

Don't Go Too Fast

압박 속에서도 품질을 지키는 소프트웨어 개발 철학

김백면

Third Edition: 2026

Table of Contents

DON'T GO TOO FAST.....	8
목차	오류! 책갈피가 정의되어 있지 않습니다.
서문: 왜 이 책을 썼는가.....	9
반복되는 패턴.....	9
나를 괴롭힌 질문	9
세 가지 발견.....	10
이론에서 실천으로	10
내가 직접 살아온 증거	11
왜 지금인가?.....	12
이 책이 해결하지 않는 것.....	12
이 책이 제공하는 것	13
누구를 위한 책인가	13
개인적인 말.....	13
Part 1: 핵심 개념과 가치	15
문제: What 이 아니라 How.....	15
이론적 기반.....	16
Daniel Kahneman: 왜 압박에서 실패하는가	16
Donella Meadows: 왜 전체를 봐야 하는가.....	19

Genrich Altshuller: 왜 타협이 아닌 해결이어야 하는가	22
왜 이 세 가지가 함께인가.....	25
ROD: Responsibility-Oriented Design	26
핵심 원칙.....	26
비대칭성: 제거 vs 추가.....	26
서비스 체인이란?.....	27
ROD 는 새로운 “What”이 아니다	28
진짜 힘: 변경 격리	28
엄격한 규칙	29
ROD 가 완전할 때	29
TFD: Test-First Development	30
핵심 원칙.....	30
왜 요구사항이 테스트여야 하는가.....	30
테스트는 완료 기준이다	31
TFD 와 ROD 의 결합	31
TDD 와의 관계	32
테스트는 즉각적 피드백을 제공한다	33
TFD 가 완전할 때	33
DGTF: Don’t Go Too Fast.....	34
핵심 원칙	34
전제조건: “Wow” 순간	34

LA 비유.....	35
왜 작동하는가	36
DGTF ≠ 느림, DGTF ≠ 의지력	36
DGTF 가 해결하지 않는 것	37
어떻게 함께 작동하는가.....	38
본질: 지식이 아니라 습관	39
이해 ≠ 실천	39
학습이 아니라 훈련.....	40
허들을 넘는 게 끝이 아니다	40
운전 비유.....	40
가장 어려운 부분	41
핵심 가치	41
1. 견고한 이론적 기반	41
2. 인간을 이해한다.....	42
3. 보편적으로 적용 가능	42
4. 상호 보완적	42
5. 지속 가능성	42
시작하기	42
Part 2: 실전 가이드	44
이 가이드 사용법	44

빠른 참조: 이론 요약	44
Kahneman (언제 생각할 것인가)	44
Meadows (무엇을 생각할 것인가)	44
TRIZ (모순을 어떻게 해결할 것인가)	44
Part 1: ROD 실전	45
규칙들	45
서비스 체인 만드는 법	47
예제: 결제 시스템	49
ROD 체크리스트	50
Part 2: TFD 실전	50
요구사항에서 테스트로	50
테스트 구조: Arrange-Act-Assert	53
체인의 각 서비스 테스트하기	53
TFD 체크리스트	54
Part 3: DGTF 실전	55
5 단계 워크플로우	55
실제 시나리오: “이 버그 지금 당장 고쳐!”	56
커뮤니케이션 템플릿	58
피해야 할 안티패턴	59
DGTF 체크리스트	60

Part 4: 완전한 예제 - 이커머스 장바구니	61
요구사항	61
Step 1: ROD - 서비스 체인 구축	61
Step 2: TFD - 테스트 설계	62
결과	62
Part 5: 주니어 개발자를 위해	63
어디서 시작할까	63
1 개월차: DGTF 부터 시작	63
2 개월차: TFD 추가	63
3 개월차: ROD 추가	64
Part 6: 성공 측정	64
개인 지표	64
진전의 징후	64
Part 7: FAQ & 어려운 질문들	65
Q: “선행 시간이 너무 오래 걸려”	65
Q: “매니저가 당장 결과를 원해”	65
Q: “레거시 코드에 어떻게 적용해?”	66
어려운 질문들 (악마의 변호인)	66
Part 8: 실행 계획	74
1 주차: 이해	74

2 주차: DGTF 연습	74
3 주차: TFD 연습	74
4 주차: ROD 연습	75
2-3 개월: 통합	75
마지막 노트	75
기억하라	75
여정	76
클릭할 때	76
저자 소개	77
김백면 (Bakmeon Kim)	77
주요 경력	77
경험한 산업 분야	77
전문 분야	78
현재 (2025~)	78
원칙의 시작	79
학력	79
철학	79
개인	79
연락처	80

DON'T GO TOO FAST

압박 속에서도 품질을 지키는 소프트웨어 개발 철학

ROD • TFD • DGTF

Kahneman, Meadows, Altshuller 의 통합

김백면

Third Edition

Don't Go Too Fast

압박 속에서도 품질을 지키는 소프트웨어 개발 철학

Copyright © 2026 김백면 (Bakmeon Kim)

All rights reserved.

Third Edition: 2026

이론적 기반:

- Daniel Kahneman: Thinking, Fast and Slow (2011)
- Donella H. Meadows: Thinking in Systems (2008)
- Genrich Altshuller: TRIZ (1946-1998)

서문: 왜 이 책을 썼는가

반복되는 패턴

25년 넘게 소프트웨어를 만들어왔다.

2000년에 주니어 개발자로 시작해서, 연구소장을 거쳐, 제조업, AI, 스마트팩토리, 보안 시스템까지 수많은 프로젝트를 이끌었다. 한국, 대만, 일본, 미국에서 팀들과 일했다. 무엇이 되고 무엇이 안 되는지 봐왔다.

그리고 그 모든 경험을 통해, 같은 패턴을 계속 목격했다.

좋은 개발자가 압박 속에서 나쁜 결정을 내린다.

실력이 없어서가 아니다. 몰라서가 아니다. 하지만 마감이 다가오면, 요구사항이 막판에 바뀌면, 매니저가 “언제 끝나요?”라고 물으면—무언가가 일어난다. 신중한 사고가 급한 반응으로 바뀐다. 체계적인 접근이 임시방편으로 무너진다.

