

O'REILLY®



기초
소프트웨어
건축학

현대 공학적 접근 방식

마크 리처즈 & 닐 포드

"현대적인 관점에서 최신 소프트웨어 아키텍처를 탐구하는 데 없어서는 안 될 자료입니다. 우연히 아키텍트 역할을 맡게 된 사람인가, 실력을 갈고닦고자 하는 베테랑인가, 이 책은 여러분이 전문가로서 탁월한 역량을 발휘하는 데 필요한 도구와 지식을 제공합니다."

Head First Git의 저자이자 Head First Software Architecture의 공동 저자인 라주 간디

소프트웨어 아키텍처의 기초

코딩을 넘어 경력을 발전시키고자 하는 개발자들은 소프트웨어 아키텍트가 되기를 열망하지만, 지금까지 그들을 위한 실질적인 지침서는 없었습니다. 이 개정판은 소프트웨어 아키텍처의 다양한 측면을 포괄적으로 다루며, 최신 분야 동향을 반영한 새로운 장들을 다수 포함하고 있습니다. 기존 및 예비 아키텍트 모두 아키텍처 특성, 아키텍처 패턴, 컴포넌트 결정, 아키텍처 디어그램 작성, 거버넌스, 데이터, 생성형 AI, 팀 구성 등 다양한 주제를 살펴볼 수 있습니다.

실무 경험이 풍부하고 수년간 소프트웨어 아키텍처 강의를 전문적으로 진행해 온 마크 리처즈와 닐 포드는 모든 기술 스택에 적용되는 아키텍처 원칙에 중점을 둡니다. 지난 10년간의 모든 혁신을 고려하여 현대적인 관점에서 소프트웨어 아키텍처를 탐구하게 될 것입니다.

이 책은 다음 내용을 살펴봅니다.

- 아키텍처 스타일 및 패턴: 마이크로서비스, 모듈형 모놀리스, 마이크로커널, 계층형 아키텍처 등 다양한 것들이 있습니다.
- 구성 요소: 식별, 결합, 응집
분할 및 세분화
- 소프트 스킬: 효과적인 팀 관리, 협업, 비즈니스 참여 모델, 협상, 프레젠테이션 등
- 현대 엔지니어링 관행: 클라우드 환경 및 생성형 AI를 포함하여 지난 몇 년 동안 급격하게 변화한 방법 및 운영 접근 방식
- 엔지니어링 분야로서의 아키텍처: 반복 가능한 결과, 측정 지표 및 구체적인 평가를 통해 소프트웨어 아키텍처에 엄밀성을 더합니다.

마크 리처즈는 마이크로서비스 아키텍처 및 기타 분산 시스템의 아키텍처, 설계 및 구현에 참여한 경험에 풍부한 실무형 소프트웨어 아키텍트입니다. 그는 오라일리 출판사에서 다수의 기술 서적과 비디오를 집필했습니다.

닐 포드는 Thoughtworks의 이사, 소프트웨어 아키텍트이자 믿 전문가이며, 소프트웨어 개발 및 제공 분야에서 국제적으로 인정받는 전문가입니다. 그는 8권의 저서(그리고 계속해서 출간 예정), 다수의 잡지 기사, 그리고 수십 편의 비디오 프레젠테이션을 집필했습니다.

소프트웨어 아키텍처

미국 79.99달러, 캐나다 99.99달러
ISBN: 978-1-098-17551-1



O'REILLY®

소프트웨어 아키텍처 기초에 대한 찬사

마크와 닐이 또 한 번 해냈습니다. 베스트셀러의 개정 및 증보판인 이 두 번째 판은 현대적인 관점에서 소프트웨어 아키텍처를 탐구하는 데 없어서는 안 될 필수 자료입니다. 소프트웨어 아키텍처의 진정한 의미를 심도 있게 이해하는 이 종합 가이드는 트레이드오프 분석의 중요성을 강조하는 것부터 시작하여 다양한 아키텍처 스타일과 그 기반이 되는 철학, 그리고 데이터와 팀 구성에 대한 자세한 분석까지 다룹니다. 우연히 아키텍트 역할을 맡게 된 초보자든, 실력을 갈고닦고자 하는 베테랑이든, 이 책은 여러분의 전문성을 극대화하는 데 필요한 도구와 지식을 제공합니다.

—라주 간디, 『Head First Git』 저자 및 『Head First Software Architecture』 공동 저자

닐과 마크는 뛰어난 소프트웨어 아키텍트일 뿐만 아니라 탁월한 교육자이기도 합니다. 그들은 『소프트웨어 아키텍처의 기초』를 통해 방대한 아키텍처라는 주제를 수십 년간 쌓아온 경험을 바탕으로 간결하게 정리했습니다. 이제 막 아키텍트 일을 시작한 사람이든, 오랜 경력을 가진 사람이든, 이 개정판은 업무 능력을 향상시키는 데 큰 도움이 될 것입니다. 제 경력 초기에 이 책이 나왔다면 얼마나 좋았을까 하는 생각이 듭니다. 저는 이미 두 판본 모두를 제 아키텍처 대학원생들에게 교재로 사용했고, 이번에 확장된 개정판을 앞으로도 널리 추천할 것입니다.

—나다니엘 슈타, 『소프트웨어 공학의 기초』 공동 저자

마크와 닐은 소프트웨어 아키텍처 분야에서 탁월한 역량을 발휘하는 데 필요한 복잡하고 다
종적인 기본 원리를 명확히 설명하는 어려운 목표를 세웠고, 이번에도 그 목표를 완수했습니다.
다. 소프트웨어 아키텍처 분야는 끊임없이 진화하고 있으며, 이 분야의 전문가에게는 방대한 지식
과 깊이 있는 기술이 요구됩니다. 이번 개정판은 소프트웨어 아키텍처 전문가가 되기 위한 여정에
서 많은 이들에게 길잡이가 되어 줄 것입니다.

—레베카 J. 파슨스, 기술 고문 및 전 오토워
스 CTO/명예 CTO

마크와 닐은 기술자들이 아키텍처의 우수성을 달성할 수 있도록 실질적인 조언을 제공합니다.
그들은 성공을 이끌어내는 데 필요한 일반적인 아키텍처 특성과 절충점을 파악함으로써 이
를 달성합니다.

—캐시 슘, 오토워스 기술 이사

제2판

소프트웨어의 기초 건축학

현대 공학적 접근 방식

마크 리처즈와 닐 포드

O'REILLY®

마크 리처즈와 널 포드의 소프트웨어 아키텍처 기초

저작권 © 2025 Mark Richards 및 Neal Ford. 모든 권리 보유.

미국에서 인쇄되었습니다.

오라일리 미디어(O'Reilly Media, Inc.), 1005 Gravenstein Highway North, Sebastopol, CA 95472에서 발행.

오라일리 출판사의 도서는 교육, 비즈니스 또는 판촉 목적으로 구매하실 수 있습니다. 대부분의 도서는 온라인판 (<http://oreilly.com>)으로도 제공됩니다. 더 자세한 정보는 기업/기관 판매 부서(800-998-9938 또는 corporate@oreilly.com)로 문의해 주십시오.

구매 담당 편집자: 루이스 코리건
개발 담당 편집자: 사라 그레이
프로덕션 편집자: 크리스토퍼 포서
교정 담당: 소니아 사루바
교정: Piper Content Partners

색인 생성업체: WordCo Indexing Services, Inc.
내지 디자인: 데이비드 푸타토 표지 디자인: 캐린 몽고메리 삽화: 케이트 둘리아

2020년 1월: 초판
2025년 3월: 제2판

제2판 개정 내역

2025년 3월 12일: 첫 출시

자세한 릴리스 정보는 <http://oreilly.com/catalog/errata.csp?isbn=9781098175511>을 참조하십시오.

오라일리 로고는 오라일리 미디어(O'Reilly Media, Inc.)의 등록 상표입니다. 소프트웨어 아키텍처의 기초, 표지 이미지 및 관련 상표 디자인은 오라일리 미디어(O'Reilly Media, Inc.)의 상표입니다.

본 저서에 표현된 견해는 저자들의 견해이며 출판사의 견해를 대변하는 것은 아닙니다. 본 출판사와 저자는 본 저작물에 포함된 정보와 지침의 정확성을 보장하기 위해 성실한 노력을 기울였지만, 오류나 누락에 대한 모든 책임을 부인하며, 특히 본 저작물의 사용 또는 의존으로 인해 발생하는 손해에 대한 책임도 부인합니다. 본 저작물에 포함된 정보와 지침의 사용은 전적으로 사용자 자신의 책임입니다. 본 저작물에 포함되거나 설명된 코드 샘플 또는 기타 기술이 오픈 소스 라이선스 또는 타인의 저작 재산권의 적용을 받는 경우, 사용자는 해당 라이선스 및/또는 권리를 준수하여 사용해야 할 책임이 있습니다.

