문제를 보는 눈

읽기 자료

박정수 (Park Jeongsoo) <toracle@gmail.com>

목차

1. 지도를 그리는 사람 . . . **2**
1.1. 이야기 . . . 2
1.2. 이 시리즈가 다루는 것 . . . 3
1.3. 이 가이드북의 사용법 . . . 3
1.4. 등장인물 . . . 4
1.5. 시리즈 차례 . . . 4
2. 승현의 30초 . . . **6**
2.1. 이야기 . . . 6
2.2. 개념 노트 . . . 7
2.2.1. 왜 우리는 30초 만에 답을 내리는가 . . . 7
2.2.2. Presumptive Architecture — 내 머릿속의 디폴트 . . . 8
2.2.3. Problem Space와 Solution Space . . . 8
2.2.4. 경험의 역설 . . . 9
2.3. 이 강에서 써볼 것 . . . 10
3. 민준의 말 속에 숨겨진 세 가지 . . . **11**
3.1. 이야기 . . . 11
3.2. 개념 노트 . . . 13
3.2.1. 요구사항의 세 가지 층 . . . 13
3.2.2. 모호함은 세 곳에서 온다 . . . 14
3.2.3. "문서가 아니라 문서화가 전부다" . . . 14
3.3. 이 강에서 써볼 것 . . . 14
3.3.1. FAC가 단순화를 만드는 방법 . . . 15
4. 질문하는 자가 설계한다 . . . **16**
4.1. 이야기 . . . 16
4.2. 개념 노트 . . . 18
4.2.1. 왜 직접 질문은 좁히는가 . . . 18
4.2.2. Meta-Question — 가장 저평가된 질문 . . . 18
4.2.3. Black Box — 구현 전에 합의하는 것 . . . 19
4.2.4. 임시방편을 물어라 . . . 19
4.3. 이 강에서 써볼 것 . . . 19
5. 지도를 그리기 전에 . . . **21**
5.1. 이야기 . . . 21
5.2. 개념 노트 . . . 23
5.2.1. 증상과 문제와 근본원인 . . . 23
5.2.2. 같은 것을 다르게 프레임하면 다른 문제가 된다 . . . 23
5.2.3. Domain Model — 코드를 짜기 전에 세계를 그린다 . . . 24
5.3. 이 강에서 써볼 것 . . . 24

5.3.1. 레이어 선택이 범위를 결정한다 25
6. 안개 속에서 걷는 법 26
6.1. 이야기 26
6.2. 개념 노트 28
6.2.1. 두 가지 모르는 것 28
6.2.2. "가장 단순한 것"의 의미 29
6.2.3. Technical Debt — 오해된 개념 29
6.2.4. 행동하면서 아는 것 30
6.3. 이 강에서 써볼 것 30
7. 이 문제, 어디서 본 것 같은데 32
7.1. 이야기 32
7.2. 개념 노트 35
7.2.1. Polya의 질문 35
7.2.2. 유사 추론의 세 단계 35
7.2.3. 환원 — 이건 어떤 종류의 문제인가 35
7.2.4. 기능 목록이 아니라 Trade-off를 읽는다 36
7.2.5. 경험이 쌓이는 방식 36
7.3. 이 강에서 써볼 것 37
8. 어디에 공을 들여야 하는가 38
8.1. 이야기 38
8.2. 개념 노트 40
8.2.1. 기능과 품질 속성은 다르게 쌓인다 40
8.2.2. Risk-Driven Model — 리스크에 비례해서 설계한다 41
8.2.3. Engineering Risk와 PM Risk는 다른 도구로 해소한다 41
8.2.4. "지금 안 해도 된다"는 결정 42
8.3. 이 강에서 써볼 것 42
9. 승현의 지도 43
9.1. 이야기 43
9.2. 개념 노트 45
9.2.1. 문제를 이해하면 해결책이 단순해진다 45
9.2.2. Problem-Solution Fit 46
9.2.3. Pre-mortem — 성공 편향 없이 약점 찾기 46
9.3. 이 강에서 가지고 가는 것 47
10. 지도를 그리는 법을 배운 사람 48
10.1. 이야기 48
10.2. 마무리 — 독자에게 49
10.3. 이 워크샵에서 배운 것들 49
10.4. 그래서, 무엇이 달라지는가 51

구성: 프롤로그 + 에피소드 1-8 + 에필로그

활용 방법:

- 각 강의 전날 밤: 해당 에피소드를 읽고 옵니다 (15-20분)
- 각 강의 후 복습: 개념 노트와 "이 강에서 써볼 것"을 다시 읽습니다
- 워크샵 없이도 독립적으로 읽을 수 있습니다

Chapter 1. 지도를 그리는 사람

> "지도를 주는 것은 쉽다. 지도를 그리는 법을 가르치는 것은 어렵다. 그러나 그것만이 진짜 길을 찾는 법이다."

1.1. 이야기

처음 일을 시작했을 때, 승현은 빠른 것이 곧 잘하는 것이라고 생각했다.

요구사항이 들어오면 가장 먼저 노트북을 열었다. 머릿속에서 구조가 그려지는 속도가 능력의 증거라고 믿었다. 실제로 그 믿음이 틀리지 않았다. 빠르게 만들었고, 대부분 동작했고, 팀에서 인정받았다.

그런데 3년차가 지나면서 이상한 일이 생기기 시작했다.

만들고 나서 다시 만드는 일이 늘었다. 열심히 짠 코드가 "이게 제가 원한 게 아니에요"라는 말에 버려졌다. 완성했다고 생각한 기능이 두 달 후에 "그게 사실 필요 없어졌어요"로 끝났다. 기술적으로는 맞는 것을 만들었는데, 왜 자꾸 틀린 것이 됐는지 알 수 없었다.

그러던 어느 날, 민준이 가져온 한 문장이 이 이야기의 시작이 됐다.

"팀 협업 기능 강화해주세요. 알림이랑 업무 배정 쪽이요. 다음 달 데모."

평소의 승현이었다면 15초 만에 WebSocket을 생각했을 것이다. 실제로 그랬다. 하지만 이번에는 뭔가가 그를 멈추게 했다.

나는 지금 무엇을 알고 있는가?

이 워크샵은 승현이 그 질문에서 시작해서, 8강에 걸쳐 배운 것들의 이야기입니다.

기술적인 이야기가 아닙니다. 어떤 프레임워크나 언어의 이야기도 아닙니다.

문제를 보는 법에 관한 이야기입니다.

1.2. 이 시리즈가 다루는 것

소프트웨어를 만드는 사람들은 두 가지 공간을 오갑니다.

Problem Space — 무엇이 문제인가, 누가 고통받는가, 왜 이것이 문제인가.

Solution Space — 어떻게 풀 것인가, 무엇을 만들 것인가, 어떤 기술을 쓸 것인가.

문제는 대부분의 엔지니어가 Problem Space에 충분히 머물지 않는다는 것입니다. 요구사항을 듣는 순간, 뇌는 자동으로 Solution Space로 넘어갑니다. 경험이 쌓일수록 이 전환은 더 빠르고 더 자연스럽게 일어납니다.

그리고 대부분의 재작업은 거기서 시작됩니다.

이 워크샵이 가르치는 것은 두 가지입니다.

첫째, Problem Space를 깊이 탐색하는 도구들. 요구사항을 해부하는 언어, 모호함을 다루는 방법, 문제의 구조를 파악하는 기술.

둘째, 그리고 이것이 더 중요합니다 — 문제를 깊이 이해하면 해결책이 오히려 단순해진다는 것.

핵심적으로 달성해야 할 것이 명확해지고, 나중에 해도 되는 것이 보이고, 하지 않아도 되는 것이 식별됩니다. 그 결과로 — 더 작고 더 빠르게 만들어서, 더 일찍 사용자에게 전달할 수 있게 됩니다.

Problem Space를 더 오래 탐색하는 것이 느린 것처럼 보일 수 있습니다. 하지만 경험상, 그 탐색이 Solution Space에서의 재작업을 막습니다.

1.3. 이 가이드북의 사용법

각 에피소드는 강의 전날 밤 읽어오거나, 강의 후 복습용으로 활용합니다.

이야기를 읽을 때, 승현의 상황에 자신을 대입해보세요. 지연의 선택이 자신의 과거처럼 느껴지는 순간이 있을 것입니다. 현태의 발견이 새롭게 느껴지는 순간도 있을 것입니다.

개념 노트는 이야기에서 일어난 것을 이론 언어로 번역합니다. 워크샵에서 실습을 마친 후 다시 읽으면 — 처음 읽을 때와 다르게 읽힐 것입니다.

그것이 이 구조의 의도입니다.

1.4. 등장인물

승현 (32세, 풀스택 엔지니어, 5년차) 이 이야기의 시점 인물. 잘하고 싶은데 뭔가 자꾸 엇나간다는 느낌을 가지고 있다. 기술적으로는 팀에서 가장 능숙하지만, 최근 들어 그것만으로는 부족하다는 것을 어렴풋이 느낀다.

지연 (28세, 풀스택 엔지니어, 3년차) 빠르게 만드는 것에 자신 있다. 요구사항이 들어오면 손이 먼저 움직인다. 틀린 게 아니다. 다만 아직 모르는 것이 있다.

현태 (26세, 풀스택 엔지니어, 1년차) 팀의 막내. 모르는 것이 많지만 그래서 오히려 당연한 것을 묻는다. 그 질문들이 종종 팀에서 가장 중요한 것을 짚어낸다.

민준 (31세, BizDev) 고객사와 계약을 따오는 사람. 기술은 잘 모르지만 현장 감각이 날카롭다. 그가 가져오는 요구사항은 항상 모호하다. 그것이 그의 잘못은 아니다.

1.5. 시리즈 차례

강	에피소드 제목	핵심 질문
프롤로그	지도를 그리는 사람	왜 이 이야기를 읽어야 하는가
1강	승현의 30초	나는 지금 문제를 이해한 건가, 패턴을 인식한 건가?
2강	민준의 말 속에 숨겨진 세 가지	요구사항이 "명확하다"는 것은 어떤 의미인가?
3강	질문하는 자가 설계한다	BizDev이 원하는 것과 사용자가 필요한 것은 다르다
4강	지도를 그리기 전에	문제를 충분히 이해했는지 어떻게 아는가?
5강	안개 속에서 걷는 법	해소할 수 없는 모호함은 어떻게 다루는가?

강	에피소드 제목	핵심 질문
6강	이 문제, 어디서 본 것 같은데	이 문제를 처음 보는 게 아닐 수 있다
7강	어디에 공을 들여야 하는가	모든 설계에 같은 에너지를 써야 하는가?
8강	승현의 지도	해결책이 그럴듯해 보이는 것과 실제로 작동하는 것은 다르다
에필로그	지도를 그리는 법을 배운 사람	이 워크샵이 끝난 후

첫 번째 에피소드: 승현의 30초 '"

Chapter 2. 승현의 30초

> "경험이 많을수록 답이 빠르게 나온다. 문제는, 그 답이 질문보다 먼저 나온다는 것이다."

2.1. 이야기

월요일 오후 세 시.

민준이 고객사 미팅을 마치고 사무실로 돌아왔다. 노트북 가방을 내려놓기도 전에 슬랙 메시지가 왔다.

> 민준: 야 승현아, 나 왔어. 저쪽이랑 얘기 됐는데, 팀 협업 기능 좀 강화해달래. 알림이랑 업무 배정 쪽. 다음 달 데모 보여줘야 해.