재능 있는 엔지니어가 더 나은 방법을 알면서도 전역 변수를 쓰는 것을 봤다. 잘 설계된 시스템이 출시 직전 몇 주 만에 스파게티 코드가 되는 것을 봤다. 다른 회사, 다른 기술, 다른 시대에서 같은 실수가 반복되는 것을 목격했다.

그리고 깨달았다: 이것은 스킬 문제가 아니다. 사고방식 문제다.

나를 괴롭힌 질문

1996년, 군 복무 후 소프트웨어 공학 수업을 들었다. 데이터베이스나 네트워킹처럼 특정 기술에 집중하는 과목과 달리, 소프트웨어 공학은 더 깊은 질문을 던졌다:

무엇이 좋은 소프트웨어인가? 어떻게 지속되는 시스템을 만드는가?

이 질문이 내 집착이 되었다.

“좋은 소프트웨어를 만든다는 것은 무엇인가?”가 평생의 탐구가 되었다. 단지 작동하는 소프트웨어가 아니다—아무 프로그래머나 작동하는 소프트웨어는 만들 수 있다. 하지만 요구사항이 바뀌어도 좋은 소프트웨어. 새 기능을 싸우지 않고 환영하는 소프트웨어. 현실과 만나도 살아남는 소프트웨어.

소프트웨어 공학 석사를 마쳤다. 15년 넘게 연구소를 이끌었다. 10년 넘게 팀과 소프트웨어 공학 스터디를 진행했다. 계속 질문했다.

그리고 천천히, 수많은 프로젝트와 실패와 성공을 통해, 답을 찾기 시작했다.

세 가지 발견

첫 번째 돌파구는 Daniel Kahneman의 인간 사고에 대한 연구를 만났을 때였다.

노벨상 수상자 Kahneman은 우리에게 두 가지 사고 모드가 있다고 밝혔다: System 1(빠르고, 자동적이고, 오류가 많은)과 System 2(느리고, 의도적이고, 정확한). 압박 속에서는 System 1이 지배한다. 이것이 내가 관찰한 모든 것을 설명했다. 마감이 압박할 때, 개발자들은 전역 변수를 쓰겠다는 의식적 결정을 내리는 게 아니었다—그들의 빠른 사고 System 1이 떠오르는 첫 번째 해결책을 잡는 것이었다.

두 번째 돌파구는 Donella Meadows와 시스템 사고에서 왔다.

Meadows는 부분이 아닌 전체를 보는 법을 가르쳐주었다. 어떤 시스템에서든 가장 강력한 개입 지점은 설계 단계—구현이 시작되기 전—라는 것을 보여주었다. 미리 완성된 서비스 체인이 구현 중의 혼란을 방지할 수 있다. “부족한 것보다 많은 게 낫다”가 핵심 원칙이 되었다.

세 번째 돌파구는 Genrich Altshuller과 TRIZ에서 왔다.

수백만 개의 특허를 분석한 Altshuller은 대부분의 문제가 사람들이 타협하려는 모순을 포함한다는 것을 발견했다. “빠른 개발” vs “높은 품질”은 트레이드오프처럼 보인다. 하지만 TRIZ가 가르쳐주었다: 타협하지 말고—모순을 해결하라. 설계 시간(느리고 신중하게)과 구현 시간(빠르지만 가이드된)을 분리함으로써, 속도와 품질 둘 다 달성할 수 있다.

이론에서 실천으로

이 이론들을 이해하는 것은 한 가지였다. 적용하는 것은 또 다른 것이었다.

수년에 걸쳐, 여러 산업에서 일하면서—SCM 시스템, AI 플랫폼, 스마트팩토리, AR/VR 솔루션, 보안 애플리케이션—세 가지 실용적인 방법론을 개발했다:

ROD (Responsibility-Oriented Design): 구현 전에 완전한 서비스 체인을 설계하여, System 1 패닉 반응을 촉발하는 “누락된 조각”을 제거한다.

TFD (Test-First Development): 요구사항을 테스트로 취급하여, 오류를 일찍 잡고 구현을 가이드하는 피드백 루프를 만든다.

DGTF (Don't Go Too Fast): 압박 속에서도 System 2 사고를 유지하고, 트리거를 인식하고 반응하기 전에 멈춘다.

이 방법론들은 효과가 있었다. 이것을 채택한 팀들은 버그가 적은 더 좋은 소프트웨어를 만들었다. 프로젝트가 예측 가능해졌다. 막판의 미친 듯한 허둥댈이 사라졌다. 엔지니어들이 제시간에 퇴근했다.

이 방법들을 팀에 가르쳤다. 스터디를 진행했다. 실제 피드백을 바탕으로 접근법을 다듬었다. 하지만 내부에만 두었고, 동료들과만 공유했다.

지금까지는.

내가 직접 살아온 증거

어떤 사람들은 묻는다: “Don't Go Too Fast? 실제 세계에서 가능한가?”

내 답은 내 커리어다.

25년 커리어 중 19년을 한 회사에서 일했다. 15년 넘게 연구소장을 맡았다.

개발자들이 보통 2-3년마다 이직하는 시대에, 나는 넓이보다 깊이를 선택했다. 머물면서 만들었다. 다듬고 개선했다. 분기가 아닌 수십 년에 걸쳐 솔루션이 진화하는 것을 지켜봤다.

왜?

제대로 가는 것이 빨리 가는 것을 이기기 때문이다. 깊이가 넓이를 이기기 때문이다. 결국 거북이가 토끼를 이긴다.

이 책은 그 25년의 결과물이다. 상아탑의 이론이 아니라, 소프트웨어를 출시하고, 마감을 맞추고—가장 중요하게—모든 것이 타협하라고 압박할 때 품질을 유지한 일상의 현실에서 단련된 지혜다.

내가 “Don’t Go Too Fast”라고 말할 때, 어딘가에서 읽은 조언을 제공하는 것이 아니다. 사반세기 동안 어떻게 일해왔는지를 공유하는 것이다.

왜 지금인가?

수년간 이 아이디어를 출판하는 것을 꺼렸다.

아마 게으름이었을 것이다. 아마 더 다듬어야 한다는 느낌이었을 것이다. 아마 “완벽”하지 않은 작업을 공유하는 것에 대한 전형적인 엔지니어의 망설임이었을 것이다.

하지만 최근 무언가가 바뀌었다.