978-1-098-17551-1

[LSI]

목차

머리말.....	xvii
1. 서론.	1
소프트웨어 아키텍처 정의	2
소프트웨어 아키텍처의 법칙	6
건축가의 기대	8
아키텍처 관련 결정을 내리세요	8
아키텍처를 지속적으로 분석하세요	9
최신 트렌드를 놓치지 마세요	9
결정 사항 준수 보장	10
다양한 기술을 이해하세요	10
비즈니스 영역을 파악하세요	11
대인관계 능력을 갖추고 있습니다	11
정치를 이해하고 해쳐나가기	12
로드맵	13
제1부. 기초	
2. 건축적 사고.	17
건축과 디자인의 차이점	17
전략적 결정과 전술적 결정	18
노력 수준	19
절충의 중요성	19
기술적 폭	20

20분 규칙	24
개인 레이더 개발하기	25
절충점 분석	30
비즈니스 동인 이해하기	33
아키텍처 설계와 실무 코딩의 균형 유지	33
건축적 사고에는 그 이상의 것이 있다	36
3. 모듈성. ...	
모듈성 대 세분화	38
모듈성 정의하기	38
모듈성 측정	40
응집력	41
연결	44
핵심 지표	45
주계열과의 거리	46
결합	49
모듈에서 컴포넌트로	54
4. 건축적 특징 정의.....	55
건축적 특징 및 시스템 설계	56
건축적 특징 (일부) 목록	59
운영 아키텍처 특성	59
구조적 건축 특성	60
클라우드의 특징	60
건축적 특징의 교차점	61
절충과 차선책 아키텍처	64
5. 건축적 특징 파악. ...	
도메인 관심사로부터 아키텍처적 특징 추출하기	67
복합적인 건축적 특징	68
건축적 특징 추출	69
Katas와 함께 일하기	70
카타: 실리콘 샌드위치	71
명시적 특징	72
암묵적 특성	75
건축적 특징의 제한 및 우선순위 설정	77
6. 건축 특성의 측정 및 관리.....	81
건축물의 특성 측정	81

운영 조치	82
구조적 조치	83
프로세스 측정	86
관리 및 적합성 기능	86
지배적인 아키텍처 특성	86
피트니스 기능	87
7. 건축적 특징의 범위.....	95
아키텍처 양자 및 세분성 동기식 통신 범위 설정의 영향	96
범위 설정과 아키텍처 스타일 카타: 친환경	99
으로 나아가기 범위 설정과 클라	100
우드	102
	103
	105
8. 구성 요소 기반 사고.....	107
논리적 구성 요소 정의	107
논리적 아키텍처와 물리적 아키텍처의 차이점	110
논리적 아키텍처 구축	112
핵심 구성 요소 식별	113
사용자 스토리를 구성 요소에 할당하기	117
역할과 책임 분석	118
건축적 특징 분석	120
구조 조정 구성 요소	120
컴포넌트 커플링	121
정적 커플링	121
시간적 결합	122
데메테르의 법칙	123
사례 연구: 떠나고, 떠나고, 사라지다—구성 요소 발견	125

제2부. 건축 양식

9. 기초.	
스타일과 패턴	131
기본 패턴	133
커다란 진흙 덩어리	133
단일 아키텍처	134
클라이언트/서버	135

아키텍처 분할	137
Kata: 실리콘 샌드위치 - 파티셔닝	140
단일형 아키텍처와 분산형 아키텍처 비교	143
오류 1: 네트워크는 신뢰할 수 있다	144
두 번째 오류: 자연 시간은 0이다	144
세 번째 오류: 대역폭은 무한하다	145
오류 4: 네트워크는 안전하다	146
오류 5: 위상은 절대 변하지 않는다	147
오류 6: 관리자는 단 한 명뿐이다	148
오류 7: 운송비는 0이다	148
오류 8: 네트워크는 동질적이다	149
다른 오류들	149
팀 토플로지 및 아키텍처	151
이제 구체적인 스타일로 넘어가겠습니다.	152
10. 겹겹이 쌓은 건축 양식	153
위상수학	153
스타일 세부 사항	155
격리의 층	155
레이어 추가	157
데이터 토플로지	159
클라우드 고려 사항	159
일반적인 위험	159
통치	159
팀 토플로지 고려 사항	160
스타일 특징	161
사용 시점	162
사용하지 말아야 할 경우	163
예시 및 사용 사례	163
11. 모듈형 모놀리스 건축 양식	165
위상수학	165
스타일 세부 사항	166
일체형 구조	167
모듈형 구조	168
모듈 통신	168
데이터 토플로지	170
클라우드 고려 사항	171
일반적인 위험	171

통치	172
팀 토플로지 고려 사항	174
스타일 특징	175
사용 시점	176
사용하지 말아야 할 경우	177
예시 및 사용 사례	177
 12. 파이프라인 아키텍처 스타일.....	181
위상수학	181
스타일 세부 사항	182
필터	182
파이프	183
데이터 토플로지	184
클라우드 고려 사항	184
일반적인 위험	185
통치	186
팀 토플로지 고려 사항	188
스타일 특징	189
사용 시점	190
사용하지 말아야 할 경우	190
예시 및 사용 사례	191
 13. 마이크로커널 아키텍처 스타일.....	193
위상수학	193
스타일 세부 사항	194
코어 시스템	194
플러그인 구성 요소	198
"마이크로커넌싱"의 스펙트럼	200
기재	201
계약	201
데이터 토플로지	202
클라우드 고려 사항	203
일반적인 위험	203
변동성 핵심	203
플러그인 종속성	204
통치	204
팀 토플로지 고려 사항	204
건축적 특징 평가	205
예시 및 사용 사례	207

14. 서비스 기반 아키텍처 스타일.....	209
위상수학	209
스타일 세부 사항	211
서비스 설계 및 세분화	213
사용자 인터페이스 옵션	213
API 게이트웨이 옵션	215
데이터 토플로지	215
클라우드 고려 사항	219
일반적인 위험	219
통치	219
팀 토플로지 고려 사항	220
스타일 특징	221
예시 및 사용 사례	224
15. 이벤트 기반 아키텍처 스타일.	227
위상수학	228
스타일 세부 사항	232
이벤트와 메시지의 차이점	232
파생 이벤트	233
확장 가능한 이벤트 트리거링	235
비동기 기능	235
방송 기능	239
이벤트 페이로드	240
모기떼 안티패턴	246
오류 처리	249
데이터 손실 방지	253
요청-응답 처리	256
매개된 이벤트 기반 아키텍처	258
데이터 토플로지	268
단일체 데이터베이스 토플로지	269
도메인 데이터베이스 토플로지	271
전용 데이터 토플로지	273
클라우드 고려 사항	275
일반적인 위험	275
통치	276
팀 토플로지 고려 사항	276
스타일 특징	277
요청 기반 모델과 이벤트 기반 모델 중 선택하기	280
예시 및 사용 사례	280

16. 공간 기반 건축 양식.....	283
위상수학	284
스타일 세부 사항	286
처리 장치	286
가상화된 미들웨어	287
메시징 그리드	287
데이터 그리드	288
처리 그리드	296
배포 관리자	297
데이터 펌프	297
데이터 작성자	298
데이터 판독기	300
데이터 토플로지	302
클라우드 고려 사항	302
일반적인 위험	303
데이터베이스에서 자주 읽는 항목	303
데이터 동기화 및 일관성	304
대용량 데이터	304
데이터 충돌	304
통치	307
팀 토플로지 고려 사항	310
스타일 특징	311
예시 및 사용 사례	313
콘서트 티켓 예매 시스템	313
온라인 경매 시스템	313
17. 오피스트레이션 기반 서비스 지향 아키텍처.....	315
토플로지 스	315
타일 세부 사항 분	316
류 체계 재사	317
용… 및 데이터 토플로지 결합 클	320
라우드 고려 사항 일반	322
적인 위험 관리 팀 토플로지 고	323
려 사항 스타일 특징	323
예시 및 사용 사	324
례	325
	325
	327

18. 마이크로서비스 아키텍처. ...	
위상수학	330
스타일 세부 사항	331
경계 컨텍스트	331
세분성	332
데이터 격리	333
API 레이어	334
운영 재사용	334
프런트엔드	338
의사소통	339
안무 및 오케스트레이션	341
거래와 이야기	345
데이터 토플로지	348
클라우드 고려 사항	351
일반적인 위험	351
통치	352
팀 토플로지 고려 사항	353
스타일 특징	354
예시 및 사용 사례	356
19. 적절한 건축 양식 선택.....	359
건축의 변화하는 "패션"	359
결정 기준	361
모놀리스 사례 연구: 실리콘 샌드위치	364
모듈형 모놀리스	365
마이크로커널	366
분산형 사례 연구: 가고, 가고, 사라지다	367
20. 건축 패턴. ...	
재사용	372
도메인과 운영 결합 분리	372
의사소통	375
오케스트라 편곡 vs. 안무	375
CQRS	378
하부 구조	379
브로커-도메인 패턴	380

제3부. 기법 및 소프트 스킬

21. 건축적 결정. ...	
아키텍처 결정 안티패턴	387
자산 보호 안티패턴	387
그라운드호그 데이 안티패턴	389
이메일 기반 아키텍처 안티패턴	389
건축학적 중요성	390
건축 결정 기록	391
기본 구조	392
예	398
ADR 저장	400
ADR을 문서화하는 방법	401
표준에 ADR을 활용하기	402
기존 시스템에서 ADR 사용	402
건축 설계 결정에 생성형 AI와 LLM을 활용하기	402
22. 아키텍처 위험 분석. ...	
위험 매트릭스	405
위험 평가	406
위험 폭풍	410
1단계: 식별	411
2단계: 합의 도출	412
3단계: 위험 완화	415
사용자 스토리 위험 분석	416
위험 분석 활용 사례	416
유효성	418
탄력	420
보안	421
요약	422
23. 아키텍처 다이어그램 작성. ...	
도표 작성	424
도구	424
다이어그램 작성 표준: UML, C4 및 ArchiMate	426
도표 작성 지침	428
요약	431

