를 설명하세요.
   아키텍처가 지원해야 하는 ("-ilities") 기능.
2. 참 또는 거짓? 대부분의 아키텍처 특성은 비즈니스 요구사항에서 비롯된다.
   그리고 사용자 스토리.
3. 비즈니스 이해관계자가 시장 출시 시간(즉, 새로운 기능과 버그 수정 사항을 사용자에게 최대한 빨리 배포하는 것)이 가장 중요한 비즈니스 문제라고 말한다면, 해당 아키텍처는 어떤 특징을 지원해야 할까요?
4. 확장성과 탄력성의 차이점은 무엇인가요?
5. 복합적인 건축적 특징이란 무엇인가? 하나를 제시하시오.
   예.

제 6장: 건축의 측정과 관리 형질

1. 순환 복잡도가 분석에 있어 왜 그렇게 중요한 지표인가요?
   건축학?
2. 아키텍처 적합도 함수란 무엇이며, 이를 사용하여 아키텍처를 분석하는 방법은 무엇입니까?
   건축학?
3. 아키텍처의 확장성을 측정하는 아키텍처 적합도 함수의 예를 제시하십시오.
4. 건축가와 개발자가 적합성 함수를 생성할 수 있도록 하는 아키텍처 특성의 가장 중요한 기준은 무엇입니까?

제 7장: 건축적 특징의 범위

1. 건축 양자란 무엇이며, 건축에 왜 중요한가요?
2. 하나의 사용자 인터페이스와 각각 별도의 데이터베이스를 포함하는 네 개의 독립적으로 배포된 서비스로 구성된 시스템을 가정해 봅시다. 이 시스템은 양자(quantum)가 하나일까요, 아니면 네 개일까요? 그 이유는 무엇일까요?
3. 정적 결합과 동적 결합의 차이점은 무엇이며, 예를 들어 설명하십시오.
   각각의 것.
4. 동기식 통신이 비동기식 통신보다 운영 아키텍처 특성에 더 큰 잠재적 영향을 미치는 이유는 무엇입니까?

제 8장: 구성 요소 기반 사고

1. 컴포넌트란 애플리케이션의 구성 요소, 즉 애플리케이션이 수행하는 기능을 의미합니다. 컴포넌트는 일반적으로 클래스 또는 소스 파일 그룹으로 구성됩니다. 애플리케이션이나 서비스 내에서 컴포넌트는 일반적으로 어떤 형태로 나타날까요?
2. 기술적 파티셔닝과 도메인 파티셔닝의 차이점은 무엇입니까?
   각각의 예를 들어 설명하십시오.
3. 도메인 분할의 장점은 무엇입니까?
4. 어떤 상황에서 기술적 분할이 다른 방식보다 더 나은 선택이 될까요?
   도메인 분할?
5. 엔티티 트랩이란 무엇이며, 컴포넌트에 적합하지 않은 이유는 무엇입니까?
   신분증?

6. 핵심 구성 요소를 식별할 때 행위자/행동 접근 방식보다 워크플로 접근 방식을 선택하는 경우는 언제입니까?

제 9장: 기초

1. 분산 컴퓨팅의 여덟 가지 오류를 나열하시오.
2. 분산 아키텍처가 단일체 아키텍처에 비해 가지는 세 가지 어려움을 설명하세요!
   건축 양식은 그렇지 않습니다.
3. 스탬프 커플링이란 무엇이며, 스탬프 커플링 문제를 해결하는 방법에는 어떤 것들이 있을까요?
4. 기술적 분할과 도메인 분할의 차이점은 무엇입니까?
5. 건축 양식과 패턴을 구분하는 세 가지 특징을 나열하시오.