2025년, AI 기반 보안 솔루션을 개발하면서 AI 도구와 집중적으로 협업하고 있다. 직접 경험해보니, AI 를 잘 다루면 3-4 명의 개발자와 함께 일하는 것과 유사한 생산성을 확보할 수 있다.

하지만 이것이 DGTF 를 불필요하게 만들었을까? 정반대다.

AI 가 코딩을 가속화할수록, 신중한 설계가 더 중요해진다. AI 가 더 빨리 만들어줄수록, 무엇을 만들지 더 잘 생각해야 한다. 도구가 강력해질수록, 도구를 쓰는 사람의 판단이 더 중요해진다.

이 책의 방법론은 AI 에 의해 위협받지 않는다—더 필수적이 된다.

이 책이 해결하지 않는 것

솔직히 말하겠다:

- 독성적인 조직 문화는 고칠 수 없다
- 지쳤을 때 마법처럼 작동하지 않는다
- 주니어를 즉시 전문가로 만들지 못한다
- 빠르게 읽는 것으로 수개월의 연습을 대체할 수 없다

이것을 포함하는 이유는, 솔직한 한계가 솔직한 방법론을 만들기 때문이다. 실전 가이드의 “어려운 질문들” 섹션에서 이러한 한계와 대응 방법을 더 자세히 다룬다.

이 책이 제공하는 것

이 책은 세 가지를 제공한다:

첫째, 이론적 기반. ROD, TFD, DGTF 가 왜 효과가 있는지, 인지심리학, 시스템 사고, 혁신 이론에서 뒷받침되어 이해하게 될 것이다.

둘째, 실용적 방법론. 무엇을 해야 하는지 구체적이고 실행 가능한 기법을 얻게 될 것이다—“더 잘 생각해라” 같은 모호한 조언이 아니라 내일 사용할 수 있는 체크리스트, 워크플로우, 패턴.

셋째, 입증된 결과. 이것들은 연구실 아이디어가 아니다. 수십 년에 걸쳐 현실 프로젝트에서 다듬어진 방법론으로, 현실의 팀과 현실의 마감에서 테스트되었다.

누구를 위한 책인가

이 책은:

- 바닥까지 태워본 적 있는 **시니어 개발자**
- 코딩은 할 수 있지만 왜 작동하는지 궁금한 **주니어 개발자**
- 예측 가능한 결과가 필요한 **기술 리더**
- 지속 가능한 문화를 만들고 싶은 **팀**

당신이 추구하는 것:

- 압박 속에서 품질 유지
- 기술 부채 감소
- 예측 가능한 개발 일정
- 막판 위기 없는 지속 가능한 페이스

를 위한 것이다.

개인적인 말

이 책을 쓰면서 과거의 나 자신을 많이 생각했다. 막내 개발자로서 서둘러 무언가를 배우고 자신을 증명하려고 힘들어하던 때. 매니저로서 팀의 과로와 버그 급증을 보며 좌절하던 때.

내가 더 일찍 이것을 알았더라면.

그래서 출판하는 것이다. 인식의 벽 너머에 손을 내밀기 위해. 25년 전의 나 자신에게, 그리고 같은 것을 더 빨리 찾고 있는 당신에게 말하기 위해.

우리 산업은 “빠른 실패, 자주 실패”를 찬양한다. 하지만 그것이 유일한 길은 아니다. 신중하게 움직이고, 한 번에 올바르게 만들고, 부수기보다 건설할 수 있다.

느리게 가는 것이 더 빠르다. 신중하게 가는 것이 승리한다. 거북이가 이긴다.

시작하자.

Part 1: 핵심 개념과 가치

문제: What 이 아니라 How

Clean Code 를 읽었는가?

SOLID 원칙을 아는가?

TDD 를 들어봤는가?

아마 “네”일 것이다.

그런데 왜 금요일 오후 마감 앞에서 전역 변수를 쓰는가?

많은 개발자:

"Clean Code? 알아요"

"SOLID? 알죠"

"TDD? 들어봤어요"

현실:

금요일 오후 5 시

매니저: “이거 언제 되나요?”

→ “일단 전역 변수로...”

→ “하드코딩하면 빨리...”

→ “테스트? 나중에...”

문제는 “What”을 모르는 게 아니다. 문제는 “How”를 모르는 것이다.

- What: Clean Code, SOLID, TDD (이미 안다)
- How: 압박 속에서 어떻게 지키는가? (이것이 필요하다)

ROD, TFD, DGTF 는 “How”에 대한 답이다.

이 방법론은 새로운 “What”이 아니다. 좋은 개발을 “왜”, “언제”, “어떻게” 해야 하는지에 대한 답이다.

이론적 기반

이 방법론은 세 가지 검증된 이론 위에 세워졌다. 이 이론들을 이해하는 것은 필수다—학술적 지식으로서가 아니라, DGTF++가 왜 작동하는지의 기반으로서.

Daniel Kahneman: 왜 압박에서 실패하는가

노벨상을 받은 통찰:

Daniel Kahneman 은 인간의 의사결정이 합리적이지 않다는 것을 증명하여 2002 년 노벨 경제학상을 받았다. 그의 연구는 두 가지 뚜렷한 사고 시스템을 밝혀냈다:

System 1 (빠른 사고)

특성:

- 자동적이고 무의식적
- 노력이 필요 없음
- 항상 작동 중
- 패턴 매칭 기반
- 감정적

강점:

- 즉각적 반응
- 에너지 효율적
- 익숙한 상황에 적합
- 우리를 살게 함 (투쟁 또는 도피)

약점:

- 인지 편향에 취약
- 성급한 결론
- 복잡성 처리 못함
- 압박에서 실수
- 과신

System 2 (느린 사고)

특성:

- 의도적이고 의식적
- 노력과 집중 필요
- 의도적으로 활성화해야 함
- 논리와 추론 기반
- 분석적

강점:

- 정확한 판단
- 복잡성 처리 가능
- 여러 요소 고려
- 장기적 사고
- 자기 인식

약점:

- 느림
- 피곤함 (정신 에너지 소모)
- 게으름 (활성화 회피)
- 압박에서 꺼짐
- 제한된 용량

핵심 발견:

System 2 는 게으르다. 강제하지 않으면 활성화되지 않는다. 압박 하에서 System 2 는 완전히 꺼지고, System 1 이 완전히 제어한다.