24. 효과적인 팀 만들기. ...	
협업 제약 조건 및 경계	433
계, 건축가의 성격 유형, 통제권 건축가, 방	435
구석 건축가, 유능한 건축가, 참여	436
수준은 어느 정도가 적당할까?	437
팀 경고 신호	437
공정 손실	438
다원적 무지	439
체크리스트 활용하기	443
개발자 코드 자동 완성 체크리스트	444
단위 및 기능 테스트 체크리스트	445
소프트웨어 릴리스 체크리스트	446
안내 제공	447
요약	448
25. 협상 및 리더십 기술.....	453
협상 및 중재	453
비즈니스 이해관계자와의 협상	453
다른 건축가들과 협상하기	456
개발자와의 협상	457
소프트웨어 아키텍트의 리더십	459
건축의 4C	459
실용적이면서도 미래지향적이어야 한다.	461
솔선수범하여 팀을 이끌다	462
개발팀과의 통합	465
요약	468
26. 건축적 교차점. ...	
아키텍처 및 구현 운영상의 고려 사항 구조적 무	470
결성 아키텍처 제약 조건 아키텍	471
처 및 인프라 아키텍처 및 데	472
이터 토플로지 데이터베이스 토플로	474
지 아키텍처 특성	475
	477
	478
	479

데이터 구조	479
읽기/쓰기 우선순위	479
건축 및 엔지니어링 실무	480
아키텍처 및 팀 토플로지	481
아키텍처 및 시스템 통합	482
건축과 기업	482
건축과 비즈니스 환경	483
건축과 생성형 인공지능	484
건축에 생성형 인공지능 통합	484
건축가 보조 도구로서의 생성형 AI	484
요약	485
27. 소프트웨어 아키텍처의 법칙 재고찰.....	487
제1법칙: 소프트웨어 아키텍처의 모든 것은 절충의 산물이다	487
공유 라이브러리 대 공유 서비스	488
동기식 메시징과 비동기식 메시징 비교	490
첫 번째 결론: 누락된 절충안	492
두 번째 결론: 한 번만으로는 충분하지 않다	494
제2법칙: '어떻게'보다 '왜'가 더 중요하다	494
맥락에서 벗어난 안티패턴	494
극단 사이의 스펙트럼	495
마지막으로 드리는 조언	496
부록. 토론 질문.	
색인.	

머리말

제2판 서문

와, 정말 많은 게 있네요!

소프트웨어 아키텍처 기초 2판을 집필하기 시작했을 때, 1판에서 보완하고 개선하고 싶은 몇 가지 아이디어가 있었지만, 많은 소프트웨어 프로젝트가 그렇듯 계속해서 내용이 늘어났습니다.

우리가 달성한 목표 중 하나는 스타일 섹션을 더욱 일관성 있게 만들어 비교에 더 유용하게 만드는 것이었습니다. 또한 별점 평가 시스템을 개편하여 섹션과 몇 가지 새로운 카테고리를 추가했고, 각 아키텍처 스타일별로 클라우드 고려 사항, 데이터 토플로지, 팀 토플로지 및 거버넌스에 대한 새로운 섹션을 추가했습니다. 그 과정에서 15장 **과** 18 장과 같은 인기 주제에 대한 여러장을 대폭 수정했고, 모듈형 모놀리식 아키텍처 스타일에 대한 새로운 장 (**11장**)을 추가했습니다.

또한 **20장**에서는 아키텍처 패턴을, **26장**에서는 아키텍처의 교차점을, 그리고 **27장**에서는 소프트웨어 아키텍처 법칙(새로운 부록과 새로운 법칙이 포함됨)을 다시 살펴보는 등 완전히 새로운 장들을 추가했습니다.

초판 서문

공리란 확

립되었거나, 받아들여졌거나, 자명하게 참이라고 여겨지는 진술 또는 명제입니다.

수학자들은 공리, 즉 논쟁의 여지가 없는 참이라는 가정에 기반하여 이론을 만듭니다. 소프트웨어 설계자들도 공리 위에 이론을 구축하지만, 소프트웨어 세계는 수학보다 훨씬 더 빠르게 변화합니다. 우리가 이론의 기반으로 삼는 공리를 포함하여 근본적인 것들이 끊임없이 급변하기 때문입니다.

소프트웨어 개발 생태계는 끊임없이 역동적인 평형 상태에 놓여 있습니다. 특정 시점에는 균형 상태를 유지하지만, 장기적으로는 역동적인 변화를 보입니다. 이러한 생태계의 특성을 잘 보여주는 현대적인 예로 컨테이너 기술의 발전과 그에 따른 변화를 들 수 있습니다. 10년 전만 해도 **쿠버네티스** 같은 도구는 존재하지 않았지만, 이제는 쿠버네티스 사용자를 위한 소프트웨어 컨퍼런스가 생겨날 정도입니다. 소프트웨어 생태계는 혼란스럽게 변화합니다. 작은 변화 하나가 또 다른 작은 변화를 일으키고, 이러한 과정이 수백 번 반복되면서 새로운 생태계가 만들어집니다.

아키텍트는 이전 시대에서 남겨진 가정과 공리에 의문을 제기해야 할 중요한 책임이 있습니다. 소프트웨어 아키텍처에 관한 많은 책들은 현재 세계와는 거의 유사점이 없는 시대에 쓰였습니다. 실제로 저자들은 향상된 엔지니어링 관행, 운영 생태계, 소프트웨어 개발 프로세스 등 아키텍트와 개발자가 매일 작업하는 복잡하고 역동적인 균형 상태를 구성하는 모든 요소를 고려하여 근본적인 공리에 대해 정기적으로 질문해야 한다고 주장합니다.

소프트웨어 아키텍처를 주의 깊게 관찰해 온 사람들은 시간이 흐르면서 기능이 진화하는 것을 목격했습니다. **익스트림 프로그래밍(EP)**과 같은 엔지니어링 기법에서 시작하여 지속적 배포(CD), DevOps 혁명, 마이크로서비스, 컨테이너화, 그리고 현재의 클라우드 기반 리소스에 이르기까지, 이러한 모든 혁신은 새로운 기능과 그에 따른 장단점을 가져왔습니다. 기능이 변화함에 따라 업계에 대한 아키텍트의 관점도 변화했습니다. 오랫동안 소프트웨어 아키텍처는 "나중에 변경하기 어려운 것들"이라는 농담조의 정의로 사용되었습니다. 이후, 변경을 최우선 설계 고려 사항으로 삼는 마이크로서비스 아키텍처 스타일이 등장했습니다.

새로운 시대가 도래할 때마다 새로운 관행, 도구, 측정 방식, 패턴 등 다양한 변화가 요구됩니다. 이 책은 지난 10년간의 혁신을 반영하고 오늘날의 새로운 구조와 관점에 적합한 새로운 측정 기준과 지표를 제시하며 소프트웨어 아키텍처를 현대적인 관점에서 살펴봅니다.

저희 책의 부제는 "현대적인 엔지니어링 접근법"입니다. 개발자들은 오랫동안 소프트웨어 개발을 숙련된 장인이 일회성 작품을 만들어내는 단순한 작업에서 반복성, 엄격함, 효과적인 분석을 수반하는 엔지니어링 분야로 바꾸고 싶어 했습니다. 소프트웨어 엔지니어링은 다른 엔지니어링 분야에 비해 아직 여러 면에서 뒤쳐져 있지만(솔직히 말하면 소프트웨어는 다른 엔지니어링 분야에 비해 매우 젊은 분야입니다), 아키텍트들은 상당한 발전을 이루어냈으며, 이에 대해 논의할 것입니다. 특히, 현대적인 애자일 엔지니어링 방식은 아키텍트가 설계하는 시스템의 유형에 큰 발전을 가져왔습니다.

또한 우리는 매우 중요한 문제인 트레이드오프 분석에 대해서도 다룹니다. 소프트웨어 개발자라면 특정 기술이나 접근 방식에 매료되기 쉽습니다.

하지만 건축가는 모든 선택의 장단점을 냉철하게 평가해야 하며, 현실 세계에서는 편리한 이분법적 선택이 가능한 것은 거의 없습니다. 모든 것은 장단점이 존재합니다. 이러한 실용적인 관점에서 우리는 가치 판단을 배제하기 위해 노력합니다.

기술 자체에 대한 이야기보다는 기술 선택에 있어 분석적인 시각을 갖도록 독자들에게 장단점을 분석하는데 집중하고자 합니다.

이 책을 읽는다고 해서 하룻밤 사이에 소프트웨어 아키텍트가 되는 것은 아닙니다. 소프트웨어 아키텍처는 다양한 측면을 가진 복잡한 분야이기 때문입니다. 이 책은 기존 및 예비 아키텍트들에게 구조부터 소프트 스킬에 이르기까지 소프트웨어 아키텍처의 다양한 측면에 대한 최신 개요를 제공하고자 합니다. 잘 알려진 패턴을 다루면서도, 우리는 경험에서 얻은 교훈, 도구, 엔지니어링 관행 및 기타 정보를 바탕으로 새로운 접근 방식을 취합니다. 소프트웨어 아키텍처의 기존 공리들을 현재의 생태계에 비추어 재해석하고, 현대적인 환경을 고려한 아키텍처 설계를 제시합니다.

본 책에서 사용된 표기법

이 책에서는 다음과 같은 인쇄 규칙을 사용합니다.