승현은 메시지를 읽었다. 그리고 멈췄다.

아, 알림. 업무 배정.

머릿속에서 무언가가 자동으로 작동하기 시작했다. 알림이면 `WebSocket`이나 `SSE`. 아니면 `Firebase`. 업무 배정이면 `assignee` 컬럼, 아니면 별도 `assignment` 테이블. 실시간 업데이트가 필요하면 `pub/sub` 구조가 맞겠지. 다음 달이면 한 달. 한 달이면…

15초도 지나지 않아 승현의 머릿속에는 이미 대략적인 구조가 그려져 있었다.

옆자리에서 지연이 의자를 돌리며 말했다.

"오빠, 봤어요? 알림 기능이요. 저 Kafka 써보고 싶었는데 이참에 써봐도 되겠다."

승현은 대답하려다가 멈췄다.

잠깐.

뭔가 이상했다. 메시지를 받은 지 30초도 안 됐다. 그런데 자신도, 지연도 이미 기술 이야기를 하고 있었다. 민준에게 아무것도 물어보지 않은 채로.

나는 지금 뭘 알고 있는 거지?

승현은 슬랙 메시지를 다시 읽었다.

"팀 협업 기능 강화." "알림이랑 업무 배정." "다음 달 데모."

세 문장. 그게 전부였다.

지금 고객사가 어떤 팀 규모인지 모른다. 알림을 왜 강화해달라는 건지 모른다. 현재 알림이 어떻게 되어 있는지도 모른다. 업무 배정이 지금 어떻게 이루어지고 있는지도 모른다. 데모에서 뭘 보여줘야 하는지도 모른다.

그런데 자신은 15초 만에 WebSocket을 생각하고 있었다.

승현은 자리에서 일어나 민준에게 걸어갔다.

"민준아, 저쪽 팀이 지금 어떻게 일하고 있어? 알림을 강화해달라는 게 구체적으로 어떤 상황에서 나온 말이야?"

민준이 노트북을 열며 대답했다.

"아, 그게 팀장이 업무를 배정하면 팀원들이 모른대. 슬랙으로 따로 연락해야 하는데 그게 너무 번거롭다고."

아.

승현은 속으로 생각했다.

이건 알림 기술의 문제가 아니라, 업무 배정 흐름 자체가 시스템에 연결되어 있지 않은 문제구나.

WebSocket이 필요한지 아닌지는 아직 모른다. 어쩌면 이메일 알림 하나로 충분할 수도 있다. 어쩌면 알림보다 업무 배정 화면 자체를 먼저 만들어야 할 수도 있다.

그 30초 동안, 승현은 무언가를 배웠다.

자신이 "문제를 이해했다"고 느낀 것이, 실제로는 패턴을 인식한 것이었다는 사실을.

2.2. 개념 노트

2.2.1. 왜 우리는 30초 만에 답을 내리는가

승현에게 일어난 일은 드문 사례가 아닙니다. 경험이 쌓인 엔지니어일수록 더 빠르고 더 자연스럽게 일어납니다.

심리학에서는 이것을 아인슈텔룽 효과(Einstellung Effect)[1] 라고 부릅니다. 과거에 잘 작동했던 해결책이 마음속에 준비된 채로 있다가, 비슷한 신호가 들어오는 순간 자동으로 발동하는 현상입니다. 뇌가 효율을 위해 만들어낸 장치입니다.

문제는 이 장치가 새로운 문제를 보는 눈을 가린다는 것입니다. 더 나은 해결책이 눈앞에 있어도, 익숙한 답이 먼저 나오면 그것이 보이지 않습니다. 그리고 아이러니하게도, 경험이 많을수록 이 효과가 더 강하게 작동합니다.

```
경험 적음 → 패턴 없음 → 느리지만 탐색적
경험 많음 → 패턴 풍부 → 빠르지만 자동적
                              ↑
                          여기가 함정
```

2.2.2. Presumptive Architecture — 내 머릿속의 디폴트

소프트웨어 아키텍처 연구자 George Fairbanks는 이것을 추정 아키텍처(Presumptive Architecture)[2] 라는 개념으로 설명합니다.

모든 개발자는 "이런 상황엔 이 아키텍처"라는 무의식적 디폴트를 가지고 있습니다.

- 실시간 알림 → `pub/sub`
- 배치 처리 → 큐
- 검색 → 엘라스틱서치
- 대용량 → `Kafka`

이 디폴트들은 틀린 게 아닙니다. 특정 상황에서는 완벽하게 맞습니다. 하지만 지금 우리가 다루는 문제에서 맞는지는 아직 모릅니다.

Presumptive Architecture의 진짜 위험은 그것이 가정(assumption)인지 분석(analysis)인지 구분하지 않고 쓸 때 생깁니다. 지연이 "Kafka 써보고 싶었는데"라고 말한 것처럼, 기술 선택이 문제 분석보다 먼저 나올 때.

2.2.3. Problem Space와 Solution Space

Polya는 문제 해결을 두 공간으로 나누어 설명합니다.[3]

Problem Space	Solution Space
"무엇이 문제인가"	"어떻게 풀 것인가"

Problem Space	Solution Space
"누가 고통받는가"	"무엇을 만들 것인가"
"왜 이것이 문제인가"	"어떤 기술을 쓸 것인가"
"이것이 정말 문제인가"	"어떤 구조로 설계할 것인가"
도구: 질문, 관찰, 인터뷰	도구: 설계, 코드, 다이어그램

두 공간 모두 반드시 거쳐야 합니다. Solution Space로 가지 말라는 이야기가 아닙니다. 핵심은 이동이 의식적이어야 한다는 것입니다.

승현이 15초 만에 WebSocket을 생각한 것은 자동으로, 무의식적으로 이동한 것이었습니다. 그가 민준에게 걸어간 것은, 그 이동을 의식적으로 되돌린 것이었습니다.

2.2.4. 경험의 역설

3-4년차 엔지니어가 특히 이 함정에 빠지기 쉬운 이유가 있습니다.

1년차는 모르는 게 많아서 질문을 많이 합니다. 10년차는 경험이 쌓이면서 자신의 패턴을 의심할 줄 압니다. 그런데 3-4년차는 충분히 많은 패턴을 알고 있어서, 문제를 보기 전에 답이 나옵니다. 그리고 그 답이 꽤 그럴듯해 보입니다.

이것이 경험의 역설입니다.

이 워크샵이 목표로 하는 것은 경험을 버리는 것이 아닙니다. 경험을 가정으로 인식하는 것입니다. "이건 pub/sub 문제처럼 보인다"는 직관은 유효합니다. 다만 그것이 "이건 pub/sub 문제다"로 굳어지기 전에, Problem Space에서 잠시 머무는 근력을 키우는 것.



경험은 자산입니다. 경험에서 나온 패턴 직관을 분석(analysis) 이 아니라 가정(assumption) 으로 취급하는 순간, 그 자산을 더 잘 쓸 수 있게 됩니다.

승현의 30초가 그 훈련의 시작이었습니다.

2.3. 이 강에서 써볼 것

다음 두 가지를 이번 주에 의식적으로 해보세요.

첫째. 요구사항이나 업무 지시를 받는 순간, 자신에게 물어보세요. "나는 지금 문제를 이해한 건가, 패턴을 인식한 건가?"

둘째. 머릿속에 떠오른 해결책을 잠시 내려놓고, Problem Space 질문을 5개 이상 만들어보세요. "아직 내가 모르는 것은 무엇인가?"

이것이 어색하고 불편하다면, 잘하고 있는 겁니다.

다음 에피소드: 민준의 말 속에 숨겨진 세 가지 — 요구사항을 Functions, Attributes, Constraints로 해부하는 법

---

[1] Einstellung(독일어로 "태도" 또는 "설정") 효과. Luchins(1942)의 물 담기 실험에서 처음 체계화된 개념으로, 기존에 성공한 해결 방식이 더 효율적인 대안을 인식하는 것을 방해하는 인지 현상. Abraham S. Luchins, Mechanization in Problem Solving, 1942.

[2] George Fairbanks, Just Enough Software Architecture, 2010. 아키텍처 결정에서 명시적 분석 없이 자동으로 선택되는 "기본값" 아키텍처 패턴을 지칭하는 개념.

[3] George Pólya, How to Solve It, 1945. 수학 문제 풀이 전략을 체계화한 고전으로, 문제 이해 → 계획 수립 → 실행 → 검토의 4단계 프레임을 제시한다. 소프트웨어 설계에도 널리 적용된다.

Chapter 3. 민준의 말 속에 숨겨진 세 가지

> "요구사항이 불명확한 게 아니다. 세 가지 다른 종류의 것이 뒤섞여 있을 뿐이다."

3.1. 이야기

목요일 오전.

승현과 지연은 민준과 함께 작은 회의실에 모였다. 화이트보드 한쪽에는 지난번 민준의 말이 적혀 있었다.

"팀 협업 기능 강화. 알림이랑 업무 배정. 다음 달 데모."

지연이 노트북을 열며 말했다.

"오빠, 저 어젯밤에 정리해봤는데요. 알림은 SSE로 하면 될 것 같고, 업무 배정은 assignee 필드 추가하면 기본은 되잖아요. 아마 2주면 MVP 나올 것 같은데."

민준이 고개를 끄덕였다. "오, 빠르네."

승현은 뭔가 말하려다가 멈췄다. 지연의 말이 틀린 것은 아니었다. 기술적으로는 충분히 가능한 이야기였다. 하지만 뭔가 빠진 느낌이 들었다.

알림을 보낸다 — 어떤 알림을? 어느 시점에? 누가 받아야 하는가? 업무를 배정한다 — 현재 어떻게 배정하고 있는가? 무엇이 불편한가? 데모 — 고객사가 데모에서 보고 싶은 게 정확히 무엇인가?

승현이 입을 열었다.

"민준아, 잠깐. 질문 하나만. 지금 고객사에서 업무 배정이 안 된다고 했잖아. 근데 그게 구체적으로 어떤 상황이야? 팀장이 배정을 했는데 팀원이 모른다는 건가, 아니면 배정 자체가 안 되어 있다는 건가?"

민준이 잠시 생각했다.

"아, 배정은 돼. 시스템 안에서. 근데 그게 배정되면 팀원한테 연락이 안 가. 팀장이 직접 슬랙으로 '이거 해줘' 라고 따로 연락해야 해."

아. 승현이 적기 시작했다.

"그러면 알림이 필요한 건 배정됐을 때만이야, 아니면 상태가 바뀔 때도?"

"음… 배정됐을 때랑, 마감일 가까워질 때 정도?"

"고객사 팀장이 쓰는 환경이 어때? PC야, 모바일도 쓰나?"

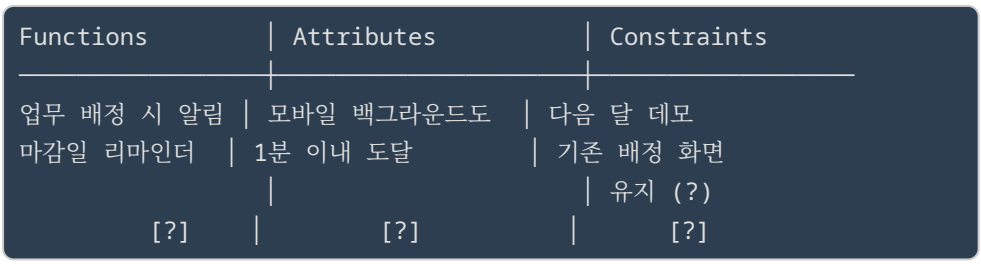
"아, 그분들 제조업체라 현장에 많이 나가거든. 모바일도 있으면 좋다고 했어."