소프트웨어 개발에서의 의미:

상황	시스템	전형적 결정
차분한 설계 회의	System 2	“엣지 케이스 생각해보자”
금요일 오후 5 시, 월요일 마감	System 1	“일단 하드코딩하자”
코드 리뷰, 압박 없음	System 2	“SRP 위반이네, 리팩터링하자”
프로덕션 다운	System 1	“일단 서버 재시작”

상황	시스템	전형적 결정
새 개념 학습	System 2	“완전히 이해해보자”
이 시간에 세 번째 버그	System 1	“if 문 하나 더 추가”

개발자의 딜레마:

설계 단계:

- 시간 여유
- 낮은 압박
- System 2 활성
- 좋은 결정 가능

구현 단계:

- 마감 압박
- 변경되는 요구사항
- System 1 이 지배
- "빠른 수정"이 쌓임

결과:

아름다운 아키텍처를 설계한 바로 그 개발자가
압박 하에서 스파게티 코드를 작성한다.
더 좋은 방법을 몰라서가 아니다.
System 1 은 "더 좋은 것"을 신경 쓰지 않기 때문이다.

DGTF++의 활용:

1. **ROD:** System 2 가 활성화되어 있을 때 설계를 완료한다. System 1 이 결정할 것을 남기지 않는다.
2. **TFD:** System 2 가 활성화되어 있을 때 테스트를 정의한다. 구현은 기계적이 된다—그냥 테스트를 통과시키면 됨.
3. **DGTF:** System 1 이 지배하려 할 때 인식한다. 압박 속에서도 System 2 활성화를 강제한다.



Donella Meadows: 왜 전체를 봐야 하는가

시스템 사고의 선구자:

Donella Meadows 는 복잡한 시스템을 이해하는 데 가장 영향력 있는 책 중 하나인 “Thinking in Systems”를 쓴 시스템 과학자다. 그녀의 통찰은 왜 개별 컴포넌트가 잘 만들어졌는데도 소프트웨어 프로젝트가 실패하는지를 설명한다.

시스템이란 무엇인가?

시스템은 목적을 달성하기 위해 조직된 상호 연결된 요소들의 집합이다. 전체는 부분의 합보다 크다.

예시:

자동차는 단순한 부품이 아니다:

- 엔진 + 바퀴 + 핸들 + 브레이크 = 이동 수단
- 어떤 부품이든 제거 = 이동 수단 아님
- 목적이 연결에서 나온다

소프트웨어도 마찬가지:

- UserService + AuthService + Database = 로그인 기능
- 어떤 부분이든 제거 = 로그인 안 됨
- 행동이 연결에서 나온다

시스템 사고의 핵심 원칙:

1. 전체를 먼저 보라

잘못된 접근:

“UserService 부터 만들고, 나머지는 나중에 파악하지”

- 구현 중 빠진 조각 발견
- 패닉 → System 1 → 나쁜 결정

올바른 접근 (ROD):

“먼저 완전한 서비스 체인을 그리자”

- 설계에서 모든 조각 파악
- 구현은 알려진 조각을 만드는 것일 뿐

2. 관계가 부품보다 중요하다

잘 설계된 서비스 + 나쁜 연결 = 나쁜 시스템
단순한 서비스 + 좋은 연결 = 좋은 시스템

ROD 가 집중하는 것:

- 서비스가 어떻게 연결되는지 (인터페이스)
- 그 사이에 뭐가 흐르는지 (데이터)
- 누가 누구에게 의존하는지 (의존성)

3. 피드백 루프가 행동을 결정한다

양성 피드백 (증폭):

버그 → 스트레스 → 급함 → 더 많은 버그 → 더 많은 스트레스 → ...
→ 시스템이 통제 불능으로

음성 피드백 (안정화):

코드 → 테스트 → 실패 → 수정 → 테스트 → 통과
→ 시스템이 자기 교정

TFD 가 음성 피드백 루프를 만든다:

모든 변경이 즉시 검증된다.

문제가 증폭되기 전에 잡힌다.

4. 레버리지 포인트: 어디에 개입할 것인가

Meadows 는 어떤 개입 지점이 다른 것보다 훨씬 효과적이라는 것을 밝혔다:

낮은 레버리지 (비싸고 영향 적음):

- 프로덕션에서 버그 수정
- 코드 작성 후 테스트 추가
- 레거시 코드 리팩터링

높은 레버리지 (싸고 영향 큼):

- 설계 단계 결정
- 코드 전에 테스트 정의
- 서비스 경계 일찍 확립

ROD + TFD 는 가장 높은 레버리지 포인트에서 작동:
설계 단계.

5. "Missing"의 위험

요소가 빠진 시스템은 "대부분" 작동하는 게 아니다.
예측 불가능하게 행동한다.

예시:

에러 처리 없는 결제 시스템:

- 99% 시간은 작동
- 1% 실패가 데이터를 조용히 손상
- 몇 달 후에 발견
- 피해가 치명적

ROD 의 "있는 게 없는 것보다 낫다":

에러 처리를 설계하고 제거하는 게
잊어버리고 프로덕션에서 발견하는 것보다 낫다.

빙산 모델:

보이는 것: 사건 (버그, 자연, 실패)
 ↑

원인: 패턴 (반복되는 문제)
 ↑

형성하는 것: 구조 (아키텍처, 프로세스)
 ↑

근본: 멘탈 모델 (우리가 생각하는 방식)

대부분의 수정은 사건을 다룬다.

DGTF++는 멘탈 모델을 다룬다.

그래서 근본에서 작동한다.

DGTF++의 활용:

1. **ROD:** 부품을 만들기 전에 전체 시스템(서비스 체인)을 본다. 빠진 조각 없음.
 2. **TFD:** 안정화 피드백 루프를 만든다. 모든 변경이 즉시 검증된다.
 3. **DGTF:** 가장 높은 레버리지 포인트에서 작업한다—당신 자신의 사고 패턴.
-

Genrich Altshuller: 왜 타협이 아닌 해결이어야 하는가

TRIZ 의 아버지:

Genrich Altshuller 는 20 만 개 이상의 특허를 분석하여 혁신의 패턴을 발견한 소비에트 엔지니어다. 그의 발견: 획기적인 해결책은 타협이 아닌 모순 해결에서 나온다.