기울임

체로 표시된 부분은 새로운 용어, URL, 이메일 주소, 파일 이름 및 파일 확장자를 나타냅니다.

고정 너비는 프로그램 목록

뿐만 아니라 단락 내에서 변수 또는 함수 이름, 데이터베이스, 데이터 유형, 환경 변수, 명령문 및 키워드와 같은 프로그램 요소를 참조할 때 사용됩니다.



이 요소는 조언이나 제안을 의미합니다.



이 요소는 일반적인 메모를 나타냅니다.



이 요소는 경고 또는 주의를 나타냅니다.

보충 자료

이 책의 추가 자료를 보려면 <http://fundamentalsofwarearchitecture.com>을 방문하십시오.

기술적인 질문이나 코드 예제 사용에 문제가 있는 경우 bookquestions@oreilly.com으로 이메일을 보내주세요.

이 책은 여러분의 업무 수행을 돋기 위해 만들어졌습니다. 일반적으로 이 책에 예제 코드가 제공되는 경우, 여러분은 해당 코드를 프로그램 및 문서에 자유롭게 사용할 수 있습니다. 코드의 상당 부분을 복제하는 경우가 아니라면, 저희에게 허가를 요청할 필요는 없습니다. 예를 들어, 이 책의 코드 일부를 사용하는 프로그램을 작성하는 것은 허가가 필요하지 않습니다. 오라일리 출판사의 책에 있는 예제를 판매하거나 배포하는 경우에는 허가가 필요합니다. 이 책을 인용하고 예제 코드를 제시하여 질문에 답변하는 것은 허가가 필요하지 않습니다. 하지만 이 책의 예제 코드를 상당량 제품 문서에 포함시키는 경우에는 허가가 필요합니다.

출처 표기를 해주시면 감사하지만, 일반적으로 필수 사항은 아닙니다. 출처 표기에는 보통 책 제목, 저자, 출판사, ISBN이 포함됩니다. 예를 들어, “Fundamentals of Software Architecture, Second Edition, by Mark Richards and Neal Ford (O'Reilly). Copyright 2025 Mark Richards and Neal Ford, 978-1-098-17551-1.” 와 같이 표기해 주시면 됩니다.

만약 코드 예제 사용이 공정 사용 원칙이나 위에 명시된 허가 범위를 벗어난다고 생각되시면 permissions@oreilly.com으로 언제든지 문의해 주세요.

오라일리 온라인 학습

O'REILLY® 오라일리 미디어는 40년 이상 기업의 성공을 돋기 위해 기술 및 비즈니스 교육, 지식, 통찰력을 제공해 왔습니다.

오라일리의 독보적인 전문가 및 혁신가 네트워크는 책, 기사, 그리고 온라인 학습 플랫폼을 통해 지식과 전문성을 공유합니다. 오라일리의 온라인 학습 플랫폼은 실시간 교육 과정, 심층 학습 경로, 대화형 코딩 환경, 그리고 오라일리를 비롯한 200개 이상의 출판사에서 제공하는 방대한 텍스트 및 비디오 컨텐츠를 언제든지 이용할 수 있도록 지원합니다. 더 자세한 정보는 <http://oreilly.com>을 방문하세요.

저희에게 연락하는 방법

이 책에 대한 의견이나 질문은 출판사로 보내주시기 바랍니다.

오라일리 미디어 주식회사
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-889-8969 (미국 또는 캐나다 내) 707-827-7019 (국제 또는 지역) 707-829-0104 (팩스)
support@oreilly.com
<https://oreilly.com/about/contact.html>

이 책에 대한 웹페이지가 있습니다. 해당 페이지에는 오류 정정, 예제 및 추가 정보가 나와 있습니다. <https://oreil.ly/fundamentals-of-software-architecture-2e>에서 해당 페이지에 접속하실 수 있습니다.

저희 도서 및 강좌에 대한 소식과 정보를 보시려면 <https://oreilly.com>을 방문하세요.

LinkedIn에서 저희를 찾아보세요: <https://linkedin.com/company/oreilly-media>.

저희 유튜브 채널을 시청하세요: <https://youtube.com/oreillymedia>.

감사의 말씀

마크와 닐은 저희 강의, 워크숍, 컨퍼런스 세션, 사용자 그룹 회의에 참석해 주신 모든 분들, 그리고 이 자료의 여러 버전을 듣고 귀중한 피드백을 주신 모든 분들께 감사의 말씀을 전합니다. 또한, 이 책을 집필하는 과정을 최대한 수월하게 만들어 주신 오라일리 출판팀에도 감사드립니다. 특히 초판 편집을 맡아주신 알리시아 영과 베지니아 윌슨, 그리고 2판 편집을 맡아주신 사라 그레이에 게 깊은 감사를 드립니다.

마크 리처즈의 감사 인사

앞서 언급한 감사 인사 외에도 사랑하는 아내 레베카에게 감사를 전하고 싶습니다. 집안일을 도맡아 하고 자신의 책을 쓸 기회를 희생해 준 덕분에 저는 컨설팅 업무를 더 많이 하고, 더 많은 컨퍼런스와 교육 과정에서 강연을 할 수 있었으며, 이 책의 내용을 연습하고 다듬을 수 있었습니다. 당신은 최고입니다.

닐 포드의 감사 인사

이 책을 집필하는 데 도움을 주신 모든 분들께 감사드립니다. 특히 저의 소중한 가족, Thoughtworks라는 회사 전체, 그리고 그 구성원인 레베카 파슨스와 마틴 파울러에게 깊은 감사를 전합니다. Thoughtworks는 고객에게 가치를 창출하는 동시에, 모든 것이 잘 되는 이유를 끊임없이 탐구하고 개선해 나가는 특별한 사람들로 이루어진 집단입니다. Thoughtworks는 이 책을 여러분으로 지원해 주었고, 매일 새로운 도전과 영감을 주는 Thoughtworkers를 계속해서 성장시켜 나가고 있습니다. 또한, 일상에서 벗어나 잠시 숨을 돌릴 수 있는 소중한 시간을 제공해 준 동네 칵테일 클럽에도 감사드립니다. 마지막으로, 책 집필과 학회 발표 같은 일들을 묵묵히 참아준 제 아내 캔디에게 진심으로 감사드립니다. 수십 년 동안 그녀는 제가 제정신을 유지하고 제 삶을 살아갈 수 있도록 든든한 버팀목이 되어 주었고, 앞으로도 오랫동안 제 삶의 사랑으로 함께해 주기를 바랍니다.

제1장

소개

소프트웨어 아키텍처에 관심이 있으시군요. 아마도 다음 단계로 도약하고 싶은 개발자이거나, 소프트웨어 아키텍처가 제대로 작동할 때 어떤 일이 일어나는지 이해하고 싶은 프로젝트 관리자일 수도 있습니다. 또는 아직 "소프트웨어 아키텍트"라는 직함은 없지만 아키텍처 관련 결정(아래 정의 참조)을 내리는 "우연한 아키텍트"일 수도 있습니다.

소프트웨어 아키텍처 분야에 뛰어들어야 하는 이유는 무엇일까요? 아마도 다양한 프로젝트 경험을 쌓았고, 시스템의 주요 구성 요소들이 어떻게 서로 연결되는지, 그리고 그 과정에서 발생하는 수많은 절충점을 더 깊이 이해하고 싶기 때문일 것입니다. 그렇다면 소프트웨어 아키텍처는 탁월한 다음 커리어 단계가 될 수 있습니다.

이 책은 여러분 모두를 위해 쓰였습니다. 이 책은 매우 다면적인 직무인 "소프트웨어 아키텍트"에 대한 개괄적인 내용을 제공합니다.

소프트웨어 아키텍트는 소프트웨어 시스템의 복잡성을 깊이 있게 이해하고 분석해야 하며, 때로는 불완전한 정보 속에서도 중요한 절충안을 결정해야 합니다. 생성형 AI가 서서히 자신들을 대체할지도 모른다는 우려 때문에 많은 소프트웨어 개발자들이 대체가 훨씬 어려운 소프트웨어 아키텍트로의 전직을 고려하고 있습니다. 소프트웨어 아키텍트는 복잡하고 끊임없이 변화하는 환경 속에서 절충안을 평가하며, AI가 할 수 없는 종류의 결정을 내립니다.

건축은 다른 많은 예술과 마찬가지로 맥락 속에서만 이해될 수 있습니다. 건축가들은 주변 환경의 현실을 바탕으로 결정을 내립니다. 예를 들어, 20세기 후반 소프트웨어 아키텍처의 주요 목표 중 하나는 공유 인프라와 자원을 최대한 효율적으로 사용하는 것이었습니다. 당시 운영 체제, 애플리케이션 서버, 데이터베이스 서버 등은 모두 상용 제품이었고 매우 비쌌기 때문입니다.

2002년에 마이크로서비스와 같은 아키텍처를 구축하려 했다면 상상할 수 없을 정도로 비용이 많이 들었을 겁니다. 2002년 데이터센터에 들어가 운영 책임자에게 "이봐요, 제가 혁신적인 아키텍처에 대한 멋진 아이디어가 하나 있는데,

각 서비스는 자체 전용 데이터베이스를 갖춘 독립적인 시스템에서 실행됩니다.

"윈도우 라이선스 50개, 애플리케이션 서버 라이선스 30개, 그리고 데이터베이스 서버 라이선스 최소 50개가 필요하겠네요." 오늘 날 이러한 아키텍처를 구축할 수 있는 것은 오픈 소스의 등장과 데브옵스 혁명으로 인한 최신 엔지니어링 방식 덕분입니다. 모든 아키텍처는 그 맥락의 산물입니다. 이 책을 읽으면서 이 점을 명심하십시오.