지연이 조용히 노트북을 닫으며 말했다.

"…SSE는 모바일에서 백그라운드 상태일 때 안 오네요."

침묵.

승현이 화이트보드 앞에 섰다. 민준이 한 말들을 세 칸으로 나누어 적기 시작했다.



"민준아, 이 세 칸에 '?'가 많은 게 보여? 이게 지금 우리가 모르는 것들이야. 이걸 모르고 만들기 시작하면 나중에 다 뜯어야 해."

민준이 화이트보드를 한참 쳐다봤다.

"근데 이거… 내가 고객사한테 더 물어봐야 하는 것도 있네."

"맞아. Attribute 쪽 — 알림이 얼마나 빨리 가야 하는지, 못 봤을 때 재알림이 있어야 하는지. 이건 내가 민준이한테 물어봐도 민준이가 모를 수 있어. 고객사 사용자한테 직접 가야 해."

지연이 말했다.

"그럼 2주 MVP는…"

승현이 웃으며 말했다.

"뭘 만들지 알고 나서 2주야."

3.2. 개념 노트

3.2.1. 요구사항의 세 가지 층

이 이야기에서 승현이 화이트보드에 그린 세 칸은, Weinberg가 Exploring Requirements에서 제시한 FAC 프레임[1]입니다.

모든 요구사항은 표면적으로는 하나의 문장처럼 보이지만, 실제로는 세 가지 다른 종류의 정보가 뒤섞여 있습니다.

Functions — 무엇을 하는가

시스템이 수행하는 동작. 가장 눈에 잘 보이고, 대화에서 가장 먼저 나옵니다.

"알림을 보낸다", "업무를 배정한다", "목록을 보여준다."

BizDev도, 사용자도 자연스럽게 Function 언어로 이야기합니다. 그래서 엔지니어는 Function만 듣고 끝내는 경향이 있습니다.

Attributes — 어떻게 해야 하는가

Function의 품질 기준입니다. "얼마나 빠르게", "얼마나 안정적으로", "어떤 환경에서", "어떤 형태로".

Attribute는 대화에서 좀처럼 먼저 나오지 않습니다. 말하는 사람도 당연하다고 여겨서 생략하거나, 아직 명확하게 생각해보지 않았거나. 하지만 Attribute는 Architecture를 결정합니다.

지연의 `SSE` 선택이 흔들린 것처럼 — 모바일 백그라운드라는 Attribute를 몰랐기 때문이었습니다.

Constraints — 건드릴 수 없는 것

해결책이 지켜야 할 조건입니다. 기술적인 것(기존 시스템, 인프라), 비즈니스적인 것(예산, 일정), 법적인 것(규정, 계약).

Constraint는 가장 캐내기 어렵습니다. 당사자에게 너무 당연해서 말하지 않습니다. 그리고 가장 늦게 발견됩니다 — 보통 설계가 끝나고 나서.

```
발견 시점이 늦을수록 재작업 비용은 기하급수적으로 커집니다.
```

3.2.2. 모호함은 세 곳에서 온다

FAC를 채우는 것이 어려운 이유는 모호함이 세 가지 다른 출처에서 오기 때문입니다.

첫째, 언어 자체가 모호합니다. "빠른 알림"이라는 말에는 측정 기준이 없습니다. 이건 질문으로 해소할 수 있습니다. "얼마나 빠른 걸 의미하시나요?"

둘째, 사람마다 다르게 해석합니다. 같은 "알림"이라는 단어에서 BizDev는 PC 팝업을 상상하고, 개발자는 서버 push를 생각하고, 사용자는 카카오톡을 기대합니다. 이건 함께 그림을 그려봐야 해소됩니다.

셋째, 말하지 않은 전제가 있습니다. "기존 화면은 건드리지 말고"처럼, 당연하다고 여겨서 말하지 않은 것들. 이건 직접 캐내는 질문이 필요합니다. 하지만 어떻게 캐내는가 — 그게 3강의 주제입니다.

3.2.3. "문서가 아니라 문서화가 전부다"

승현이 화이트보드에 세 칸을 그린 것 자체가 중요한 행위였습니다.

Weinberg는 말합니다. 요구사항 문서를 만드는 것의 목적은 문서를 완성하는 것이 아닙니다. 그 과정에서 팀 전체가 같은 그림을 갖게 되는 것입니다.



화이트보드에 FAC 세 칸을 그리고 "?" 표시를 함께 채워가는 것 자체가 팀의 이해를 정렬하는 행위입니다. 완성된 문서가 아니라, 그 과정이 목적입니다.

승현이 "?"를 표시하는 것을 보면서 민준도 무언가를 발견했습니다. "이거 내가 고객사한테 더 물어봐야 하는 것도 있네." 화이트보드가 대화를 만들었습니다.

3.3. 이 강에서 써볼 것

다음 번에 요구사항을 받았을 때, 머릿속에서 자동으로 세 칸을 펼쳐보세요.

Functions: 이 문장에서 말하는 동작은 무엇인가?

Attributes: 그 동작의 품질 기준이 말해졌는가? 말해지지 않았다면, 어떤 기준이 Architecture에 영향을 줄 수 있는가?

Constraints: 내가 전제하고 있는 "당연한 것"이 있는가? 상대방이 말하지 않은 조건이 있는가?

특히 "?" 칸이 "확인됨" 칸보다 많은 것이 정상입니다. 모른다는 것을 아는 것이 시작입니다.

3.3.1. FAC가 단순화를 만드는 방법

FAC 체크에는 한 가지 더 중요한 효과가 있습니다.

"?" 칸을 채워가는 과정에서 — "이것은 우리가 만들 것이 아니다"가 보이기 시작합니다.

Attributes가 명확해지면, 그 기준을 충족하지 않아도 되는 기능들이 자연스럽게 범위에서 빠집니다. "응답시간 1초 이내"가 Attribute로 확인됐다면, 실시간이 필요 없는 부가 기능에 WebSocket을 쓸 이유가 없어집니다.

Constraints가 명확해지면, 그 조건 밖에서 작동해야 할 것들이 지금 만들 필요가 없는 것으로 분류됩니다. 예산 한도가 Constraint로 확인되면, 그 범위를 넘는 인프라는 논의 대상에서 빠집니다.

"?" 칸을 채우는 것은 만들어야 할 것을 찾는 작업인 동시에, 만들지 않아도 되는 것을 찾는 작업이기도 합니다. Problem Space를 더 깊이 탐색할수록 — Solution Space가 오히려 좁아지고 명확해집니다. 그 명확함이 더 빠른 전달을 가능하게 합니다.

다음 에피소드: 질문하는 자가 설계한다 — Context-Free Questions와 Black Box Test로 FAC의 빈 칸을 채우는 법 '"

[1] Gerald M. Weinberg & Donald C. Gause, Exploring Requirements: Quality Before Design, 1989. 요구사항을 Functions(기능), Attributes(속성), Constraints(제약)의 세 층으로 분리하는 프레임을 소개한다. FAC라는 약칭은 이 책에서 유래한다.

Chapter 4. 질문하는 자가 설계한다

> "직접 질문은 답을 좁힌다. 좋은 질문은 아직 보지 못한 것을 열어준다."

4.1. 이야기

금요일 오전.

승현, 지연, 민준이 다시 회의실에 모였다. 이번에는 민준이 고객사 담당자와 영상통화를 연결하겠다고 했다. 고객사 팀장 김 부장이었다.

영상이 연결됐다. 지연이 먼저 입을 열었다.

"안녕하세요, 부장님. 알림 기능 관련해서 여쭤보고 싶은 게 있어서요. 지금 슬랙으로 따로 연락하시는 부분을 자동화하면 어떨까 해서요. 이메일 알림이 좋으실까요, 아니면 앱 푸시가 좋으실까요?"

김 부장이 잠시 멈췄다.

"음… 이메일이나 앱 푸시나… 잘 모르겠네요. 그냥 편하면 되는 거라."

지연이 민준을 봤다. 민준이 어깨를 으쓱했다.

승현이 조용히 끼어들었다.

"부장님, 잠깐 다른 걸 여쭤봐도 될까요. 지금 업무 배정하실 때 어떻게 하고 계세요? 흐름을 한번 들어볼 수 있을까요?"

김 부장이 조금 더 편안한 표정이 됐다.

"아, 그거요. 저희가 현장에 있으면서 시스템에 배정 입력하고, 그 다음에 팀원한테 슬랙 보내요. '이거 맡아줘' 하고. 근데 팀원들이 못 보는 경우가 있어요. 현장에 있으면 슬랙을 잘 안 보거든요."

현장. 승현이 적었다.

"팀원분들도 현장에 많이 계세요?"

"네, 저희가 공장 라인 관리자들이라. 다들 현장에서 태블릿 들고 다니거나 그냥 폰이에요."

태블릿이나 폰. 또 적었다.

"업무를 놓쳤을 때 어떤 일이 생기나요?"

"라인 스탑이요. 제때 처리 안 하면 공정이 멈춰요. 그러면 저희가 다 뒤집어 쓰는 거죠."

방 안이 조용해졌다.

지연이 노트에 뭔가를 빠르게 고쳐 적고 있었다.

승현이 계속했다.

"부장님, 제가 이걸 여쭤봐도 될까요. 혹시 제가 놓친 부분이 있을까요? 제가 생각 못 한 상황이 있다면요."

김 부장이 잠깐 생각하더니 말했다.

"음… 저희 야간 조도 있어요. 야간에 긴급 배정이 생기면 더 빠르게 연락이 가야 하는데, 그때가 제일 문제예요."

승현이 멈췄다. 야간 긴급 배정. 이건 전혀 예상 못 한 것이었다.

통화가 끝나고, 지연이 노트북을 닫으며 말했다.

"저 처음에 이메일이냐 푸시냐 물어봤는데, 그게 얼마나 좁은 질문이었는지."

승현이 화이트보드에 적기 시작했다.

```
Trigger: 업무 배정 시 / 긴급 배정 시 (야간 포함)
Input:   팀장의 배정 행위 (시스템 내)
Output:  팀원 디바이스 (태블릿/폰)로 즉시 전달
예외:    팀원이 오프라인 상태일 때
         야간 수면 중 긴급 발생 시
```

"이제 좀 다르게 보이지?"

지연이 고개를 끄덕였다.

"라인 스탑이 걸린다는 걸 몰랐으면… 알림이 1분 늦어도 괜찮다고 생각했을 것 같아요."

승현이 덧붙였다.

"그리고 이메일은 선택지에서 빠졌어. 라인 스탑 막아야 하는데 이메일 보고 있을 사람은 없으니까."

4.2. 개념 노트

4.2.1. 왜 직접 질문은 좁히는가

지연의 첫 질문 — "이메일이 좋으실까요, 앱 푸시가 좋으실까요?" — 은 나쁜 질문이 아닙니다. 다만, 이미 해결책을 가정하고 있었습니다.