모순이란 무엇인가?

한 측면을 개선하면 다른 측면이 나빠질 때 모순이 존재한다.

소프트웨어의 근본적 모순:

"빠른 개발을 원한다"

vs

"높은 품질을 원한다"

전통적 "해결책" (사실은 타협):

- 빠르게 → 품질 희생 (기술 부채)
- 품질 → 속도 희생 (마감 놓침)
- 균형 → 둘 다 못 얻음 (모든 게 평범)

이 중 어떤 것도 해결책이 아니다.

이것들은 항복이다.

TRIZ 의 혁명적 통찰:

모순은 균형 잡아야 할 문제가 아니다. 모순은 혁신의 기회다.

40 가지 발명 원리:

Altshuller 는 발명가들이 반복적으로 사용하는 40 가지 원리를 파악했다. 세 가지가 소프트웨어에 특히 관련된다:

원리 1: 분할 (Segmentation)

문제: 모놀리식 시스템은 변경하기 어렵다

타협: 변경이 위험하다는 것을 받아들임

TRIZ 해결: 독립적인 세그먼트로 분할

- 명확한 경계를 가진 서비스들
- 다른 것에 영향 없이 하나 변경
- ROD 의 서비스 체인

원리 10: 사전 조치 (Prior Action)

문제: 구현 중 문제 발견

타협: 디버깅이 개발의 일부라는 것을 받아들임

TRIZ 해결: 필요하기 전에 미리 하기

- 구현 전 완전히 설계
- 코드 전 테스트 작성
- ROD + TFD

원리 15: 동적화 / 시간 분리 (Dynamization)

문제: 빠르면서 동시에 신중해야 함

타협: 어느 정도 빠르고, 어느 정도 신중 (둘 다 아님)

TRIZ 해결: 다른 시간에 다른 것이 되기

- 설계 단계: 느리고 철저 (100% 신중)
- 구현 단계: 빠른 실행 (100% 빠름)
- DGTF++ 접근법

분리 원리:

TRIZ 는 모순을 해결하는 네 가지 방법을 제시한다:

1. 시간 분리

"빠르면서 신중하게" → 지금은 신중, 나중에 빠르게

→ 설계는 느리게, 구현은 빠르게

2. 공간 분리

"결합되면서 독립적" → 다른 경계

→ 서비스는 독립적, 시스템은 결합

3. 스케일 분리

"단순하면서 완전" → 다른 레벨

→ 각 서비스 단순, 전체 시스템 완전

4. 조건 분리

"유연하면서 안정적" → 다른 트리거

→ 인터페이스 안정, 구현 유연

이상적 최종 결과 (Ideal Final Result):

Altshuller 가 물었다: "이상적인 해결책은 어떻게 생겼을까?"

이상적인 소프트웨어 개발:

- 버그 제로
- 재작업 제로
- 혼란 제로
- 즉각 완료

어떻게 접근하나:

- 버그는 빠진 설계에서 → 완전한 설계 (ROD)
- 재작업은 불명확한 요구사항에서 → 테스트가 요구사항 (TFD)
- 혼란은 System 1에서 → System 1 통제 (DGTF)
- 느린 완료는 실수 수정에서 → 실수 방지 (세 가지 모두)

DGTF++의 활용:

1. **ROD:** 사전 조치 + 분할. 구현 전 모든 것 설계. 독립적 서비스로 분할.

2. **TFD**: 사전 조치. 만들기 전 성공 기준 정의.
 3. **DGTF**: 시간 분리. 먼저 생각 (System 2), 그다음 행동 (빠를 수 있음).
-

왜 이 세 가지가 함께인가

각 이론은 다른 질문에 답한다:

Kahneman → "왜 압박에서 실패하는가?"

답: System 1이 지배하고 나쁜 결정을 내린다.

Meadows → "실패를 방지하려면 무엇을 준비해야 하는가?"

답: 피드백 루프를 가진 완전한 시스템.

TRIZ → "속도와 품질을 어떻게 둘 다 얻는가?"

답: 분리를 통해 모순을 해결.

통합:

DGTF++ 프레임워크

Kahneman: 언제 생각할 것인가

설계 단계 = System 2 시간

구현 = System 1으로부터 보호

Meadows: 무엇을 생각할 것인가

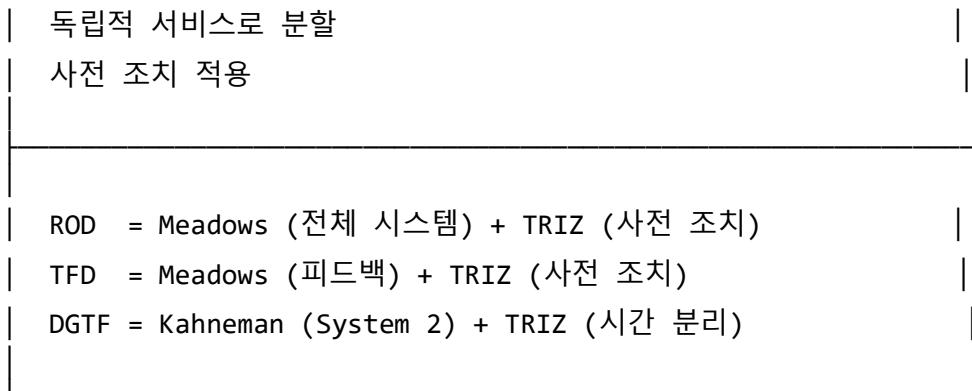
전체 시스템을 보라

피드백 루프를 만들라

높은 레버리지 포인트에서 작업하라

TRIZ: 모순을 어떻게 해결할 것인가

생각과 실행을 시간으로 분리



어떤 이론 하나만으로는 충분하지 않다:

- Meadows 없이 Kahneman: 언제 생각할지 알지만, 무엇을 생각할지 모름
- TRIZ 없이 Meadows: 무엇을 준비할지 알지만, 타협에 간힘
- Kahneman 없이 TRIZ: 어떻게 해결할지 알지만, 압박에서 실행 못함

함께, 지속 가능한 소프트웨어 개발을 위한 완전한 시스템을 형성한다.