소프트웨어 아키텍처 정의

그렇다면 소프트웨어 아키텍처란 무엇일까요? 그림 1-1은 우리가 소프트웨어 아키텍처를 생각하는 방식을 보여줍니다. 이 정의는 네 가지 차원으로 구성됩니다. 시스템의 소프트웨어 아키텍처는 출발점인 아키텍처 스타일, 지원해야 하는 아키텍처 특성, 동작을 구현하는 논리적 구성 요소, 그리고 이 모든 것을 정당화하는 아키텍처 결정으로 이루어져 있습니다. 시스템의 구조는 아키텍처를 뒷받침하는 굵은 검은색 선으로 표시됩니다. 아키텍트가 분석하는 순서대로 이러한 차원들을 간략하게 살펴보고, 다음 장에서 더 자세히 다루겠습니다.

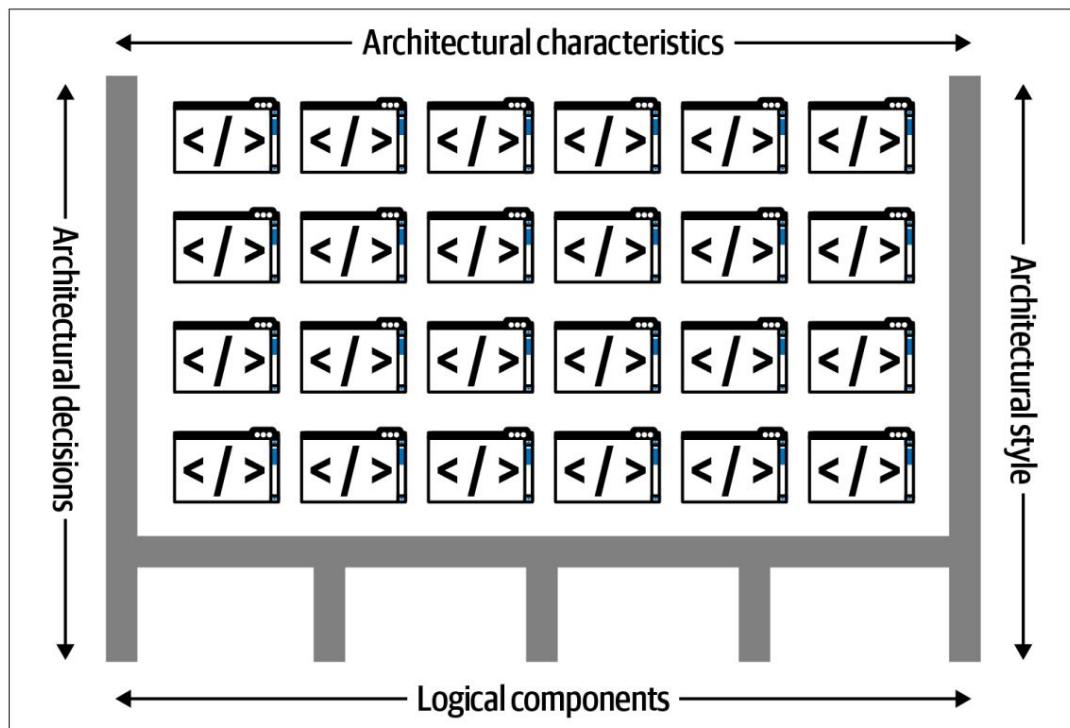


그림 1-1. 아키텍처는 시스템의 구조와 아키텍처 특성(접미사 "-ilities"), 논리적 구성 요소, 아키텍처 스타일 및 결정 사항을 결합한 것입니다.

아키텍처 특성(그림 1-2 참조)은 시스템의 기능(일반적으로 "-ilities"로 약칭됨)과 성공 기준을 정의합니다. 즉, 시스템이 무엇을 해야 하는지를 나타냅니다. 아키텍처 특성은 매우 중요하기 때문에 이 책에서는 아키텍처 특성을 이해하고 정의하는 데 여러 장을 할애했습니다.

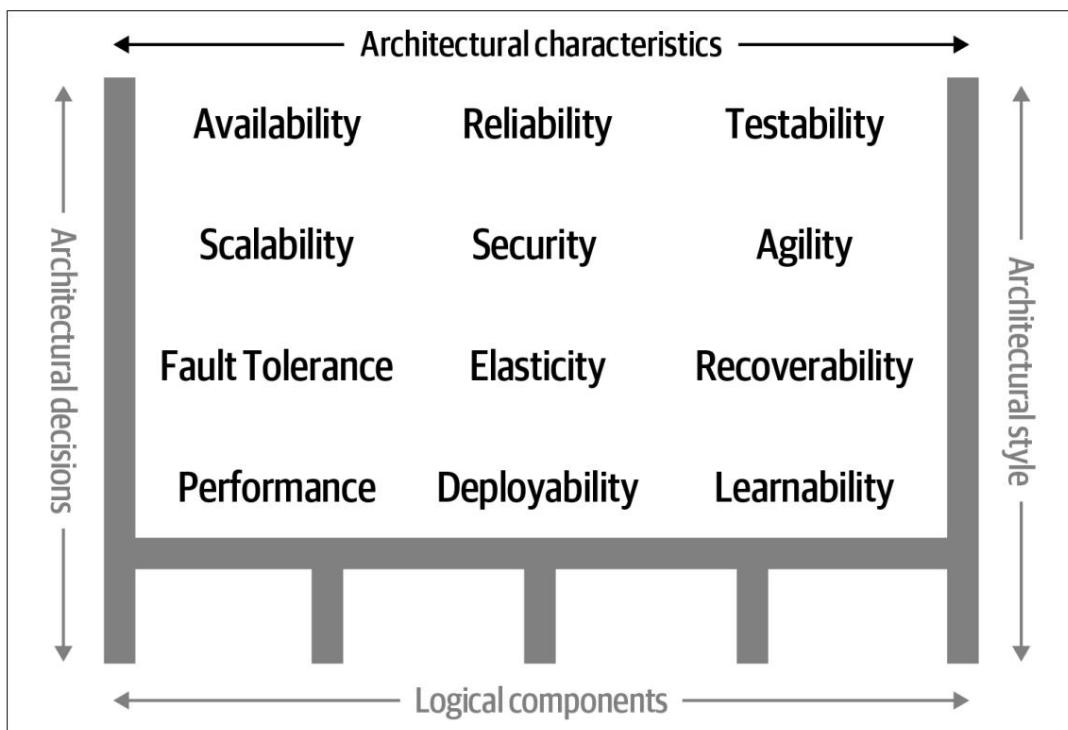


그림 1-2. "아키텍처 특성"은 시스템이 지원해야 하는 "-ilities"를 의미합니다.

아키텍처적 특성이 시스템의 기능을 정의하는 반면, 논리적 구성 요소는 시스템의 동작을 정의합니다. 논리적 구성 요소를 설계하는 것은 아키텍트에게 있어 핵심적인 구조적 활동 중 하나입니다. 그림 1-3에서 논리적 구성 요소는 애플리케이션의 도메인, 엔티티 및 워크플로를 구성합니다.

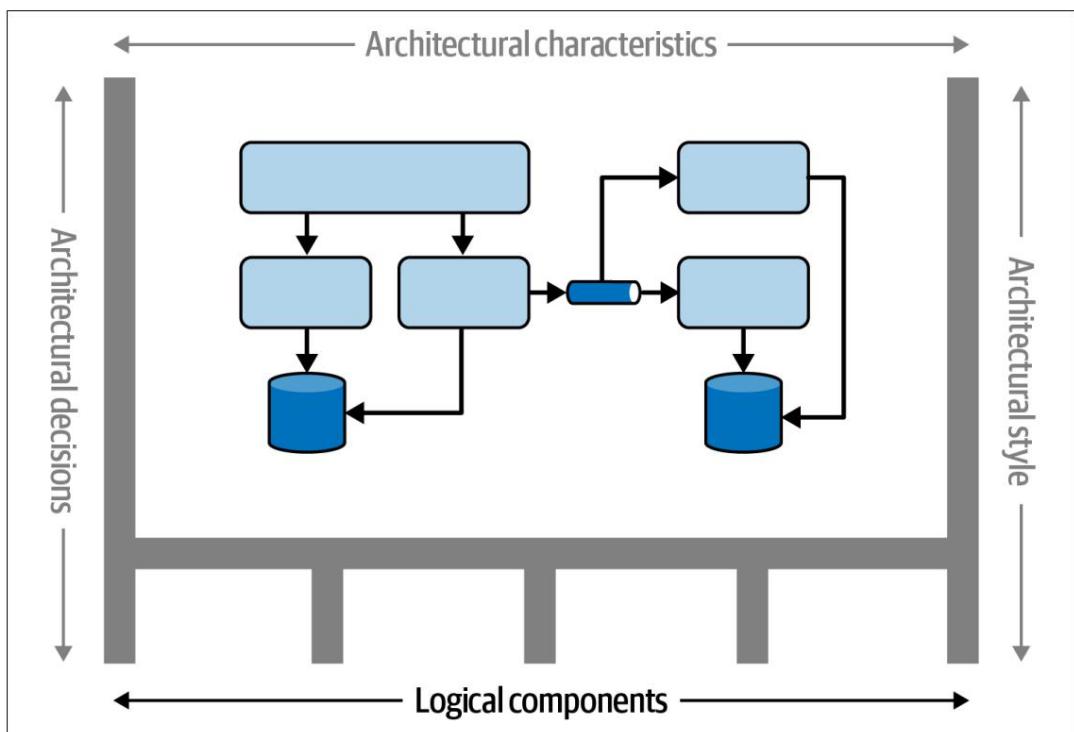


그림 1-3. 논리적 구성 요소는 시스템의 동작을 구조화합니다.