제10장: 겹겹이 쌓인 건축 양식

1. 열린 레이어와 닫힌 레이어의 차이점은 무엇입니까?
2. 격리 계층 개념과 그 이점에 대해 설명하십시오.
3. 아키텍처 싱크홀 안티패턴이란 무엇인가요?
4. 계층형 아키텍처를 사용하게 만드는 주요 아키텍처 특징은 무엇입니까?
5. 계층형 아키텍처 방식에서 테스트 용이성이 제대로 지원되지 않는 이유는 무엇입니까?
6. 계층형 아키텍처 스타일에서 애자일 개발 방식이 제대로 지원되지 않는 이유는 무엇입니까?

제11장: 모듈형 모놀리스 아키텍처 스타일

1. 모듈형 모놀리스는 n계층형 아키텍처와 어떻게 다른가요?
2. 모듈 간 의사소통 시 동료 간 소통 방식과 중재자 방식의 차이점을 설명하십시오.
3. 모듈형 모놀리스 아키텍처와 관련된 세 가지 일반적인 위험은 무엇입니까?
   건축 양식?
4. 모듈형 모놀리스의 주요 장점 세 가지와 사용을 고려해야 할 시점을 설명하십시오.
5. 모듈형 구조를 고려할 때, 확장성 및 기타 운영 특성은 왜 고려되지 않는 것입니까?
   모듈형 모놀리식 구조에서 내결함성은 우수한가요?

제12장: 파이프라인 아키텍처 스타일

1. 파이프라인 구조에서 파이프는 양방향일 수 있습니까?
2. 필터의 네 가지 유형과 각각의 용도를 설명하시오.
3. 필터가 여러 파이프를 통해 데이터를 전송할 수 있습니까?
4. 파이프라인 아키텍처가 클라우드 기반 환경에 적합한 이유를 설명하십시오.
   환경.
5. 파이프라인 아키텍처 스타일은 기술적으로 파티션 방식인가요, 아니면 도메인 파티션 방식인가요?
6. 파이프라인 아키텍처는 어떤 방식으로 모듈성을 지원합니까?

제13장: 마이크로커널 아키텍처 스타일

1. 마이크로커널 아키텍처 스타일을 다른 말로 무엇이라고 부릅니까?
2. 플러그인 구성 요소가 다른 구성 요소에 의존하는 것이 허용되는 상황은 어떤 경우입니까?
   플러그인 구성 요소?
3. 플러그인을 관리하는 데 사용할 수 있는 도구 및 프레임워크에는 어떤 것들이 있습니까?
4. 타사 플러그인이 당사의 규정을 준수하지 않는 경우 어떻게 하시겠습니까?
   핵심 시스템의 표준 플러그인 계약인가요?
5. 마이크로커널 아키텍처 스타일의 예시를 두 가지 제시하시오.
6. 코어 시스템의 어떤 특성이 "마이크로커널성"의 정도를 결정하는가?
7. 마이크로커널 아키텍처가 항상 단일 아키텍처 양자 컴퓨팅인 이유는 무엇입니까?
8. 도메인/아키텍처 동형성이란 무엇인가요?

제14장: 서비스 기반 아키텍처 스타일

1. 서비스 기반 아키텍처에서 서비스를 도메인 서비스라고 부르는 이유는 무엇입니까?
2. 서비스 기반 아키텍처와 관련된 일반적인 위험 요소 두 가지를 설명하십시오.
3. 어떤 데이터베이스 토폴로지(모놀리식, 도메인, 전용)를 사용할 수 있습니까?
   서비스 기반 아키텍처를 사용하면 어떨까요?
4. 서비스 기반 환경에서 데이터베이스 변경 사항을 관리하기 위해 어떤 기술을 사용할 수 있습니까?
   건축학?
5. 도메인 서비스를 실행하려면 컨테이너(예: Docker)가 필요한가요?
6. 서비스 기반 아키텍처 스타일은 어떤 아키텍처적 특징을 지원합니까?
   항구 잘 있나요?
7. 서비스 기반 아키텍처에서 탄력성이 제대로 지원되지 않는 이유는 무엇입니까?