ROD: Responsibility-Oriented Design

핵심 원칙

“있는 게 없는 것보다 낫다” (More is better than missing)

설계 단계에서 완전한 서비스 체인을 만들어라.

새 서비스를 만들지 말지 헷갈릴 때—만들어라.

왜? 체인이 빠져있고 구현 중에 발견하면, System 1 이 그 상황을 좋아할 것이다. 그리고 System 1 이 좋아한다는 것은 나쁜 급한 결정을 의미한다.

비대칭성: 제거 vs 추가

이 원칙은 근본적인 비대칭성에 기반한다:

구현 중 불필요한 서비스 제거:

상황: "이 서비스 호출되는 곳이 없네"

증거: 명확함 (사용 안 됨)

정신 상태: 차분함 (System 2)

행동: 삭제

위험: 낮음

노력: 쉬움

구현 중 빠진 서비스 추가:

상황: "이게 필요한데 없잖아!"

증거: 패닉 발견

정신 상태: 압박 (System 1 지배)

행동: 빠른 땀질

위험: 높음

노력: 어려움 그리고 위험함

비대칭성이 명확하다: - 제거 = 쉬움, 안전함, System 2 - 추가 = 어려움, 위험함, System 1

따라서: 설계 중 헛갈리면 포함시켜라. 나중에 언제든 제거할 수 있다.

서비스 체인이란?

서비스 체인은 요구사항을 충족하는데 필요한 모든 서비스와 그 관계의 완전한 지도다.

예: "사용자가 상품을 구매할 수 있다"

서비스 체인:

UserService → CartService → PaymentService → OrderService → InventoryService
→ NotificationService

각 화살표 = 의존성

각 서비스 = 명확한 책임

완전한 체인 = 구현 중 놀람 없음

핵심 질문: > “이 요구사항이 서비스 체인만으로 달성을 수 있는가?”

- YES → 체인 완전
- NO → 뭔가 빠짐 → 설계 단계에서 서비스 추가 (구현 단계가 아니라!)

ROD 는 새로운 “What”이 아니다

솔직해지자.

ROD 는 완전히 새로운 게 아니다: - Clean Architecture? 비슷하다. - Service-Oriented Design? 그렇다. - Domain-Driven Design? 겹치는 부분이 있다. - SOLID? 포함되어 있다.

하지만 질문:

Clean Architecture 를 안다. SOLID 를 안다. 의존성 주입을 안다.

그런데 왜 마감이 다가오면 지키지 못하는가? 왜 “일단 되게 하고 나중에 고치자”가 되는가?

ROD 는 새로운 기술이 아니다.

ROD 는 좋은 설계를 “왜”, “언제”, “어떻게” 해야 하는지에 대한 답이다.

Clean Architecture:

“이렇게 구조화해라”

→ What (무엇을 할지)

ROD:

“왜 이렇게 해야 하는지” (Kahneman: System 1 방지)

“언제 이렇게 해야 하는지” (설계 단계, System 2 가 활성화될 때)

“어떻게 유지하는지” (완전한 서비스 체인, Missing 없음)

→ Why + When + How

진짜 힘: 변경 격리

아무리 완벽하게 설계해도, 구현 중 변경은 일어난다.

- 요구사항 변경
- 더 좋은 방법 발견
- 버그 수정 필요

ROD 는 변경을 막지 않는다. ROD 는 변경을 격리한다.

ROD 없이 (강결합):

A 변경 → B 에 영향 → C 에 영향 → D 에 영향 → ...

→ 변경이 두려움

- "건드리지 마, 동작하잖아"
- 기술 부채 축적

ROD로 (서비스 체인):

- A 변경 → A의 인터페이스 유지 → B, C, D 영향 없음
- 변경이 안전함
- 자신 있게 리팩터링
- 지속 가능한 코드베이스

서비스 체인이 경계를 만든다. 각 서비스는 명확한 책임이 있다. 서비스 내부의 변경은 외부로 새지 않는다.

엄격한 규칙

서비스 체인 무결성을 유지하기 위해:

- ✗ 서비스 내부에서 생성자(new) 금지
- ✗ Static 필드나 메서드 금지
- ✗ 구현에 대한 가정 금지
- ✓ 모든 것을 서비스로 표현
- ✓ 의존성은 주입, 생성하지 않음

왜 이런 규칙인가? 왜냐하면: - new 는 숨겨진 의존성을 만든다 (체인 가시성 파괴) - static 은 전역 상태를 만든다 (격리 파괴) - 가정은 놀람이 된다 (System 1 트리거)

ROD 가 완전할 때

검증 기준:

각 요구사항에 대해 물기:

"서비스 체인만으로 이것이 달성될 수 있는가?"

모든 요구사항에 YES → ROD 완전

어떤 요구사항에 NO → Missing 존재 → 설계에 추가

완전한 ROD 의 징후:

- 모든 서비스가 정확히 하나의 책임을 가짐
 - 모든 의존성이 체인에서 보임
 - 어떤 서비스도 다른 서비스의 구현을 알 필요 없음
 - 요구사항이 서비스 체인에 직접 대응
 - 구현은 “체인을 따라가기만 하면 됨”
-

TFD: Test-First Development

핵심 원칙

“요구사항 = 테스트”

테스트를 작성할 수 없다면, 요구사항을 이해하지 못한 것이다.

테스트는 사후 검증이 아니다. 테스트는 요구사항이 무엇을 의미하는지의 정밀한 정의다.

왜 요구사항이 테스트여야 하는가

자연어 요구사항의 문제:

요구사항: “사용자가 로그인할 수 있어야 한다”

이것이 답하지 않는 질문들:

- 틀린 비밀번호면 어떻게 되나?
- 잠금 전까지 몇 번 시도 가능한가?
- “로그인됨”이 뭘 의미하나? 토큰? 세션? 쿠키?
- 만료된 계정은?
- 인증 안 된 이메일은?