건축가는 시스템에 필요한 아키텍처적 특성과 논리적 구성 요소(둘 다 나중에 자세히 설명됨)를 분석한 후에는 솔루션 구현을 위한 출발점으로 적절한 아키텍처 스타일을 선택할 수 있을 만큼 충분한 정보를 갖게 됩니다(그림 1-4).

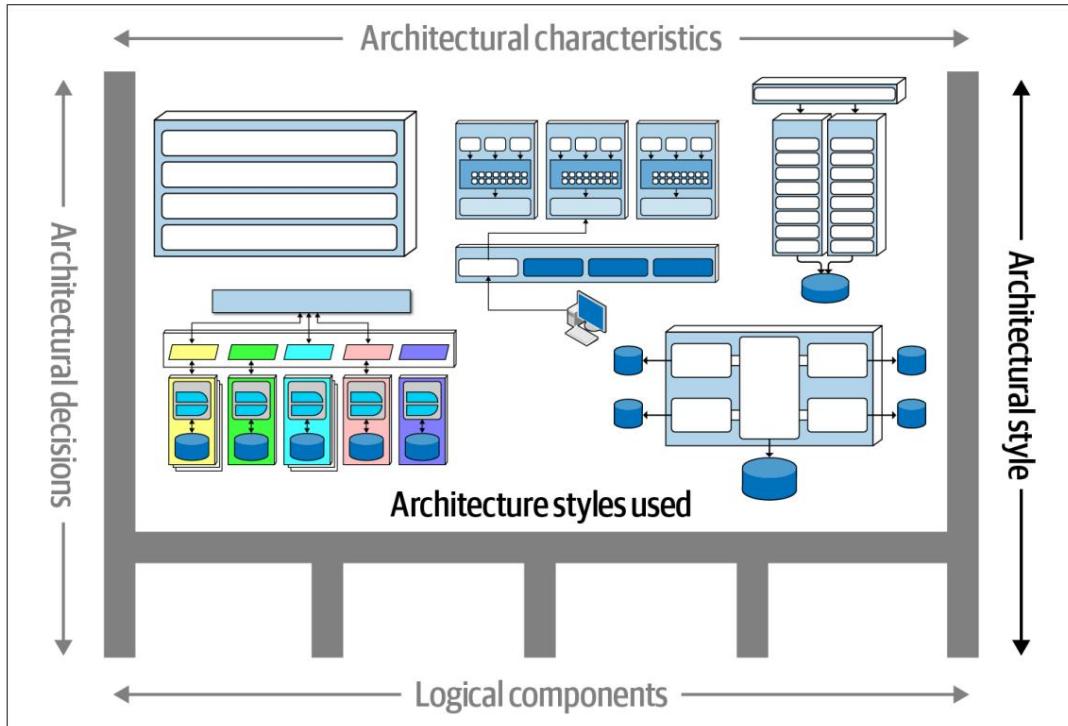


그림 1-4. 아키텍처 스타일을 선택하는 것은 주어진 요구 사항에 대해 가장 쉬운 구현 경로를 찾는 것을 의미합니다.

소프트웨어 아키텍처를 정의하는 네 번째 차원은 아키텍처 결정입니다. 이는 시스템을 어떻게 구축해야 하는지에 대한 규칙을 정의 합니다. 예를 들어, 계층형 아키텍처에서 아키텍트는 비즈니스 및 서비스 계층만 데이터베이스에 접근할 수 있도록 결정하여(그림 1-5 참조), 프레젠테이션 계층이 데이터베이스를 직접 호출하는 것을 제한할 수 있습니다. 아키텍처 결정은 시스템의 제약 조건을 형성하고 개발 팀에게 무엇이 허용되고 무엇이 허용되지 않는지를 알려줍니다.

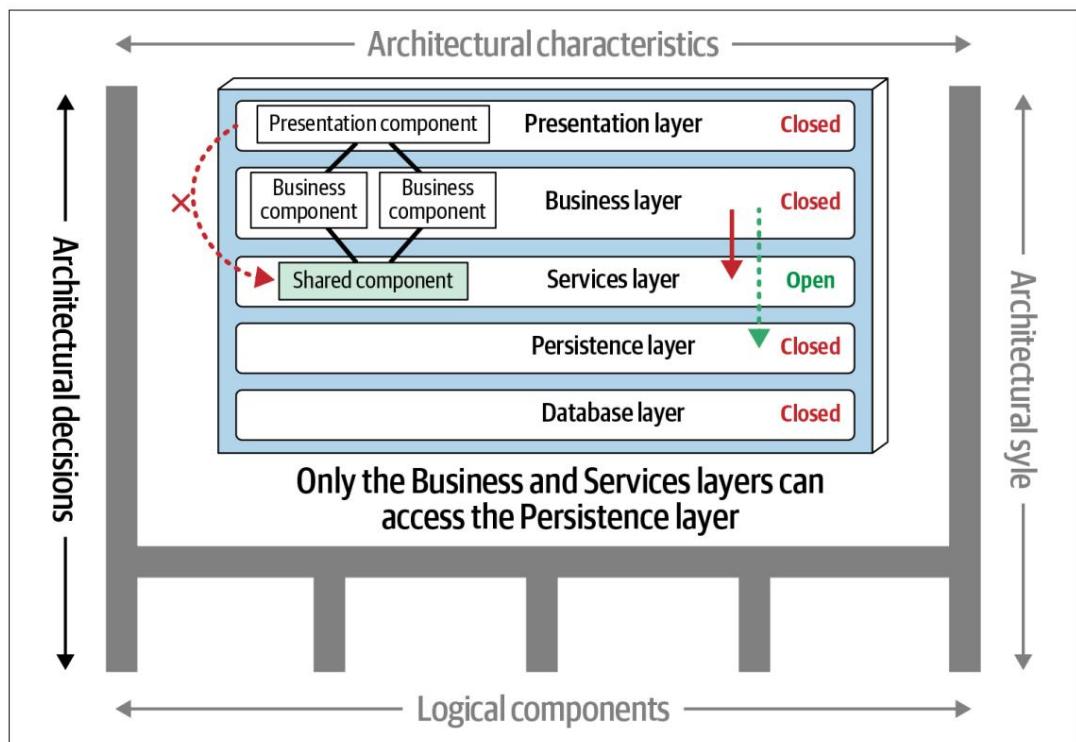


그림 1-5. 아키텍처 결정은 시스템 구축을 위한 규칙입니다.

이 장에서는 아키텍처 결정 사항과 이를 간결하게 문서화하는 방법에 대해 논의합니다 !

21번째 학기.

소프트웨어 아키텍처의 법칙

이 책의 초판을 집필하면서 두 저자는 야심찬 목표를 세웠습니다. 소프트웨어 아키텍처에 대해 보편적으로 적용되는 진리를 찾아 "소프트웨어 아키텍처 법칙"으로 정리하는 것이었습니다. 집필 과정에서 우리는 포착할 수 있는 것들을 주의 깊게 살폈고, 10개에서 15개 정도를 발견하기를 기대했습니다. 하지만 놀랍게도 초판에서는 단 두 개의 법칙만을 찾아냈고, 2판을 집필하는 과정에서 하나를 더 발견했습니다. 우리의 원래 의도대로, 이 세 가지 법칙은 매우 보편적이며 소프트웨어 아키텍트에게 중요한 여러 관점을 제시합니다.

우리는 소프트웨어 아키텍처의 첫 번째 법칙을 끊임없이 마주치면서 배우게 되었고, 이 법칙이 왜 이러한 보편적인 진리들이 그토록 파악하기 어려운지 그 핵심을 꿰뚫고 있다고 생각합니다.

소프트웨어 아키텍처의 모든 것은 절충의 문제입니다.

—소프트웨어 아키텍처의 제1법칙

세상에 깔끔하고 명확한 스펙트럼은 존재하지 않습니다. 소프트웨어 아키텍트가 내리는 모든 결정은 상황에 따라 다양한 값을 갖는 수많은 변수를 고려해야 합니다. 결국, 절충안을 찾는 것이 소프트웨어 아키텍처 결정의 핵심입니다.

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

—1차 정리

절충 분석을 한 번만 하고 끝낼 수는 없습니다.

—제2부칙

팀들은 표준을 좋아하고, 아키텍트들이 아키텍처 스타일, 아키텍처 구성 요소 간의 통신 방식, 공유 기능 관리 방법 등 여러 가지 까다로운 결정들을 한 번에 정할 수 있다면 얼마나 좋을까요? 하지만 현실은 그렇지 않습니다. 모든 상황에서 이러한 절충안들을 재평가해야 하기 때문입니다. (팀들이 실제로 그렇게 하려고 시도하는 경우를 본 적이 있습니다. 예를 들어, 분산 워크플로에서 코렉션만 기본값으로 사용하려다가 어떤 때는 잘 작동하지만 어떤 때는 엄청난 실패로 끝나는 경우입니다. 자세한 내용은 [341페이지](#)의 "코렉션과 오케스트레이션"을 참조하세요.)