8. 서비스 기반 아키텍처에서 아키텍처 양자의 수를 어떻게 늘릴 수 있습니까?
건축학?

제15장: 이벤트 기반 아키텍처 스타일

1. 사건과 메시지의 차이점 네 가지를 설명하세요.
2. 시작 이벤트와 파생 이벤트의 차이점은 무엇인가요?
3. 독성 사건이란 무엇인가요?
4. 이벤트 프로세서가 여러 파생 이벤트를 트리거할 수 있습니까? 그렇다면 그 이유는 무엇입니까?
이거 하고 싶어?
5. 다른 이벤트 프로세서가 응답하지 않는 이벤트를 특정 이벤트 프로세서에서 발생시키는 이유는 무엇입니까?
6. 비동기 처리를 사용할 때 발생할 수 있는 부정적인 단점은 무엇입니까?
7. 모기떼 방제 패턴(Swarm of Gnats antipattern)을 설명하고, 왜 이를 피해야 하는지 설명하십시오.
8. 이벤트 기반 아키텍처가 높은 응답성과 뛰어난 성능 특성을 갖는 주요 이유 두 가지를 제시하십시오.
9. 큐를 통해 메시지를 송수신할 때 데이터 손실을 방지하는 기술에는 어떤 것들이 있을까요?

제16장: 공간 기반 건축 양식

1. 우주 기반 건축이라는 용어는 어디에서 유래되었습니까?
2. 공간 기반 건축을 다른 건축 양식과 구별 짓는 주요 특징은 무엇입니까?
3. 우주 기반 아키텍처 내에서 가상화된 미들웨어를 구성하는 네 가지 구성 요소를 나열하십시오.
4. 우주 기반 아키텍처에서 데이터 기록자의 역할은 무엇입니까?
5. 서비스가 데이터베이스의 데이터에 접근해야 하는 경우는 어떤 조건에서 발생할까요?
데이터 판독기 말인가요?
6. 캐시 크기가 작을수록 데이터 충돌 가능성이 높아지나요, 낮아지나요?
7. 복제 캐시와 분산 캐시의 차이점은 무엇입니까?
하나는 주로 우주 기반 건축에서 사용되는 것인가요?
8. 우주 기반 시스템에서 가장 강력하게 지지받는 아키텍처 특성 세 가지를 나열하시오.
건축학.
9. 우주 기반 아키텍처의 테스트 용이성 평가가 왜 이렇게 낮은가요?

제17장: 오케스트레이션 기반 서비스 지향 건축학

1. 서비스 지향 아키텍처의 주요 원동력은 무엇이었습니까?
2. 서비스 지향 아키텍처에서 네 가지 주요 서비스 유형은 무엇입니까?
3. 서비스 지향 아키텍처의 몰락을 초래한 요인들을 몇 가지 나열하시오.
4. SOA는 기술적으로 파티션 방식인가요, 아니면 도메인 파티션 방식인가요?
5. SOA에서 도메인 재사용은 어떻게 처리되나요? 운영 재사용은 어떻게 처리되나요?
6. 현대 시스템에서 이러한 아키텍처 스타일의 주요 용도는 무엇입니까?