이 질문이 전제하는 것들:

- 알림이 필요하다 (사실이지만 아직 확인 안 됨)
- 이메일과 앱 푸시가 선택지다 (다른 방법은 없나?)
- 상대방이 이 두 가지의 차이를 안다 (실제로는 몰랐다)

Weinberg는 이런 질문을 Context-Specific Question이라고 부릅니다. 특정 맥락(여기서는 알림 기술)을 전제하고 시작하는 질문. 이런 질문은 그 맥락 안에서만 대화를 흐르게 합니다.

반대로, 승현의 "지금 업무 배정 흐름을 들어볼 수 있을까요?"는 어떤 해결책도 전제하지 않습니다. 문제 공간 자체를 탐색합니다. 이것이 Context-Free Question[1]입니다.

4.2.2. Meta-Question — 가장 저평가된 질문

승현이 마지막에 한 질문 — "제가 놓친 부분이 있을까요?" — 에서 야간 긴급 배정이 나왔습니다.

이 정보는 다른 어떤 질문으로도 나오지 않았을 것입니다. "야간 운영하시나요?"라고 먼저 물어볼 이유가 없었으니까요.

Meta-Question이 강력한 이유는 이것입니다. 대화는 질문하는 사람이 알고 있는 영역 안에서만 흐릅니다. Meta-Question은 상대방에게 그 경계를 넘을 권한을 줍니다.

```
일반 질문: 내가 아는 영역에서 → 상대방에게 묻는다
Meta-Question: 내가 모르는 영역이 있다는 것을 → 상대방이 알려준다
```

"제가 놓친 게 있을까요?"는 어떤 상황에서도 쓸 수 있는 가장 안전하고 강력한 질문입니다.

4.2.3. Black Box — 구현 전에 합의하는 것

승현이 통화 후 화이트보드에 적은 것은 Black Box Template의 핵심 요소들이었습니다.

Black Box를 먼저 합의하면 두 가지 일이 생깁니다.

하나, 구현 방법 논쟁보다 경계 합의가 먼저 됩니다. "어떻게 만들까"보다 "무엇이 들어가고 나와야 하는가"가 먼저 정해집니다. 이것이 합의되면 구현 방법은 엔지니어가 결정할 수 있는 영역이 됩니다.

둘, 예외 케이스가 자연스럽게 나옵니다. "이 기능이 동작 안 하는 경우는?"이라는 질문이 Black Box 경계에서 자연스럽게 생깁니다. 라인 스탑이 걸리는 환경에서 팀원이 오프라인이면 어떻게 되는가 — 이런 예외는 구현 중에 발견하면 비용이 큽니다.

4.2.4. 임시방편을 물어라

Weinberg가 특히 강조하는 Process Question 하나를 기억해 두세요.

"지금 임시방편으로 어떻게 하고 있나요?"[2]

사람들은 말로는 표현 못 하는 것을 행동으로 이미 하고 있습니다. 슬랙으로 따로 연락하는 것이 임시방편이었습니다. 그 임시방편이 정확히 무엇인지를 알면, 진짜 문제가 보입니다. 임시방편이 없다면 — 그만큼 급하지 않은 문제입니다.

4.3. 이 강에서 써볼 것

다음 번에 요구사항 대화를 할 때, 세 가지를 의식하세요.

첫째. 내가 막 하려는 질문이 해결책을 가정하고 있지 않은지 확인하세요. "A가 좋을까요, B가 좋을까요?"는 A와 B 중 하나가 답이라는 전제입니다.

둘째. 대화가 막힐 때, "지금 이걸 어떻게 해결하고 계세요?"를 써보세요. 임시방편이 나옵니다.

셋째. 대화를 마무리하기 전에 한 번은, "제가 놓친 게 있을까요?"를 말해보세요. 가장 중요한 것이 여기서 나올 수 있습니다.

다음 에피소드: 지도를 그리기 전에 — 증상과 문제와 근본원인을 구분하고, Domain Model로 문제 공간의 지도를 그리는 법 '"

[1] Weinberg & Gause, Exploring Requirements, 1989. Context-Free Question은 답변자가 어떤 맥락을 전제해도 좋도록 열린 형태로 설계된 질문이다. "지금 어떻게 하고 계세요?", "무엇이 이상적인가요?" 등이 전형적인 예다.

[2] Weinberg는 이를 "Process Question" 또는 "현재 해결 방법 질문"이라고 부른다. 사람들이 언어로 표현하지 못하는 문제를 이미 행동으로 해결하고 있다는 관찰에서 출발한다. Exploring Requirements, 1989.

Chapter 5. 지도를 그리기 전에

> "지도 없이 길을 찾다 보면, 내가 가야 할 곳이 아닌 곳에 도착한다."

5.1. 이야기

월요일 오후.

승현, 지연, 그리고 팀의 막내 현태가 화이트보드 앞에 모였다. 김 부장과의 통화 이후 처음 갖는 설계 회의였다.

지연이 먼저 화이트보드에 그리기 시작했다.

"제가 생각한 건데요. 업무 테이블에 status 컬럼 추가하고, 배정 이벤트 발생하면 FCM 토큰으로 푸시 보내는 구조. 야간은 priority 플래그 추가하면 될 것 같아요."

현태가 고개를 끄덕였다. "심플하네요."

승현은 지연이 그린 다이어그램을 한참 봤다. 뭔가 이상한 느낌이 들었다. 틀린 건 아닌데.

이게 맞는 문제를 풀고 있는 건가?

승현이 입을 열었다.

"지연아, 잠깐. 우리가 뭘 고치려고 하는 건지 한번 정리해보자."

"팀원이 업무 놓치는 거요. 알림이 없어서."

"알림이 없어서 놓치는 건 맞아. 근데 그게 증상이야, 문제야?"

지연이 펜을 내려놓았다.

승현이 화이트보드를 지우고 세 줄을 그었다.

```
증상:        팀원이 업무를 놓친다
문제:        배정 후 알림이 없다
근본 원인:  배정 흐름이 시스템 밖에서 일어난다
```

"우리가 알림을 만들면 '문제'를 해결하는 거야. 근데 근본 원인은 남아있어. 팀장이 시스템에 입력하고, 또 슬랙으로 따로 연락하는 이중 작업. 이건 그대로야."

현태가 물었다. "그러면 근본 원인까지 해결해야 해요?"

"아니, 꼭 그럴 필요는 없어. 다음 달 데모면 알림으로 충분할 수 있어. 근데 그게 의식적인 결정이어야 해. '우리는 지금 문제 레이어를 고치고 있다. 근본 원인은 나중에' — 이게 결정이어야 한다는 거야."

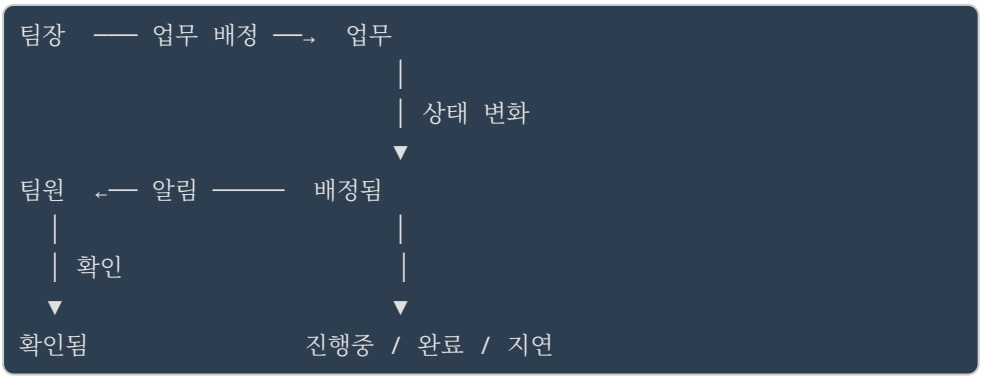
지연이 잠깐 생각하다가 말했다.

"그러면 나중에 또 비슷한 요청이 올 수 있겠네요."

"올 거야. 그리고 그때 대응할 수 있어야 해. 아니면 지금 근본 원인까지 같이 건드릴 건지 민준이랑 얘기해야 하고."

승현이 화이트보드를 다시 지우고, 이번엔 다른 것을 그리기 시작했다.

"자, 그전에 하나 더. 이 시스템에서 실제로 누가 뭘 하는지 한번 그려보자. 우리가 코드 짜기 전에."



현태가 보다가 말했다.

"오빠, '확인됨' 상태가 필요한가요? 그냥 알림 보내면 되지 않아요?"

지연이 끼어들었다.

"아, 근데 김 부장님이 '팀원이 확인했는지 알고 싶다'고 하셨잖아요. 기억 안 나요?"

현태가 고개를 갸웃했다. "그랬어요? 저는 그냥 알림인 줄 알았는데."

승현이 가만히 봤다.

지금 같은 대화를 했는데, 세 사람이 다르게 이해하고 있다.

"우리 지금 '배정됨'이랑 '확인됨'을 같은 걸로 생각하는 사람이 있어. 이 둘은 달라. '배정됨'은 팀장이 배정한 것, '확인됨'은 팀원이 봤다는 것. 이게 같은 거면 acknowledgement 기능이 필요 없고, 다른 거면 필요해."

현태가 작게 말했다.

"그거 DB 설계가 달라지는 거죠."

"맞아."

방이 조용해졌다. 지연이 처음 그렸던 다이어그램은 이미 낡은 것이 되어 있었다.

5.2. 개념 노트

5.2.1. 증상과 문제와 근본원인

의사는 환자가 "머리가 아파요"라고 하면 두통약을 바로 처방하지 않습니다. 왜 아픈지를 먼저 물어봅니다. 두통은 증상일 수 있기 때문입니다.

소프트웨어 개발에서도 같은 레이어 구분이 있습니다.

증상(Symptom)은 관찰 가능한 결과입니다. "팀원이 업무를 놓친다." 고통이 느껴지는 곳이지만, 이것을 고치는 것이 문제를 고치는 것은 아닙니다.

문제(Problem)는 증상의 직접 원인입니다. "배정 후 알림이 없다." 이것을 고치면 이 증상은 사라집니다. 하지만 더 깊은 원인이 남아있을 수 있습니다.

근본 원인(Root Cause)은 문제를 만들어내는 더 깊은 구조입니다. "배정 흐름이 시스템 밖에서 일어난다." 이것을 고치면 여러 증상이 동시에 사라질 수 있습니다.

어떤 레이어에서 해결할 것인가는 맥락에 따른 의식적 선택이어야 합니다. 다음 달 데모라면 문제 레이어 해결이 현실적입니다. 장기 제품 로드맵이라면 근본 원인까지 건드릴 수 있습니다. 문제는 — 많은 팀이 이것을 모르고 증상 레이어에서 해결합니다.



"어느 레이어를 고칠 것인가"는 기술적 결정이 아니라 **범위 결정**입니다. 이 선택을 팀이 명시적으로 공유하지 않으면, 서로 다른 레이어를 해결하고 있다고 착각하게 됩니다.

5.2.2. 같은 것을 다르게 프레임하면 다른 문제가 된다

"팀원이 업무를 놓친다"는 현상은, 어떻게 프레임하느냐에 따라 전혀 다른 문제가 됩니다.

알림 문제로 프레임하면 알림을 만들면 됩니다. 커뮤니케이션 문제로 프레임하면

프로세스를 바꿔야 합니다. 가시성 문제로 프레임하면 대시보드가 필요합니다. 책임 명확성 문제로 프레임하면 확인(acknowledgement) 기능이 필요합니다.