건축은 단순히 구조적 요소들의 조합 그 이상이기 때문에, 우리의 차원적 정의에는 원칙, 특징 등이 포함됩니다.

이는 소프트웨어 아키텍처의 두 번째 법칙에 반영되어 있습니다.

'어떻게'보다 '왜'가 더 중요하다.

—소프트웨어 아키텍처의 두 번째 법칙

경험이 풍부한 아키텍트로서, 누군가 제가 이전에 본 적 없는 아키텍처를 보여준다면, 저는 그 작동 방식은 이해할 수 있지만, 이전 아키텍트나 팀이 특정 결정을 내린 이유를 이해하는 데는 어려움을 겪을 수 있습니다. 아키텍트는 매우 구체적인 맥락 속에서 결정을 내리기 때문에, 포괄적이고 일반적인 결정을 내리기는 어렵습니다. 아키텍트가 특정 결정을 내린 이유에는 고려했던 장단점이 포함되며, 이는 왜 그 결정이 다른 결정보다 나은지에 대한 맥락을 더해줍니다. 소프트웨어 아키텍처 결정의 핵심적인 특징 중 하나는 이분법적인 경우가 거의 없다는 것입니다. 이것이 바로 소프트웨어 아키텍처의 제3법칙으로 이어집니다.

대부분의 아키텍처 결정은 이분법적인 것이 아니라 스펙트럼 상에 존재합니다.
과격한 수단.

—소프트웨어 아키텍처의 세 번째 법칙

이 책 전반에 걸쳐 건축가가 특정 결정을 내리는 이유와 그에 따른 장단점을 강조합니다. 또한 [21장에서는 중요한 결정을 기록하는 데 유용한 기법을 소개합니다.](#)

독자들은 이 책 전체에서 이러한 법칙들이 적용되는 것을 알게 될 것이며, 소프트웨어 아키텍처 결정을 평가할 때 이 법칙들을 염두에 두는 것이 좋습니다. [27장](#) 에서 몇 가지 추가 예시와 함께 이러한 법칙들을 다시 살펴보겠습니다.

[소프트웨어 아키텍처에 대한 명확한 정의가 세워졌으니, 이제 실제 역할에 대해 살펴보겠습니다.](#)

건축가의 기대

소프트웨어 아키텍트의 역할은 전문 프로그래머 역할부터 회사 전체의 전략적 기술 방향을 정의하는 것까지 매우 다양합니다. 따라서 소프트웨어 아키텍트의 역할을 명확히 정의하는 것은 불가능에 가깝지만, 소프트웨어 아키텍트에게 기대되는 바는 분명합니다. 직책, 직함, 직무 설명과 관계없이 모든 소프트웨어 아키텍트에게 요구되는 8가지 핵심 사항을 다음과 같이 정리했습니다.

- 아키텍처 관련 결정을 내립니다
- 아키텍처를 지속적으로 분석합니다. • 최신 트렌드를 파악합니다. • 의사 결정에 대한 규정 준수를 보장합니다. • 다양한 기술, 프레임워크, 플랫폼 및 환경을 이해합니다.
- 해당 사업 영역을 이해하십시오
- 팀을 이끌고 뛰어난 대인 관계 능력을 보유해야 합니다. • 조직 내 정치적 상황을 이해하고 효과적으로 대처해야 합니다.

소프트웨어 아키텍트로 성공하려면 이러한 기대치를 이해하고 충족시키는 것이 중요합니다. 이 섹션에서는 여덟 가지 기대치를 차례로 살펴봅니다.

아키텍처 관련 결정을 내리세요

아키텍트는 팀, 부서 또는 기업 전체에 걸쳐 기술 관련 결정을 안내하는 데 사용되는 아키텍처 결정 및 설계 원칙을 정의해야 합니다.

첫 번째 기대 사항에서 핵심 단어는 '안내'입니다. 아키텍트는 기술 선택을 직접 지정하기보다는 안내하는 역할을 해야 합니다. 예를 들어, 프런트엔드 개발에 React.js를 사용하는 것은 아키텍처적 결정이라기보다는 기술적 결정입니다. 아키텍트는 직접 결정을 내리는 대신 개발팀에게 프런트엔드 웹 개발에 반응형 프레임워크를 사용하도록 지시하고, Angular, Elm, React.js, Vue 또는 기타 반응형 웹 프레임워크 중에서 선택하도록 안내해야 합니다. 핵심은 아키텍처적 결정이 팀이 올바른 기술적 선택을 하도록 안내하는 것인지, 아니면 팀 대신 결정을 내리는 것인지를 묻는 것입니다. 물론, 아키텍처의 핵심 가치를 유지하기 위해 아키텍트가 특정 기술에 대한 결정을 내려야 하는 경우도 있습니다.

확장성, 성능 또는 가용성과 같은 특정 아키텍처 특성.

건축가들은 이러한 균형을 찾는 데 어려움을 겪는 경우가 많으므로, [21장은](#) 전적으로 건축적 결정에 관한 내용입니다.

아키텍처를 지속적으로 분석하라. 건축가는 아키텍처와 현재

기술 환경을 지속적으로 분석하고 개선 방안을 제시해야 한다.

아키텍처 활력은 3년 이상 전에 정의된 아키텍처가 비즈니스 및 기술 변화를 고려했을 때 오늘날 얼마나 실행 가능한지를 평가하는 것입니다. 저희 경험상, 충분한 수의 아키텍트가 기존 아키텍처를 지속적으로 분석하는 데 에너지를 쓸지 않습니다. 그 결과, 대부분의 아키텍처는 구조적 퇴보를 경험하게 되는데, 이는 개발자가 성능, 가용성, 확장성 등 필수적인 아키텍처 특성에 영향을 미치는 코딩 또는 설계 변경을 할 때 발생합니다.

이러한 기대에서 자주 간과되는 또 다른 측면은 테스트 및 릴리스 환경입니다! 코드를 빠르게 수정할 수 있는 능력은 분명한 이점을 제공하는 일종의 애자일성이지만, 변경 사항을 테스트하는 데 몇 주가 걸리고 릴리스하는 데 몇 달이 걸린다면 전체 아키텍처는 애자일성을 달성할 수 없습니다.

건축가는 기술과 문제 영역의 변화를 전체적으로 분석하여 아키텍처의 지속적인 타당성을 판단해야 합니다. 이러한 고려 사항은 채용 공고에 거의 언급되지 않더라도 애플리케이션의 관련성을 유지하는 데 필수적입니다.

최신 트렌드를 파악하세요. 건축가는 최신 기술과 업

계 트렌드를 꾸준히 파악해야 합니다.

개발자는 최신 기술뿐 아니라 매일 사용하는 기술까지 꾸준히 습득해야만 경쟁력을 유지하고 (그리고 일자리를 지킬 수 있습니다!). 아키텍트에게는 최신 기술 및 업계 동향을 파악하는 것이 더욱 중요합니다. 아키텍트의 결정은 장기적인 영향을 미치고 변경하기 어렵기 때문입니다. 따라서 트렌드를 이해하는 것은 아키텍트가 미래에도 유효한 결정을 내리는 데 도움이 됩니다. 예를 들어, 최근 몇 년 동안 아키텍트는 클라우드 기반 스토리지 및 배포에 대해 학습해야 했으며, 이 두 번째 판을 짍힐하는 시점에서는 생성형 AI가 개발 생태계의 여러 부분에 엄청난 영향을 미치고 있습니다.

소프트웨어 아키텍트에게 있어 트렌드를 파악하고 최신 정보를 유지하는 것은 어려운 일입니다. [2 장](#) 에서는 이를 위한 몇 가지 기법과 자료들을 소개합니다.

결정 사항 준수 보장

건축가는 건축 설계 결정 및 디자인 원칙을 준수하도록 해야 할 책임이 있습니다.

규정 준수를 보장한다는 것은 개발팀이 아키텍트가 정의, 문서화 및 전달한 결정 사항과 설계 원칙을 따르고 있는지 지속적으로 확인하는 것을 의미합니다.

여러분이 아키텍트로서 계층형 아키텍처에서 데이터베이스 접근을 비즈니스 및 서비스 계층(프레젠테이션 계층 제외)으로만 제한하기로 결정했다고 가정해 보세요. ([10장에서 살펴보겠지만](#)) 이 경우 프레젠테이션 계층은 가장 간단한 데이터베이스 호출조차도 아키텍처의 모든 계층을 거쳐야 합니다.

하지만 사용자 인터페이스(UI) 개발자는 성능상의 이유로 프레젠테이션 레이어가 데이터베이스에 직접 접근할 수 있도록 하는 결정을 내렸습니다. 이러한 아키텍처 설계는 데이터베이스 변경이 프레젠테이션 레이어에 영향을 미치지 않도록 하기 위한 특정한 목적에서 이루어졌습니다. 만약 아키텍처 설계가 규정을 준수하는지 확인하지 않으면 이와 같은 문제가 발생할 수 있습니다. 그 결과, 아키텍처가 필요한 특성을 제공하지 못하게 되어 애플리케이션이나 시스템이 예상대로 작동하지 않을 수 있습니다. [6 장](#)에서는 자동화된 적합성 함수 및 기타 도구를 사용하여 규정 준수 여부를 측정하는 방법에 대해 자세히 설명합니다.

다양한 기술에 대한 이해 아키텍트는 다양하

고 폭넓은 기술, 프레임워크, 플랫폼 및 환경을 접해볼 수 있어야 합니다.