제18장: 마이크로서비스 아키텍처

1. 마이크로서비스에서 경계 컨텍스트(Bounded Context) 개념을 설명하고, 왜 이 개념이 중요한지 설명하십시오.
   마이크로서비스 아키텍처?
2. 마이크로서비스 아키텍처에서 데이터를 격리하는 것이 왜 그렇게 중요한가요?
3. 프로토콜 인식 이기종 상호 운용성이란 무엇이며 마이크로서비스가 이를 어떻게 지원하는지 설명하십시오.
4. 마이크로서비스에서 오케스트레이션과 안무의 차이점은 무엇인가요? 각각의 커뮤니케이션 스타일이 적절하게 선택될 수 있는 사례를 제시하세요.
5. 마이크로서비스는 왜 서비스별 데이터베이스 토폴로지를 사용하나요? 다른 방법은 없나요?
   단일형 데이터베이스 토폴로지 또는 도메인형 데이터베이스 토폴로지?
6. 마이크로서비스 아키텍처에서 가장 큰 위험 요소 두 가지는 무엇입니까?
7. 마이크로서비스에서 민첩성, 테스트 용이성, 배포 용이성이 그토록 잘 지원되는 이유는 무엇인가요?
8. 마이크로서비스에서 성능 문제가 흔히 발생하는 세 가지 이유는 무엇입니까?
9. 마이크로서비스 생태계에 단 하나의 서버만 존재하는 토폴로지를 설명하십시오.
   양자.

제19장: 적절한 건축 양식 선택하기

1. 시스템의 데이터 아키텍처(논리적 및 데이터 구조)는 어떤 방식으로 시스템의 데이터 아키텍처에 영향을 미칩니까?
   물리적 데이터 모델이 아키텍처 스타일 선택에 영향을 미치나요?
2. 아키텍트가 시스템의 아키텍처 스타일을 결정하는 데 사용하는 단계를 설명하십시오.
   데이터 분할 및 통신 방식.
3. 어떤 요인이 아키텍트로 하여금 분산 아키텍처를 선택하게 하는가?

4. 아키텍처가 선택을 위해 수행해야 하는 두 가지 입력 분석은 무엇입니까?
적절한 건축 양식인가요?

제20장: 건축 패턴

1. 운영 관련 사항과 도메인 관련 사항의 분리라는 아키텍처적 고려 사항을 구현하는 두 가지 패턴을 제시하시오.
2. 오케스트레이션된 워크플로우에는 있지만 코레아그래피된 워크플로우에는 없는 구성 요소는 무엇입니까?
3. 이벤트 기반 아키텍처에서 단일 브로커를 사용하는 장점은 무엇이며, 단점은 무엇입니까?
4. CQRS는 어떤 두 가지 데이터 연산을 분리합니까?

제21장: 건축적 결정

1. 자산 보호 안티패턴이란 무엇인가요?
2. 이메일 중심의 아키텍처를 피하기 위한 몇 가지 기법은 무엇인가요?
안티패턴?
3. 마이클 나이가드는 건축적으로 중요한 것을 식별하는 다섯 가지 요소로 무엇을 제시하는가?
4. 아키텍처 결정 기록의 다섯 가지 기본 구성 요소는 무엇입니까?
5. 일반적으로 ADR의 어느 부분에 아키텍처에 대한 정당성을 추가합니까?
진정한 결정인가요?
6. 별도의 대안 섹션이 필요하지 않다고 가정할 때, 어느 섹션에 포함시켜야 할까요?
ADR에서 제안하신 해결책에 대한 대안들을 나열해 주시겠습니까?
7. ADR의 상태를 '해결되지 않음'으로 표시하는 세 가지 기본적인 시나리오는 무엇입니까?
청혼했나요?

제22장: 아키텍처 위험 분석

1. 위험 평가 매트릭스의 두 가지 차원은 무엇입니까?
2. 리스크 스토밍의 세 가지 주요 활동을 설명하십시오.
3. 위험 분석이 협업적인 활동이어야 하는 이유는 무엇입니까?
4. 위험 요소 분석 과정에서 식별 활동이 협업 활동이 아닌 개별 활동이어야 하는 이유는 무엇입니까?
5. 만약 세 명의 참가자가 건축물의 특정 영역에 대해 위험을 높음(6)으로 식별했지만 다른 참가자 한 명이 위험을 중간(3)으로만 식별했다면 어떻게 하시겠습니까?

6. 검증되지 않았거나 알려지지 않은 기술에 대해 어떤 위험 등급(1~9)을 부여하시겠습니까?