좋은 엔지니어는 여러 프레임이 가능하다는 것을 인식하고, 팀과 함께 어떤 프레임으로 접근할지 선택합니다.

5.2.3. Domain Model — 코드를 짜기 전에 세계를 그린다

Fairbanks는 소프트웨어를 둘러싼 세 가지 모델을 구분합니다.[1]

Domain Model은 현실 세계의 개념들입니다. 사람, 행위, 상태, 규칙, 사물. 이것은 코드가 아니라 현실에 대한 이해입니다.

Design Model은 만들 소프트웨어의 구조입니다. `ERD`, `API`, 시스템 다이어그램. 이것은 Solution Space의 시작점입니다.

Code Model은 실제 구현입니다.

대부분의 엔지니어는 Domain Model을 건너뛰고 바로 Design Model로 갑니다. 현태가 "배정됨"과 "확인됨"을 같은 것으로 이해한 것처럼 — Domain Model이 팀 안에서 정렬되지 않으면, 같은 회의를 해도 다른 것을 설계합니다.

Domain Model이 안정될 때를 어떻게 아는가. 팀이 같은 어휘로 이야기할 때입니다. "업무"가 무엇인지, "배정"과 "확인"이 무엇인지, 팀원 모두가 같은 것을 가리킬 때. 그 전까지는 아무리 정밀한 ERD를 그려도 기반이 흔들립니다.

5.3. 이 강에서 써볼 것

다음 번에 설계 회의를 시작하기 전에, 팀에게 이 질문을 해보세요.

"우리가 지금 증상을 고치려는 건가요, 문제를 고치려는 건가요?"

그리고 설계를 시작하기 전에 5분만 투자해서 도메인 어휘를 정렬해보세요.

"이 시스템에서 가장 중요한 개념 3개는 무엇인가요? 그것들이 가질 수 있는 상태는 무엇인가요?"

이 두 가지 질문이 나중에 몇 시간의 재작업을 막을 수 있습니다.

5.3.1. 레이어 선택이 범위를 결정한다

"어느 레이어에서 해결할 것인가"의 결정은 만들어야 할 것의 범위를 자연스럽게 결정합니다.

증상 레이어에서 해결하기로 했다면 — 근본 원인 레이어와 관련된 것들은 지금 만들 필요가 없습니다. 그것은 나중의 일입니다.

문제 레이어에서 해결하기로 했다면 — 그 결정이 이번 스프린트의 범위를 정의합니다. 증상을 만드는 다른 근본 원인들은 지금 다루지 않아도 됩니다.

Problem Space를 깊이 이해하는 것이 더 많은 것을 만들게 하는 것이 아닙니다. 오히려 — 지금 만들어야 할 것과 그렇지 않은 것을 구분하게 합니다. 그리고 지금 해야 할 것이 명확해질수록, 그것을 더 빨리 완성해서 사용자에게 먼저 전달할 수 있게 됩니다.

레이어 선택은 단순히 "무엇을 고치는가"의 결정이 아닙니다. "지금 무엇을 전달할 것인가"의 결정이기도 합니다.

다음 에피소드: 안개 속에서 걷는 법 — 해소할 수 없는 모호함 앞에서 Cunningham이 한 선택 '"

[1] George Fairbanks, Just Enough Software Architecture, 2010. Domain Model, Design Model, Code Model의 구분은 이 책 3장 "Model Kinds"에서 상세히 다룬다. 세 모델의 정렬 여부가 아키텍처 품질을 결정하는 핵심 요소로 제시된다.

Chapter 6. 안개 속에서 걷는 법

> "모든 안개가 걷히기를 기다리는 사람은 영원히 출발하지 못한다. 어떤 안개는 걸으면서 걷힌다."

6.1. 이야기

수요일 오전.

팀은 막혀 있었다.

기술 검토 회의가 두 시간째 이어지고 있었다. 칠판에는 알림 구조 다이어그램이 세 가지 버전으로 그려져 있었다. 지연의 버전, 현태의 버전, 그리고 둘이 합친 버전.

막힌 지점은 같았다.

"모바일 백그라운드에서 알림을 확실히 받을 수 있는가?"

지연이 말했다.

"`FCM`은 백그라운드 알림 지원하거든요. 안드로이드는 기본이고, `iOS`는 `APN` 설정만 하면 돼요."

현태가 반박했다.

"근데 공장 환경이잖아요. 와이파이가 끊어지거나 절전 모드가 강하게 걸려있으면 못 받을 수도 있어요. 저 그런 거 본 적 있어요."

"그건 일반적인 케이스가 아니잖아요."

"공장이 일반적인 환경이에요?"

승현이 둘을 봤다. 두 사람 모두 맞는 말을 하고 있었다. 그리고 둘 다 확신이 없었다.

이건 조사해서 풀리는 문제가 아니다.

FCM 문서는 읽었다. 스택 오버플로우도 읽었다. 하지만 "공장 현장에서 태블릿 절전 모드 상태로 야간에 푸시가 오는가?"라는 질문에 명확하게 답하는 자료는 없었다. 어떤 글은 된다고 하고, 어떤 글은 안 된다고 했다.

승현이 칠판을 지웠다.

"잠깐. 우리가 지금 뭘 하고 있는 건지 생각해보자."

둘이 쳐다봤다.

"우리가 이 질문에 답하려고 두 시간 동안 이야기하고 있는데, 이 질문은 이야기로 풀리는 게 아니야. 문서로도 안 풀려. 실제로 해봐야 알아."

현태가 물었다.

"그러면 어떻게 해요?"

"제일 단순한 걸 해보자. 테스트 앱 하나 만들어. 버릴 거야. FCM 설정만 넣고, 푸시 보내는 기능만. 그걸로 실제 태블릿에 테스트해보는 거야. 절전 모드 켜고, 와이파이 강도 낮추고. 실제 공장 환경을 최대한 흉내 내서."

"그거… 반나절이면 될 것 같은데요."

"그래. 반나절. 두 시간 더 이야기하는 것보다 빠르고 확실해."

지연이 조용히 물었다.

"근데 실패하면요? 푸시가 안 오면?"

승현이 대답했다.

"그것도 답이야. 그러면 우리가 FCM만으로는 안 된다는 걸 알게 되는 거고, 다른 방법을 찾으면 돼. 실패가 나쁜 게 아니야. 모르는 채로 계속 짜는 게 나쁜 거야."

현태가 일어서며 말했다.

"제가 할게요. 오늘 오후 안에 해볼 수 있을 것 같아요."

그날 저녁 여섯 시.

현태가 슬랙에 메시지를 보냈다.

> 현태: 결과 나왔어요.

> 와이파이 정상 상태, 절전 모드 없음 → 정상 수신

> 와이파이 약함, 절전 일반 → 지연되지만 수신

> 절전 강함 (배터리 절약 모드 최대) → 안 옴

> 와이파이 끊김 + 절전 → 완전히 안 옴

> 결론: `FCM`만으로는 야간 긴급 케이스 보장 못 함.

> 근데 일반 배정 알림은 괜찮을 것 같아요.

승현이 답장했다.

> 승현: 완벽해. 이제 알았다.

> 야간 긴급은 별도 처리 필요. 일반은 FCM으로 간다.

> 반나절에 두 시간짜리 논쟁을 끝냈네.

지연이 답했다.

> 지연: 근데 이거 버리는 코드 아닌가요?

> 현태: 네. 근데 이 코드가 없었으면 우리는 아직도 회의하고 있었을 것 같은데요.

6.2. 개념 노트

6.2.1. 두 가지 모르는 것

이 이야기에서 팀이 직면한 것은 두 종류의 모르는 것이 섞인 상황이었습니다.

`FCM`이 백그라운드 알림을 지원하는지 — 이것은 문서를 읽으면 알 수 있는 것입니다. Resolvable Ambiguity입니다.

하지만 공장 현장의 특정 절전 환경에서 실제로 알림이 도달하는지 — 이것은 해봐야 아는 것입니다. 문서는 일반적인 케이스를 다루지, 이 특수한 환경을 다루지 않습니다. Essential Ambiguity입니다.

팀은 두 시간 동안 Essential Ambiguity를 Resolvable처럼 다루었습니다. 답이 없는 질문을 계속 던졌습니다. 승현이 한 것은 이 둘을 구분하는 것이었습니다.

6.2.2. "가장 단순한 것"의 의미

Ward Cunningham이 던진 질문 — "What is the simplest thing that could possibly work?" — 은 종종 오해됩니다.[1]

이것은 "최소한만 만들어라"가 아닙니다.

정확한 의미는 이것입니다.

"이 모호함을 드러내줄 가장 작은 행동은 무엇인가?"

현태가 만든 테스트 앱은 프로덕션 코드가 아니었습니다. 버릴 것을 알고 만들었습니다. FCM 설정만, 푸시 기능만. 목적은 학습이었고, 반나절이면 충분했습니다.

이것이 Spike Solution[2]입니다. XP(Extreme Programming)에서 나온 실천으로, 특정 기술적 또는 설계적 모호함을 탐색하기 위한 버려지는 탐색 코드입니다. 산출물은 코드가 아니라 학습입니다.

지연의 질문 — "이거 버리는 코드 아닌가요?" — 에 대한 답은 현태가 이미 했습니다. 없었다면 두 시간짜리 논쟁이 계속됐을 것입니다.

6.2.3. Technical Debt — 오해된 개념

Ward Cunningham이 Technical Debt를 처음 이야기했을 때, 많은 사람들이 그 원래 의미를 놓쳤습니다.

그것은 나쁜 코드에 대한 비유가 아니었습니다.

Cunningham의 의도는 이것이었습니다.

지금 우리가 이해하는 것보다 더 정교한 설계가 필요하지만, 지금 당장 그것을 알 수 없습니다. 그래서 의도적으로 단순하게 만들고, 사용하면서 배우고, 배운 것을 바탕으로 리팩토링합니다. 현재 설계와 이상적인 설계의 차이가 '부채'입니다.

핵심은 의도적이라는 것입니다. 모르는 채로 만드는 것이 부채를 지는 것입니다. 모르고 아무렇게나 만드는 것은 부채가 아니라 그냥 나쁜 설계입니다.

팀이 야간 긴급 알림을 일단 FCM으로 처리하고 나중에 별도 처리를 추가하기로 했다면 — 이것은 의식적으로 부채를 지는 것입니다. "지금 이것이 완벽한 해결책이 아님을 알고 있다. 나중에 고친다." 이 선언이 부채를 관리 가능한 것으로 만듭니다.

6.2.4. 행동하면서 아는 것

철학자 Donald Schön은 전문가의 실천을 연구하면서, 가장 뛰어난 전문가들이 공통적으로 하는 것을 발견했습니다.[3]

그들은 행동 전에 완벽한 계획을 세우지 않았습니다. 행동하면서 생각하고, 행동의 결과로 문제를 더 깊이 이해했습니다. 그는 이것을 Reflection-in-Action이라고 불렀습니다.

승현이 두 시간의 회의를 중단하고 테스트를 시작한 것이 그것입니다. 더 많은 논의가 아니라, 가장 작은 행동으로 가장 빠른 학습을 선택한 것.

이것이 미숙함의 표시가 아닙니다. 복잡한 문제 앞에서의 성숙한 인식론입니다.

6.3. 이 강에서 써볼 것

다음 번에 팀이 같은 질문을 반복하며 논의가 제자리를 돌고 있다면, 이것을 해보세요.