같은 요구사항을 테스트로:

`test_login_success:`

Given: 유효한 이메일, 올바른 비밀번호, 인증된 계정

When: 로그인 시도

Then: JWT 토큰 반환, 24 시간 유효

`test_login_wrong_password:`

Given: 유효한 이메일, 틀린 비밀번호

When: 로그인 시도

Then: 401 반환, 메시지 "잘못된 자격증명"

test_login_account_locked:

Given: 연속 3회 실패

When: 올바른 비밀번호로 4번째 시도

Then: 403 반환, 메시지 "30분간 계정 잠금"

test_login_unverified_email:

Given: 유효한 자격증명, 이메일 미인증

When: 로그인 시도

Then: 403 반환, 메시지 "이메일을 인증해주세요"

이제 요구사항이 정밀하다. 모호함 없음. 구현 중 놀람 없음.

테스트는 완료 기준이다

테스트 없이 "완료"는 주관적이다:

개발자: "끝났어요"

QA: "이 케이스 처리 안 되는데요"

개발자: "요구사항에 없었어요"

QA: "당연한 거잖아요"

개발자: "저한텐 아닌데요"

→ 충돌, 재작업, 좌절

TFD로 "완료"는 객관적이다:

모든 테스트 통과 → 완료

테스트 하나라도 실패 → 미완료

새 케이스 발견 → 먼저 테스트 추가, 그다음 구현

테스트는 요구사항과 구현 사이의 계약이다.

TFD 와 ROD 의 결합

TFD는 ROD와 결합할 때 강력해진다:

ROD 가 제공: 완전한 서비스 체인
TFD 가 제공: 각 서비스에 대한 테스트

함께:

- 체인의 각 서비스가 테스트를 가짐
- 테스트가 서비스의 계약을 정의
- 구현은 그냥 계약을 이행

예:

ROD 서비스 체인:
UserService → AuthService → TokenService

TFD 테스트:

UserService:

- test_find_by_email_exists
- test_find_by_email_not_found

AuthService:

- test_validate_password_correct
- test_validate_password_wrong
- test_check_account_status_active
- test_check_account_status_locked

TokenService:

- test_generate_token_valid_user
- test_token_contains_required_claims
- test_token_expires_in_24_hours

구현할 때, 그냥 테스트를 통과시키는 것이다. 추측 없음. 모호함 없음. System 1 결정 없음.

TDD 와의 관계

TFD 는 TDD 를 대체하지 않는다. TFD 는 TDD 를 쉽게 만든다.

TDD 의 흔한 어려움:

TDD 가 말함: "테스트 먼저 작성해"

개발자 생각: "뭘 테스트하지? 아직 뭘 만들지도 모르는데"

→ 코드 먼저 작성

→ TDD 포기

TFD 가 해결:

ROD: 서비스 체인 정의 (어떤 서비스가 존재하는지)

TFD: 각 서비스에 대한 테스트 정의 (각 서비스가 뭘 하는지)

TDD: Red → Green → Refactor (어떻게 구현하는지)

진행 순서:

설계 단계 (System 2):

1. ROD: 서비스 체인 구축
2. TFD: 각 서비스에 대한 테스트 케이스 작성

구현 단계 (DGTF 가 System 2 보호):

3. TDD: 각 테스트에 대해 Red → Green → Refactor

TFD 가 TDD 시작 전에 “뭘 테스트할지”를 답한다.

테스트는 즉각적 피드백을 제공한다

Meadows 의 시스템 사고에서: > “피드백 루프가 길수록 배우기 어렵다.”

테스트 없이 (긴 피드백 루프):

코드 작성 → 배포 → 사용자가 버그 보고 → 디버깅 → 원인 찾기 → 수정

시간: 며칠에서 몇 주

학습: 어려움, 맥락 잊음

테스트로 (짧은 피드백 루프):

코드 작성 → 테스트 실행 → 실패 → 수정 → 통과

시간: 초에서 분

학습: 즉각적, 맥락 신선

테스트는 올바른 방향을 유지하게 하는 피드백 메커니즘이다.

TFD 가 완전할 때

ROD 의 각 서비스에 대해:

- 모든 공개 메서드에 테스트 있음
- 모든 엣지 케이스 커버됨
- 모든 오류 조건 테스트됨
- 테스트가 요구사항으로 읽힘

완전한 TFD 의 징후:

- 새 팀원이 테스트를 읽고 서비스가 뭘 하는지 이해할 수 있음
 - “행복 경로”와 “불행 경로” 모두 테스트됨
 - 테스트가 구현 상세에 의존하지 않음
 - 테스트가 살아있는 문서 역할
-

DGTF: Don't Go Too Fast

핵심 원칙

“Slow is smooth, smooth is fast”

신중함은 느린 게 아니다—부드러운 것이다. 그리고 부드러움은 결국 빠르다, 재작업이 없기 때문에.

전제조건: “Wow” 순간

DGTF 에는 전제조건이 있다.

DGTF 의 첫 단계는 “인식”—System 1 이 활성화되고 있음을 인식하는 것이다. 그런데 여기에 역설이 있다:

System 1 이 완전히 활성화되면, 자신이 System 1 상태인지 모른다.

누가 DGTF 를 사용할 수 있는가?

“Wow” 순간을 경험하는 사람들:

“잠깐... 이게 맞나?”

“음, 뭔가 이상한데...”

“잠시만, 생각해보자...”

이 짧은 의문의 순간—이것이 “Wow”다.

핵심 통찰:

Wow 없이 → 100% System 1 → 인식 불가 → DGTF 적용 불가

Wow 있지만 DGTF 없이 → 문이 잠깐 열림 → 뭘 해야 할지 모름 → 문 닫힘

Wow + DGTF → 문이 열림 → 정확히 뭘 해야 할지 앎 → System 2 활성화

그래서 DGTF는 Wow가 오기 전에 배워둬야 한다.

Wow 가 왔을 때, 무엇을 해야 하는지 바로 알아야 한다.

LA 비유

매니저가 팀에게 말한다: “LA로 가. 빨리.”

D 씨: LA를 향해 바로 뛰기 시작한다.

“빨리라고 했잖아! 지금 가야해!”

→ 가장 열심히, 가장 느린 결과

C 씨: 자전거를 잡는다.

“최소한 뭔가 하고 있긴 하지...”

→ 타협, 여전히 느림

A 씨: 차를 가지러 집에 간다.

“제대로 된 도구부터 챙기자.”

→ 반대로 가는 것처럼 보이지만, 더 빠름

B 씨: 비행기 스케줄을 검색한다.

“가장 빠른 방법이 뭘까?”