제23장: 아키텍처 다이어그램 작성

1. 비이성적인 인공물 애착이란 무엇이며, 그것이 어떤 점에서 중요한가?
   아키텍처를 문서화하고 다이어그램으로 나타내는 것인가요?
2. C4 모델링 기법의 4C는 무엇입니까?
3. 아키텍처 다이어그램을 작성할 때, 구성 요소 사이의 점선은 무엇을 의미합니까?
   평균?
4. 아키텍처 다이어그램에 제목과 범례를 항상 포함하는 것이 중요한 이유는 무엇입니까?

제24장: 효과적인 팀 만들기

1. 건축가의 성격 유형은 세 가지가 있으며, 각 유형은 어떤 제약을 만들어내는가?
2. 참여 수준을 결정할 때 고려해야 할 다섯 가지 요소는 무엇입니까?
   팀과 함께요?
3. 팀 규모가 너무 커지고 있다는 세 가지 경고 신호는 무엇입니까?
4. 개발팀에서 활용하기 좋은 기본적인 체크리스트 세 가지를 나열하세요.

제25장: 협상 및 리더십 기술

1. 건축가에게 협상 능력이 왜 그토록 중요한가요?
2. 비즈니스 이해관계자가 가용성의 99.9 ...
3. 비즈니스 이해관계자가 "저는 그게 필요했어요"라고 말할 때 무엇을 알 수 있을까요?
   어제"?
4. 시연을 통한 토론 무력화 기법이 그토록 효과적인 이유는 무엇일까요?
5. 분할 정복 법칙이란 무엇인가요? 비즈니스 이해관계자와 아키텍처 특성을 협상할 때 이 법칙을 어떻게 적용할 수 있나요? 예를 들어 설명해 주세요.
6. 건축의 4C를 나열하시오.
7. 건축가가 실용적이면서도 미래지향적이어야 하는 이유를 설명하십시오.
8. 참석해야 하는 회의 횟수를 관리하고 줄이는 데 도움이 되는 몇 가지 방법은 무엇인가요?

제26장: 건축의 교차점

1. 건축 설계와의 구조적 정렬을 보장하기 위해 어떤 기술을 사용할 수 있습니까?
   구현 중에 문제가 발생합니까?
2. 아키텍처와 인프라를 일치시키는 것이 왜 그렇게 중요한지 설명하고, 그 근거를 제시하십시오.
   이러한 정렬의 예입니다.
3. 팀의 토폴로지가 목표와 일치하지 않을 수 있는 경우의 예를 드시오.
   건축학.
4. 특정 시스템을 담당하는 아키텍트는 왜 아키텍처와 시스템 통합의 교차점에 관심을 가져야 할까요? 아키텍처가 시스템 통합과 어긋날 수 있는 사례를 제시하세요.
5. 도메인/아키텍처 동형성이란 무엇이며, 아키텍처와 비즈니스의 교차점을 고려할 때 왜 그렇게 중요한가요?
6. 건축과 기업이 만나는 지점의 예를 제시하십시오.

제27장: 소프트웨어 아키텍처의 법칙 재고찰

1. 공유 라이브러리가 공유 서비스에 비해 갖는 일반적인 장점 세 가지를 나열하십시오. 이러한 장점들이 아키텍트가 두 가지 선택지 사이에서 고민할 때 항상 공유 라이브러리를 선택해야 한다는 것을 의미합니까?
2. 소프트웨어 아키텍처 제1법칙의 첫 번째 귀결은 무엇입니까?
3. 소프트웨어 아키텍처 제1법칙의 두 번째 귀결은 왜 특정 내용을 강조하는가?
   절충 분석을 계속해서 반복해야 한다는 말인가요?
4. 소프트웨어 아키텍처 결정 사항 중 상당수가 편리한 이분법적 구분보다는 스펙트럼으로 이해하는 것이 더 나은 이유는 무엇일까요?