첫째. 멈추고 물어보세요. "이것은 이야기해서 풀리는 문제인가, 해봐야 아는 문제인가?"

둘째. Essential Ambiguity라면, 물어보세요. "이것을 확인하기 위한 가장 단순한 행동이 무엇인가? 얼마나 걸리는가?"

셋째. 그 행동이 두 시간짜리 논의보다 빠르다면, 그냥 해보세요.

버릴 것을 만드는 것이 아닙니다. 배울 것을 만드는 것입니다.

다음 에피소드: 이 문제, 어디서 본 것 같은데 — 유사 추론으로 이미 누군가 풀어놓은 문제를 찾는 법 '"

[1] Ward Cunningham은 Extreme Programming(XP)의 핵심 실천들을 정립한 소프트웨어 엔지니어로, Technical Debt 개념과 Wiki의 발명자이기도 하다. 이 질문은 XP 커뮤니티에서 설계 단순성의 원칙으로 널리 인용된다.

[2] Ron Jeffries 등이 체계화한 XP(Extreme Programming)의 실천 중 하나. "Spike"는 못을 박는 동작에서 온 비유로, 불확실성을 빠르게 뚫어내는 탐색 코드를 가리킨다. Kent Beck, Extreme Programming Explained, 2000.

[3] Donald A. Schön, The Reflective Practitioner: How Professionals Think in Action, 1983. 건축가, 엔지니어, 의사 등 다양한 전문가의 실천을 관찰하여 "행동 중 반성(Reflection-in-Action)"이 전문성의 핵심임을 주장한다.

Chapter 7. 이 문제, 어디서 본 것 같은데

> "처음 보는 문제는 없다. 처음 보는 포장지가 있을 뿐이다."

7.1. 이야기

목요일 점심.

팀은 식당 테이블에 앉아 있었다. 야간 긴급 알림을 어떻게 처리할지 아직 결론이 나지 않은 상태였다. Spike로 FCM만으로는 불충분하다는 것은 알았다. 그러면 어떻게 해야 하는가.

지연이 라면 국물을 젓다가 말했다.

"야간 긴급 알림… 결국 사람이 반드시 받아야 하는 거잖아요. 놓치면 라인이 멈추니까."

현태가 말했다.

"그거 PagerDuty 같은 거 아닌가요? 온콜 알림."

승현이 숟가락을 놓았다.

"맞아. 잠깐, PagerDuty가 이 문제를 어떻게 푸는지 생각해봐."

지연이 폰을 꺼냈다.

"PagerDuty는… 알림을 보내고 일정 시간 안에 확인 안 하면 다음 사람한테 에스컬레이션해요. 그리고 결국엔 전화도 해요."

승현이 고개를 끄덕였다.

"그게 핵심이야. 확인 없으면 재시도. 재시도 채널을 바꾸면서. 이게 `TCP` 재전송이랑 구조가 같아."

현태가 눈을 떴다.

"`TCP`… 패킷 안 오면 재전송하는 거요? 그게 알림이랑 같아요?"

"구조가 같다는 거야. 채널이 뭐냐는 다르지만 — 전달 확인이 없으면 재시도하고, 재시도 간격을 점점 늘리고, 최후 수단은 다른 채널로 간다. 이 원리."

지연이 정리했다.

"그러면 우리도. 앱 푸시 보내고 — 5분 안에 확인 없으면 — 다시 보내고 — 야간 긴급이면 `SMS`도 보내고."

"비슷한 방향이야. 근데 팀원이 열두 명이면 에스컬레이션 대상이 한 명인 PagerDuty랑 달라. 우리는 배정된 한 명이 못 받으면 어떻게 하지? 팀장한테 역으로 알려야 하나?"

현태가 생각하다가 말했다.

"Visibility 문제네요. 팀장이 팀원이 받았는지 모르는 거."

승현이 처음으로 웃었다.

"맞아, 현태야. 그게 이 문제의 두 번째 구조야."

식당을 나서면서 지연이 말했다.

"오빠, 근데 우리 알림 기능 만드는 거 맞죠? TCP 재전송 구현하는 거 아니고."

승현이 걸으며 대답했다.

"당연하지. TCP를 갖다 쓰는 게 아니라, 그 원리를 쓰는 거야. '전달 확인 없으면 재시도'라는 논리. 구현은 완전히 우리 방식으로 해."

그날 오후.

승현은 슬랙에 카카오워크와 네이버 웍스 링크를 올렸다.

> 승현: 우리랑 비슷한 포지션의 서비스들. 기능 목록 말고,

```
이 서비스들이 어떤 Trade-off를 선택했는지 봐보자.
```

```
카카오워크는 뭘 포기하고 뭘 얻었는가.
```

```
네이버 웍스는 어떤 사용자에게 최적화되어 있는가.
```

```
한 시간만 각자 써보고 2-3문장으로 정리해와.
```

한 시간 후, 지연이 정리를 올렸다.

> 지연: 카카오워크는 카카오 생태계 연동에 올인함.

```
알림 커스터마이징이 거의 없음.
```

```
카카오 계정 있으면 편하지만 없으면 불편함.
```

```
→ 우리 고객사처럼 기업 계정만 쓰는 환경에는 안 맞을 수 있음.
```

>

```
네이버 웍스는 그룹웨어 기능이 강함.
```

```
결재 라인이 있는 기업 구조에 맞춰져 있음.
```

```
근데 알림이 너무 많아서 무시하게 됨 (내 경험).
```

```
→ 알림 피로 문제를 해결 못 함.
```

승현이 답했다.

> 승현: 좋아. 그러면 우리가 차별화할 수 있는 지점이 보이지?

```
"현장 근무자에게 꼭 필요한 알림만, 확실하게."
```

```
이게 우리 설계 방향이야.
```

7.2. 개념 노트

7.2.1. Polya의 질문

조지 폴리아는 수학 문제 풀이의 방법을 연구하면서 책 How to Solve It [1]을 썼습니다. 그가 제시한 가장 강력한 질문 중 하나는 이것입니다.

"Do you know a related problem?"

수학 문제에서 출발한 이 질문은, 소프트웨어 설계에서도 그대로 유효합니다.

모든 문제를 처음부터 풀 필요가 없습니다. 이 문제의 구조가 이미 알려진 문제와 닮았다면, 그 해결책의 논리를 가져올 수 있습니다. 이것이 유사 추론입니다.

승현이 한 것이 그것입니다. 알림 문제를 TCP 재전송의 논리 구조와 연결했습니다. PagerDuty의 에스컬레이션 패턴과 연결했습니다. 도메인은 완전히 달랐지만, 전달 확인이 없으면 재시도하는 구조는 같았습니다.

7.2.2. 유사 추론의 세 단계

유사 추론은 세 단계로 이루어집니다.

첫째, 구조를 추출합니다. 이 문제에서 도메인 특수성을 벗겨내면 무엇이 남는가. 팀오더에서 "팀원", "알림", "공장"을 벗겨내면 — "수신자의 상태가 불확실한 환경에서 중요도가 다른 메시지를 확실히 전달하고 확인해야 한다"가 남습니다.

둘째, 유사 문제를 탐색합니다. 이 구조와 닮은 문제가 다른 도메인에 있는가. TCP 재전송, PagerDuty 에스컬레이션, 병원 트리아지, 항공 관제 우선순위 통신.

셋째, 원리를 역수입합니다. 유사 문제의 해결책이 아니라 원리를. TCP를 구현하는 것이 아니라, "확인 없으면 채널을 바꿔서 재시도"라는 논리를.

7.2.3. 환원 — 이건 어떤 종류의 문제인가

때로는 유사 사례를 구체적으로 찾는 것보다, 문제 유형 언어로 환원하는 것이 더 빠릅니다.

"이건 Visibility 문제다"라고 말하는 순간, 팀은 Visibility 문제에서 알려진 해결 패턴들을 참고할 수 있습니다. 상태 공유, 실시간 동기화, 확인 요청. 각 패턴이 어떤 Trade-off를 가지는지도 이미 알려져 있습니다.

현태가 "Visibility 문제네요"라고 말한 것은 단순한 관찰이 아니었습니다. 그 한 마디가 이미 가능한 해결 방향들을 열어주었습니다.

7.2.4. 기능 목록이 아니라 Trade-off를 읽는다

비슷한 서비스를 조사할 때, 기능 목록을 만드는 것은 별로 도움이 안 됩니다. 어떤 기능이 있다는 것을 알아도, 왜 그 설계를 선택했는지는 모릅니다.

Cunningham의 관점에서 보면, 이미 출시된 서비스는 수많은 Spike와 Essential Ambiguity 탐색의 결과입니다. 그들이 선택한 Trade-off 안에 그 탐색의 흔적이 있습니다.

네이버 웍스의 알림이 너무 많아서 무시하게 된다는 지연의 관찰 — 그것이 Trade-off 읽기입니다. 그들이 알림 빈도보다 완결성을 선택했다는 것. 그리고 그 결과로 알림 피로가 생겼다는 것. 우리는 다른 선택을 할 수 있습니다.

7.2.5. 경험이 쌓이는 방식

3-4년차와 5-6년차의 차이를 만드는 능력 중 하나가 유사 추론입니다.

경험이 쌓인다는 것은 단순히 더 많은 것을 해봤다는 것이 아닙니다. 이미 해결된 구조들이 새 문제에서 보이기 시작하는 것입니다. 새 문제를 받았을 때, "이건 어디서 본 구조인데"가 자동으로 떠오를 때 — 그것이 경험이 지식 자산이 되는 방식입니다.

이 능력은 훈련할 수 있습니다. 문제를 볼 때 마다 한 번씩 물어보는 것으로 시작합니다.



유사 추론은 "기억 회상"이 아니라 "구조 인식"입니다. `PagerDuty`를 알아야 에스컬레이션 패턴을 쓸 수 있는 게 아닙니다. "확인 없으면 재시도"라는 구조를 인식하는 능력이 핵심입니다. 도메인을 가리고 보면 구조가 보입니다.

"이건 어떤 종류의 문제인가? 이 구조를 어디서 본 적 있는가?"

7.3. 이 강에서 써볼 것

다음 번에 새 요구사항을 받았을 때, 설계를 시작하기 전에 5분만 투자해보세요.

첫째. 이 문제의 핵심 구조를 한 문장으로 추출해보세요. 도메인 특수 용어를 최대한 빼고.

둘째. "이 구조, 어디서 본 것 같은데?"를 물어보세요. 다른 서비스, 다른 기술, 다른 도메인에서.

셋째. 찾은 유사 사례에서 해결책이 아니라 원리를 가져오세요. "그들은 이 문제를 어떤 논리로 풀었는가?"

이 5분이 나중에 설계 방향의 발산을 줄여줄 수 있습니다.

다음 에피소드: 어디에 공을 들여야 하는가 — 리스크 기준으로 설계 에너지를 배분하는 법 '''

[1] George Pólya, How to Solve It, Princeton University Press, 1945. 수학 교육의 고전으로, 문제 해결의 4단계(이해→계획→실행→검토)와 함께 "유사 문제 탐색", "일반화", "특수화" 등 구체적인 사고 전략을 제시한다.

Chapter 8. 어디에 공을 들여야 하는가

> "모든 곳에 공을 들이면, 정작 중요한 곳에 공이 없다."

8.1. 이야기

월요일 오전.

설계 리뷰 시간이었다. 지연이 지난 주에 작업한 아키텍처 다이어그램을 화면에 띄웠다.