모든 아키텍트가 모든 프레임워크, 플랫폼, 언어에 능통할 필요는 없지만, 적어도 다양한 기술에 대한 기본적인 이해는 있어야 합니다. 요즘 환경은 대부분 이기종 시스템들로 구성되어 있기 때문에, 아키텍트는 최소한 언어, 플랫폼, 기술에 관계없이 여러 시스템 및 서비스와 상호 작용하는 방법을 알고 있어야 합니다.

이러한 기대에 부응하는 가장 좋은 방법 중 하나는 익숙한 영역에서 벗어나 다양한 언어, 플랫폼, 기술 분야에서 경험을 쌓을 기회를 적극적으로 찾는 것입니다. 아키텍트는 기술적 깊이보다는 기술적 폭에 집중해야 합니다. 기술적 폭이란 세부적인 수준까지는 아니더라도 알고 있는 분야와 깊이 있게 알고 있는 분야를 모두 포함합니다. 예를 들어, 아키텍트에게 있어 10가지 캐싱 제품의 장단점을 잘 아는 것이 단 하나의 제품에만 전문가인 것보다 훨씬 더 가치 있는 일입니다.

비즈니스 영역을 파악하세요

건축가는 일정 수준의 비즈니스 영역 전문 지식을 갖추고 있어야 합니다.

유능한 소프트웨어 아키텍트는 아키텍처가 해결하고자 하는 비즈니스 문제, 목표 및 요구 사항, 즉 문제 영역의 비즈니스 도메인을 이해합니다. 비즈니스 요구 사항을 이해하지 못하면 효과적인 아키텍처를 설계하기 어렵습니다. 대형 은행의 아키텍트라고 가정해 보세요. 평균 방향성 지수, 확률 계약, 금리 급등, 심지어 비우선 채권과 같은 일반적인 금융 용어를 이해하지 못한다면 이해관계자 및 비즈니스 사용자와 소통할 수 없을 뿐더러 신뢰를 빠르게 잃게 될 것입니다.

우리가 아는 가장 성공적인 아키텍트들은 폭넓고 실무적인 기술 지식과 특정 분야에 대한 깊이 있는 이해를 갖추고 있습니다. 또한 최고 경영진과 비즈니스 사용자들이 알고 이해하는 언어로 소통하여, 자신들이 무엇을 하고 있는지 정확히 알고 있으며 효과적이고 올바른 아키텍처를 설계할 역량을 갖추고 있다는 확신을 심어줍니다.

건축가는 팀워크, 진행 능력, 리더십을

포함한 뛰어난 대인 관계 능력을 갖추어야 합니다.

기술 전문가, 개발자, 아키텍트는 기술적인 문제 해결에 집중하는 경향이 있어 사람 문제를 다루는데 어려움을 느낄 수 있습니다. 따라서 뛰어난 리더십과 대인 관계 능력을 기대하기는 쉽지 않습니다. 하지만 [제럴드 와인버그](#)의 명언처럼 "사람들이 뭐라고 하든, 결국 문제는 사람입니다." 아키텍트가 제공하는 지침은 기술적인 부분에만 국한되지 않습니다. 개발팀이 아키텍처를 구현하는 과정을 이끌어가는 것 또한 아키텍트의 역할입니다. 직책이나 역할에 관계없이, 효과적인 소프트웨어 아키텍트가 되기 위해서는 리더십 능력이 최소 절반은 필수적입니다.

소프트웨어 아키텍트 시장은 포화 상태이며, 제한된 일자리를 놓고 모두가 경쟁하고 있습니다. 뛰어난 리더십과 대인 관계 능력을 갖춘 아키텍트는 단연 돋보입니다. 반대로, 기술적으로는 탁월하지만 팀을 이끌거나 개발자를 코칭 및 멘토링하거나 아이디어와 아키텍처 결정 및 원칙을 전달하는 데 어려움을 겪는 소프트웨어 아키텍트도 많이 있습니다. 당연히 이러한 아키텍트들은 직장을 오래 유지하기 어렵습니다.

건축가는 기업의 정치적 환경을 이해하고
그 안에서 효과적으로 대처할 수 있어야 합니다 .

소프트웨어 아키텍처에 관한 책에서 사내 정치에 대해 이야기하는 것이 이상하게 보일 수도 있지만, 협상 능력은 이 직무에 매우 중요합니다. 예를 들어 두 가지 시나리오를 살펴보겠습니다.

첫 번째 시나리오에서 개발자는 복잡한 코드의 복잡성을 줄이기 위해 특정 디자인 패턴을 활용하기로 결정합니다. 이는 훌륭한 결정이며, 개발자는 이에 대한 승인을 구할 필요가 없습니다. 코드 구조, 클래스 설계, 디자인 패턴 선택, 심지어 언어 선택과 같은 프로그래밍 측면은 모두 프로그래밍의 예술적인 영역에 속합니다.

두 번째 시나리오에서, 대규모 고객 관계 관리(CRM) 시스템을 담당하는 아키텍트는 다른 시스템의 데이터베이스 접근 제어, 특정 고객 데이터 보안, 그리고 데이터베이스 스키마 변경에 어려움을 겪고 있습니다. 이러한 문제들은 모두 너무 많은 시스템이 CRM 데이터베이스를 사용하고 있기 때문에 발생합니다. 따라서 아키텍트는 각 애플리케이션의 데이터베이스가 해당 데이터베이스를 소유한 애플리케이션에서만 접근 가능하도록 애플리케이션 사일로를 구축하기로 결정합니다. 이러한 결정은 아키텍트에게 고객 데이터, 보안 및 변경 사항에 대한 더 나은 제어권을 제공할 것입니다.

하지만 첫 번째 시나리오에서 개발자의 결정과는 달리, 아키텍트는 CRM 애플리케이션 팀을 제외한 회사 내 거의 모든 구성원이 자신의 결정에 이의를 제기할 것을 예상해야 합니다. 다른 애플리케이션들도 CRM 데이터가 필요하며, 데이터베이스에 직접 접근할 수 없게 되면 원격 접속을 통해 CRM 시스템에 데이터를 요청해야 합니다. 제품 소유자, 프로젝트 관리자, 그리고 비즈니스 이해관계자들은 비용이나 노력 증가에 반대할 수 있는 반면, 개발자들은 자신들의 접근 방식이 더 낫다고 생각할 수 있습니다. 아키텍트가 내리는 거의 모든 결정은 이의 제기를 받게 될 것입니다.

어떤 반대가 있더라도, 아키텍트는 조직 내 정치적 역학 관계를 헤쳐나가고 협상 기술을 활용하여 대부분의 결정을 승인받아야 합니다. 이는 매우 답답한 일일 수 있습니다. 대부분의 소프트웨어 아키텍트는 개발자로 경력을 시작하여 승인이나 검토 없이 스스로 결정을 내리는 데 익숙해져 있기 때문입니다. 아키텍트가 된 지금에서야 광범위하고 중요한 결정을 내릴 수 있게 되었지만, 거의 모든 결정에 대해 정당성을 입증하고 반대 의견을 수용해야 합니다. 협상 기술은 리더십 기술과 마찬가지로 매우 중요하기 때문에, 이 장(25장) 전체를 할애하여 다루고 있습니다.

로드맵

이 책은 세 부분으로 구성되어 있습니다.

파트 I: 기초 파트 I에서는 소

프트웨어 아키텍처의 핵심 구성 요소를 정의하고, 아키텍처 구조의 두 가지 핵심 요소인 아키텍처 특성과 논리적 구성 요소에 초점을 맞춥니다. 이러한 각 요소를 분석하는 데에는 서로 다른 기법이 필요하며, 본 파트에서는 이러한 기법들을 심도 있게 다룹니다. 이러한 활동을 통해 얻은 결과물은 소프트웨어 아키텍트에게 구현의 전반적인 철학을 뒷받침할 적절한 아키텍처 스타일을 선택하는 데 필요한 충분한 정보를 제공합니다.

제2부: 아키텍처 스타일 제2부에서는

소프트웨어 아키텍처의 명명된 토플로지인 아키텍처 스타일 목록을 제공합니다. 각 스타일의 구조적 및 통신적 차이점을 보여주고, 데이터 토플로지, 팀 구성, 물리적 아키텍처를 포함한 광범위한 영역에서 비교할 수 있는 기준을 제시합니다.

파트 III: 기술 및 소프트 스킬 파트 I과 II에서는 업

무의 기술적인 측면에 초점을 맞추었지만, 소프트웨어 아키텍트의 역할에서 중요한 부분은 전통적으로 소프트 스킬이라고 불리는, 기술이 아닌 다른 사람과의 소통 능력을 필요로 합니다. 아이러니하게도, 소프트웨어 아키텍트로 가는 주된 경로는 기술적 능력보다는 대인관계 능력을 중시하는 경우가 많기 때문에, 소프트 스킬은 신진 아키텍트들이 습득하기 가장 어려운 역량 중 하나입니다. 그러나 일단 실무에 투입되면 이러한 능력이 얼마나 중요한지 깨닫게 됩니다. 따라서 본 책의 파트 III에서는 소프트웨어 아키텍트로서 성공하는데 도움이 되는 몇 가지 핵심 소프트 스킬을 다룹니다.

제1부

기초

아키텍처에서 중요한 절충점을 이해하려면 개발자는 구성 요소, 모듈성, 결합도 및 연결성에 관한 몇 가지 기본 개념과 용어를 이해해야 합니다.