컴포넌트 다이어그램, 시퀀스 다이어그램, ERD, API 명세 초안. 열심히 그렸다는 것이 한눈에 보였다.

현태가 말했다.

"와, 상세하다."

지연이 조심스럽게 물었다.

"어떤가요, 오빠?"

승현은 한참 화면을 봤다. 뭔가 이상한 느낌이 들었다. 틀린 것은 아니었다. 하지만.

"지연아, 이거 그리는 데 얼마나 걸렸어?"

"… 이틀이요."

"알림 전달 보장 — FCM 실패했을 때의 fallback — 이 부분이 여기 어디 있어?"

지연이 다이어그램을 이리저리 가리켰다.

"여기… 아, 이건 아직 없네요."

"야간 긴급 알림이 배달되지 않았을 때 팀장한테 알려주는 부분은?"

"그것도 아직…"

"acknowledgement 타임아웃 처리는?"

지연이 조용해졌다.

승현이 부드럽게 말했다.

"이 다이어그램 잘 그렸어. 근데 우리가 가장 걱정해야 할 부분들이 빠져있어. 그리고 컴포넌트 구조 — 이건 지금 리스크가 낮아. 어떻게 나눌지는 나중에 바꿀 수 있거든. 근데 알림 전달 보장은 나중에 바꾸면 비싸."

현태가 끼어들었다.

"저도 그거 궁금했어요. 푸시가 안 가면 어떻게 해요?"

승현이 화이트보드에 작게 적었다.

```
Engineering Risk 목록:
1. 알림 미전달 (네트워크/절전)        → Reliability 리스크
2. 야간 긴급 확인 실패                 → Reliability 리스크
3. acknowledgement 없을 때 처리       → Reliability 리스크
4. 팀원 15명→150명 확장 시 성능       → Performance 리스크
5. 알림 채널 추가 시 코드 변경 범위  → Modifiability 리스크
```

"이 다섯 개 중에 지금 당장 설계 에너지를 써야 하는 게 뭔지 보여?"

지연이 천천히 읽었다.

"1, 2, 3은 데모에서도 중요해요. 라인 스탑 나면 데모가 끝나니까."

"맞아."

"4번은… 데모는 15명이라 괜찮은 거 아닌가요?"

"맞아. 지금은 리스크 낮아. 근데 나중에 150명 될 때 다시 봐야 해. 지금은 안 해도 돼."

"5번은?"

승현이 잠깐 생각했다.

"이건 중간이야. 지금 SMS 채널 추가하는 거 고려하고 있잖아. 그러면 채널 추상화를 지금 어느 정도 해두는 게 맞아. 근데 완벽하게 할 필요 없어 — SMS 추가할 때 비용이 크지 않을 정도로만."

지연이 노트에 적으며 말했다.

"그러면 제가 이틀 동안 그린 것 중에 지금 필요한 건…"

"1, 2, 3번을 어떻게 처리할지. 그리고 5번의 채널 추상화 수준. 이게 다야. 나머지는 나중에."

현태가 물었다.

"그럼 나머지 다이어그램은요?"

승현이 웃었다.

"버려. 지금은. 나중에 필요해지면 그때 그려."

지연이 조용히 말했다.

"이틀을…"

"그 이틀이 헛된 게 아니야. 전체 구조를 머릿속에 갖게 됐잖아. 근데 다음부터는 — 그릴 때마다 물어봐. 이게 어떤 리스크를 낮추기 위한 건지."

그날 오후.

지연은 세 가지에만 집중했다.

첫째, `FCM` 전달 실패 시 재시도 로직. 몇 번, 간격은 어떻게. 둘째, 야간 긴급 플래그가 있을 때 `SMS` fallback 트리거 조건. 셋째, 알림 채널을 추상화하는 인터페이스 — 앱 푸시와 `SMS`를 같은 인터페이스로.

저녁 여섯 시에 슬랙을 보냈다.

> 지연: 오늘 세 가지 완료했어요.

```
이틀치를 하루에 했네요.
```

8.2. 개념 노트

8.2.1. 기능과 품질 속성은 다르게 쌓인다

소프트웨어에는 두 종류의 속성이 있습니다.

기능(Functionality)은 "시스템이 무엇을 하는가"입니다. 알림을 보낸다, 업무를 배정한다. 이것은 스프린트마다 쌓을 수 있고, 나중에 추가하기 상대적으로 쉽습니다.

품질 속성(Quality Attributes)은 "시스템이 얼마나 잘 하는가"입니다. 신뢰성, 성능, 보안, 변경용이성. 이것은 아키텍처 결정으로 확보됩니다. 나중에 추가하려면 시스템

전체를 건드려야 할 수 있습니다.

지연이 이틀 동안 그린 다이어그램은 기능 구조를 상세히 보여주었지만, 가장 중요한 품질 속성 — 알림 전달 신뢰성 — 을 다루지 않았습니다. 컴포넌트 구조는 나중에 바꿀 수 있습니다. 하지만 알림 전달 실패 처리는 처음부터 고려하지 않으면 나중에 비쌉니다.

8.2.2. Risk-Driven Model — 리스크에 비례해서 설계한다

George Fairbanks는 소프트웨어 아키텍처에서 가장 자주 받는 질문에 답했습니다.[1]

"얼마나 설계해야 하는가?"

그의 답: 리스크에 비례해서. 리스크가 높은 곳은 깊이 설계하고, 리스크가 낮은 곳은 설계를 최소화하거나 미룬다.

이것은 게으름이 아닙니다. 한정된 시간 안에 가장 중요한 것에 집중하는 것입니다.

승현이 한 것이 그것입니다. 다섯 개의 Engineering Risk를 나열하고, 지금 당장 중요한 것(1, 2, 3)과 나중에 봐도 되는 것(4), 중간 수준(5)을 구분했습니다. 지연은 그 결과로 하루 만에 이틀치 작업을 끝냈습니다.

8.2.3. Engineering Risk와 PM Risk는 다른 도구로 해소한다

중요한 구분이 있습니다.

Engineering Risk는 시스템 분석, 설계, 구현에서 오는 기술적 실패 가능성입니다. 알림이 전달되지 않는다, 데이터 일관성이 깨진다, 보안 취약점이 있다. 이것은 엔지니어링 기법으로 해소합니다. Spike, 프로토타입, 부하 테스트, 아키텍처 리뷰.

Project Management Risk는 일정, 팀, 우선순위의 문제입니다. 1개월 안에 못 끝낼 것 같다, 팀원이 이 기술을 모른다. 이것은 PM 기법으로 해소합니다. 일정 조정, 교육, 범위 축소.

핵심: 리스크 유형과 기법 유형이 맞아야 합니다. 성능 리스크에 Gantt 차트를 그리는 것은 무효입니다. 팀 일정 문제에 아키텍처 다이어그램을 그리는 것도 무효입니다.

8.2.4. "지금 안 해도 된다"는 결정

Fairbanks의 Risk-Driven Model에서 가장 실용적인 통찰은 이것입니다.

리스크가 낮은 것은 지금 설계하지 않아도 됩니다. 그리고 그것은 "영원히 안 한다"가 아닙니다. "리스크가 커지는 시점에 다시 본다"입니다.

현재 사용자가 15명인 시스템의 확장성을 지금 깊이 설계하는 것은 낭비입니다. 150명이 될 것 같은 신호가 보일 때, 그 시점에 투자하는 것이 적절합니다.

이 판단이 어려운 이유는 — 엔지니어는 완벽하게 설계하고 싶은 본능이 있기 때문입니다. Risk-Driven Model은 그 본능에 기준을 줍니다. "이 설계가 어떤 리스크를 낮추는가?"라는 질문이 그 기준입니다.

8.3. 이 강에서 써볼 것

다음 번에 설계를 시작하기 전에, 이 질문을 먼저 해보세요.

"이 시스템에서 실패 가능성이 가장 높은 곳은 어디인가?"

그 답이 나오면 — 그곳에 설계 에너지를 집중하세요. 그 외의 곳은 일단 단순하게 두고, 리스크가 커지면 그때 다시 보세요.

설계 회의에서 누군가 긴 다이어그램을 가져오면, 물어보세요.

"이 다이어그램이 어떤 리스크를 낮추기 위한 건가요?"

답이 나오면 좋은 설계입니다. 답이 안 나오면 — 지금 당장 필요한 것인지 다시 생각해볼 시간입니다.

다음 에피소드: 승현의 지도 — 해결책을 원래의 문제에 대입하고, 8강에 걸쳐 쌓아온 것들을 통합하는 마지막 강 '"

[1] George Fairbanks, Just Enough Software Architecture, 2010. "Risk-Driven Model"은 이 책의 핵심 주장으로, 아키텍처 설계 노력을 리스크 크기에 비례해서 배분해야 한다는 원칙이다. 과도한 설계와 과소한 설계 모두 낭비라는 관점에서 "딱 필요한 만큼"을 찾는 실용적 접근을 제시한다.

Chapter 9. 승현의 지도

> "지도는 영토가 아니다. 하지만 좋은 지도는 어디로 가야 하는지, 어디로 가지 않아도 되는지를 동시에 알려준다."

9.1. 이야기

데모 전날 밤.

승현은 화이트보드 앞에 혼자 서 있었다.

지난 한 달을 돌아봤다. 민준의 한 문장에서 시작했다. 팀오더의 알림 기능. 처음 들었을 때 15초 만에 WebSocket을 생각했던 것도 기억났다.

지금 만들어진 것은 세 가지였다.

FCM 앱 푸시와 수신 확인. 야간 긴급 SMS 채널. 알림 채널 인터페이스.

그게 전부였다.

처음에 떠올렸던 것들과 비교하면 훨씬 적었다. 실시간 대시보드도 없었다. 알림 히스토리 페이지도 없었다. `Kafka`도 없었다. 지연이 써보고 싶다던 `Kafka`는 그 어디에도 없었다.

이게 맞는가?

승현은 화이트보드에 적기 시작했다.

```
처음 민준의 말에서 파악한 것:
  "팀원이 업무를 놓쳐서 라인이 멈춘다"
  "팀장이 배정하고 따로 슬랙으로 연락해야 한다"
  "현장에서 태블릿/폰 사용. 야간 조 있음"

만든 것:
  FCM + acknowledgement → "따로 연락" 없애고, "놓침" 줄임
  SMS fallback → 야간 긴급 케이스 커버
  채널 추상화 → 나중에 채널 추가 시 변경 최소화

만들지 않은 것 + 이유:
  알림 히스토리 → 팀장이 원한 건 "지금 받았는가"지, 과거 로그가 아님
```

```
실시간 대시보드 → 데모 범위 밖, 리스크 낮음
`Kafka` → 15명 팀에 과한 인프라, 지금 리스크 없음
자동 보고서 → 요구사항에 없었음. FAC에서 Function이 아니었음
```

적고 나서 승현은 오래 바라봤다.

만들지 않은 것들이 왜 만들지 않아도 됐는지가 보인다.

알림 히스토리를 만들지 않은 것은 게으름이 아니었다. 팀장이 원한 것이 무엇인지 — 3강에서 Context-Free Questions로 캐낸 것, "팀원이 확인했는지 알고 싶다"는 것 — 를 알았기 때문이었다. 실시간 과거 로그가 아니라 현재 수신 여부.

Kafka가 없는 것은 기술 선택의 문제가 아니었다. 지금 15명 팀에서 리스크가 없다는 것을 — 7강 리스크 체크로 확인했기 때문이었다.

자동 보고서가 없는 것은 — 2강에서 FAC를 채울 때 Functions 목록에 없었기 때문이었다.

문제를 이해했기 때문에 만들지 않아도 되는 것이 보였다.

그리고 만들지 않은 것들이 많아졌기 때문에 — 실제로 필요한 세 가지에 집중할 수 있었다. 한 달 안에 완성될 수 있었다.

승현은 마커를 내려놨다.

내일 데모에서 팀장에게 이것을 보여줄 것이다. 세 가지만. 하지만 그 세 가지가 팀장이 말한 "라인 스탑을 막고 싶다"는 것과 정확하게 연결된다는 것을 승현은 알고 있었다.

데모 다음 날.

슬랙에 민준의 메시지가 왔다.

> 민준: 야 대박. 팀장님이 이거 바로 도입하고 싶다고 했어.

> `"딱 필요한 것만 있다"고.`

지연이 답했다.

> 지연: 진짜요?? 저 처음엔 기능이 너무 적은 거 아닌가 걱정했는데.

승현이 답했다.

> 승현: "딱 필요한 것만"이 칭찬이야.

```
우리가 뭐가 필요한지 알았다는 거야.
```

현태가 마지막으로 올렸다.

> 현태: 근데 Kafka는 언제 써요?

9.2. 개념 노트

9.2.1. 문제를 이해하면 해결책이 단순해진다

이것이 이 워크샵 전체를 관통하는 하나의 명제입니다.



Problem Space를 충분히 탐색하면 Solution Space가 **더 넓어지는 것이 아니라 더 좁아집니다.** 만들 것이 많아지는 것이 아니라, 만들지 않아도 되는 것이 보입니다.

처음에는 직관에 반하는 것처럼 보일 수 있습니다. 문제를 더 깊이 이해하려면 더 많은 시간이 필요하고, 그러면 Solution Space에 진입하는 것이 늦어질 것 같습니다.

그러나 실제로는 반대입니다.

Problem Space를 충분히 탐색하면 세 가지가 명확해집니다.

첫째, 핵심적으로 달성해야 할 것이 보입니다. 팀장이 원한 것은 "팀원이 업무를 받았음을 확인하는 것"이었습니다. 그것이 보이면 — 그것이 아닌 모든 것이 핵심이 아니게 됩니다.

둘째, 나중에 해도 되는 것이 보입니다. 리스크가 낮은 것, 지금 당장 사용자에게 필요하지 않은 것. 이것을 지금 만들지 않는 것은 게으름이 아니라 판단입니다.

셋째, 하지 않아도 되는 것이 식별됩니다. FAC에서 Function이 아니었던 것, Domain Model에 등장하지 않는 개념, Pre-mortem에서 실패 원인이 아닌 것. 이것들이 명확해지면 범위가 자연스럽게 줄어듭니다.

이 세 가지가 보일 때 — 만드는 것이 간단해집니다. 그리고 간단한 것을 먼저 만들어서 사용자에게 먼저 전달할 수 있습니다.

팀오더의 세 가지 기능이 한 달 만에 데모로 나올 수 있었던 것은, 나머지를 게으르게 건너뛴 것이 아닙니다. 문제를 충분히 이해했기 때문에 나머지를 만들지 않아도 된다는 것을 알았습니다.

9.2.2. Problem-Solution Fit

해결책을 만들면서 또는 만든 후에 반드시 확인해야 할 것이 있습니다.

내가 정의한 문제와 내가 만든 해결책이 실제로 연결되어 있는가.

이것이 Problem-Solution Fit입니다.

확인 방법은 간단합니다. 이 해결책이 없었다면 사용자는 어떻게 했을 것인가. 임시방편이 그대로 남아있다면 — 핵심을 놓친 것입니다. 이 해결책이 있어도 문제가 여전히 발생할 수 있다면 — 핵심이 빠진 것입니다.

승현이 화이트보드에 "만든 것"과 "처음 문제에서 파악한 것"을 나란히 적어서 대조한 것이 바로 이 체크였습니다.

9.2.3. Pre-mortem — 성공 편향 없이 약점 찾기

Pre-mortem은 Gary Klein이 제안한 기법입니다.[1] "6개월 후에 실패했다고 가정하면, 이유는 무엇인가?"라고 묻습니다.

일반적인 리뷰가 "이게 왜 잘 될 것인가"를 찾는다면, Pre-mortem은 "이게 왜 실패할 것인가"를 찾습니다. 성공 편향이 없어서 약점이 더 자연스럽게 나옵니다.

Pre-mortem에서 나온 실패 이유들은 — 대부분 Problem Space에서 이미 발견했어야 할 것들입니다. Constraint를 몰랐거나(2강), Essential Ambiguity를 Spike하지 않았거나(5강), 리스크를 과소평가했거나(7강). 그래서 Pre-mortem은 마지막 점검인 동시에 — 이 워크샵에서 배운 도구들이 실제로 작동했는지를 확인하는 방법이기도 합니다.

9.3. 이 강에서 가지고 가는 것

워크샵이 끝납니다.

도구들을 가지고 갑니다. FAC, Context-Free Questions, Black Box, 레이어 분석, Domain Model, Resolvable vs Essential, Spike, 유사 추론, Risk-Driven 설계.

그러나 이 도구들을 항상 꺼낼 필요는 없습니다.

이 모든 도구가 가리키는 하나의 습관이 있습니다.

요구사항을 받으면 — Solution Space로 넘어가기 전에, 딱 한 번 멈추고 물어보는 것.

"나는 지금 무엇을 알고 있는가? 그리고 무엇을 아직 모르는가?"

그 질문이 Problem Space를 열어줍니다. 그 탐색이 해결책을 단순하게 만들어 줍니다. 그 단순함이 사용자에게 더 빨리 전달됩니다.

에필로그: 지도를 그리는 법을 배운 사람 — 6개월 후의 팀오더, 그리고 이 워크샵이 남긴 것 '"

[1] Gary Klein, "Performing a Project Premortem", Harvard Business Review, 2007. Klein은 의사결정 심리학자로, 전문가들이 직관적 판단을 내리는 방식을 연구했다. Pre-mortem은 계획이 완료되기 전에 미래의 실패를 상상함으로써 성공 편향을 제거하는 기법이다.

Chapter 10. 지도를 그리는 법을 배운 사람

> "선생은 길을 가르쳐준다. 스승은 길을 찾는 법을 가르쳐준다."

10.1. 이야기

팀오더의 알림 기능은 데모를 넘어 실서비스에 반영됐다.

3개월이 지나면서 예상한 일들이 생겼다. 팀장 두 명이 알림을 너무 많이 받는다며 설정을 줄여달라고 했다. Pre-mortem에서 나왔던 "알림 피로" 실패 시나리오가 현실이 된 것이다. 팀은 이미 그것을 예상했기 때문에, 우선순위 필터링 기능을 1주일 만에 추가했다.

6개월이 지나면서 사용자가 150명으로 늘었다. 7강에서 "지금은 리스크 낮다"고 미뤄둔 확장성 설계를 꺼낼 시점이 됐다. 팀은 이것도 예상했기 때문에, 당황하지 않고 재검토 일정을 잡았다.

지연이 어느 날 슬랙에 올렸다.

> 지연: 근데 저 요즘 요구사항 받으면 예전이랑 다르게 반응해요.

```
예전엔 바로 DB 설계부터 생각했는데
```

```
지금은 "이게 어떤 문제야?"가 먼저 나오더라고요.
```

현태가 답했다.

> 현태: 저도요. 근데 그러면 속도가 느려지는 거 아닌가요?

```
오빠는 어떻게 생각해요?
```

승현이 답했다.

> 승현: 처음엔 느린 것 같아. 근데 재작업이 줄었잖아.

```
재작업 안 하는 게 더 빠른 거야 결국.
```

민준이 나중에 끼어들었다.

> 민준: 야 나도 이제 요구사항 가져올 때

```
"이게 왜 필요한지"를 먼저 생각하고 오게 됐어.
```

```
나도 모르게 배운 것 같은데.
```

10.2. 마무리 — 독자에게

이 워크샵은 기술을 가르치지 않았습니다.

FAC, Context-Free Questions, Black Box, Domain Model, Spike, 유사 추론, Risk-Driven. 이것들은 모두 도구입니다. 도구는 언젠가 쓸모가 달라집니다. 더 좋은 도구가 나옵니다.

하지만 도구 뒤에 있는 것은 남습니다.

문제를 먼저 이해하려는 의지. 모호함을 불편해하지 않고 탐색하려는 인내. 무엇을 만들지 않아도 되는지를 알아내는 능력. 그리고 — 빠르게 움직이는 것보다 올바른 방향으로 움직이는 것이 결국 더 빠르다는 믿음.

이것은 3-4년의 경험으로는 자동으로 생기지 않습니다. 의식적으로 훈련해야 합니다. 이 워크샵은 그 훈련의 시작이었습니다.

10.3. 이 워크샵에서 배운 것들

8강에 걸쳐 다룬 도구와 개념들을 마지막으로 정리합니다.

Problem Space 탐색

강	도구	핵심 질문
1강	Einstellung Effect, Presumptive Architecture 인식	나는 문제를 이해한 건가, 패턴을 인식한 건가?
2강	FAC (Functions / Attributes / Constraints)	요구사항의 세 층을 모두 캐냈는가?
3강	Context-Free Questions, Black Box Test	해결책을 전제하지 않고 탐색했는가?
4강	증상-문제-근본원인 레이어, Domain Model	어느 레이어를 해결할 것인가를 의식적으로 선택했는가?

모호함 다루기

강	도구	핵심 질문
5강	Resolvable vs Essential 진단, Spike	이 모호함은 질문으로 풀리는가, 행동으로만 아는가?

Solution Space 탐색

강	도구	핵심 질문
6강	유사 추론, 문제 유형 환원, Trade-off 리서치	이 구조, 어디서 본 적 있는가?
7강	Engineering Risk 식별, Quality Attributes, Risk-Driven 설계	어디에 설계 에너지를 집중해야 하는가?
8강	Problem-Solution Fit, Pre-mortem	내가 정의한 문제와 내가 만든 해결책이 연결되어 있는가?

10.4. 그래서, 무엇이 달라지는가

이 워크샵 전에:

```
요구사항 수신
    ↓ (15초)
Solution Space 진입
    ↓
설계 + 구현
    ↓
"이게 제가 원한 게 아니에요"
    ↓
재작업
```

이 워크샵 후:

```
요구사항 수신
    ↓
"나는 지금 무엇을 알고 있는가?"
    ↓
Problem Space 탐색 (FAC, Context-Free Q, Domain Model...)
    ↓
핵심 파악 + 나중에 해도 되는 것 + 하지 않아도 되는 것 식별
    ↓
집중해서 만들기
    ↓
일찍 전달 → 피드백 → 검증된 확장
```

재작업이 줄어드는 것이 목표가 아닙니다. 재작업이 줄어드는 것은 부산물입니다. 목표는 — 사용자가 필요한 것을 더 빨리 손에 쥐게 하는 것입니다.

승현이 배운 것을 한 줄로 쓴다면:

"문제를 오래 보는 것이 해결책을 빠르게 만드는 방법이다."

여러분이 다음 번에 요구사항을 받을 때, 그 한 줄이 생각난다면 — 이 워크샵은 성공한 것입니다.