→ 아무것도 안 하는 것처럼 보이지만, 가장 빠름

밖에서 보면: - D 가 가장 열심히 하는 것처럼 보임 (뛰고 있으니까!) - A 는 반대 방향으로 가는 것처럼 보임 - B 는 그냥 앉아있는 것처럼 보임

하지만 결과는 정반대다.

이것이 DGTF다.

누군가 “빨리!”라고 하면: - System 1 (D): “네!” → 뛰기 시작 - DGTF (A/B): “잠깐, 거기 가는 가장 빠른 방법이 뭐지?”

멈춰서 생각하는 사람이 급하게 실패하는 사람을 이긴다.

왜 작동하는가

DGTF 는 System 1 을 통제하고 System 2 를 활성화한다:

1. 인식: “지금 서두르고 있나?”
2. 멈춤: “잠깐, 생각하자”
3. 확인: ROD 설계, TFD 테스트, 영향 분석
4. 계획: 명확한 단계
5. 실행: 신중하게, 검증하며

진짜 긴급상황에서도 DGTF 는 적용된다. 5 분 멈추는 게 아니라 5 초 멈추는 것일 뿐.
그래도 여전히: 멈춤 → 생각 → 행동.

DGTF ≠ 느림, DGTF ≠ 의지력

흔한 오해:

- ✗ DGTF = 천천히 일하기
- ✗ DGTF = 급하게 가고 싶은 충동을 참는 것
- ✗ DGTF = 빨리 가려는 욕구를 견디는 것

진실:

- ✓ DGTF = 신중하게 일하기
- ✓ DGTF = 더 힘든 게 아니라 더 효과적인 것
- ✓ DGTF = 재작업 회피로 에너지 절약

DGTF 는 의지력을 소모하지 않는다. 에너지를 절약한다.

DGTF 없이:

급하게 → 버그 → 디버깅 → 수정 → 새 버그 → 더 디버깅 → 지침

DGTF로:

생각 → 올바르게 구현 → 완료 → 에너지 절약

애자일 비유:

어떤 팀들은 말한다: “이 프로젝트 쉬우니까 애자일 하자.” 이건 거꾸로다. 애자일은 어려운 프로젝트에서 더 효과적이다.

DGTF 도 마찬가지다.

- "시간 여유 있으니까 DGTF 하자"
- "압박 받고 있으니까 DGTF 가 필요해"

상황이 어려울수록 DGTF 가 더 도움된다.

DGTF 가 해결하지 않는 것

DGTF 는 만능이 아니다.

DGTF 가 해결하지 않는 것:

1. **독성적인 조직**
 - “천천히 신중하게”가 해고 사유가 되는 곳
 - 비합리적인 마감이 일상인 곳 → 이건 환경 문제지, DGTF 문제가 아니다.
2. **비합리적인 매니저**
 - 신중한 작업이 인정받지 못하는 경우
 - “바빠 보이는 것”만 중요한 경우 → DGTF 가 다른 사람의 태도를 바꿀 수는 없다.
3. **근본적으로 망가진 시스템**
 - 완전히 재작성이 필요한 레거시 코드 → DGTF 는 예방을 위한 것이지, 치료를 위한 게 아니다.

답:

환경이 DGTF 를 허용하지 않으면:

선택 1: 그 환경을 떠나라.

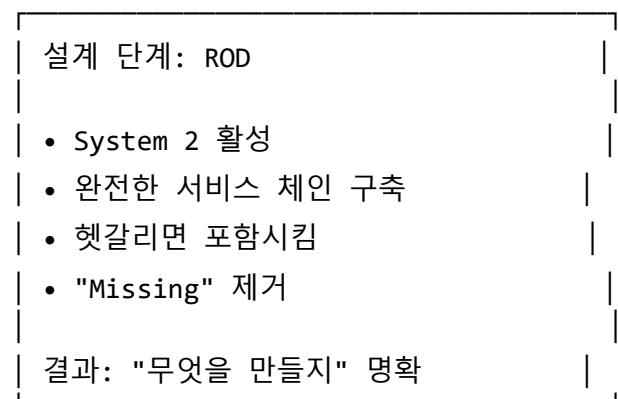
선택 2: 내가 통제할 수 있는 범위 안에서 DGTF 를 적용하라.

선택 2 가 핵심이다:

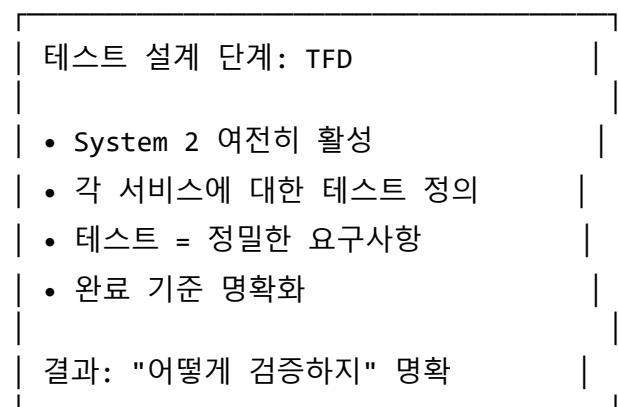
DGTF 는 내면의 프로세스다. - 타이핑 전 3 초 생각하는 것—아무도 모른다. - 코딩 전에 설계 확인하는 것—보이지 않는다. - “빨리” 느낄 때 멈추는 것—나만 안다.

밖에서 보면 똑같아 보인다. 결과만 더 좋다.

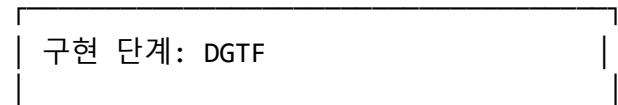
어떻게 함께 작동하는가



↓



↓



• 압박 증가	
• DGTF 가 System 2 유지	
• ROD 설계 따름	
• TFD 테스트로 검증	
결과: "어떻게 만들지" 안전	

시너지:

- ROD 만: 완전한 설계, 하지만 검증 메커니즘 없음
 - TFD 만: 검증 존재, 하지만 설계가 불완전할 수 있음
 - DGTF 만: 신중한 진행, 하지만 방향이나 검증 없음
 - **ROD + TFD + DGTF:** 완전한 설계 + 정밀한 검증 + 신중한 실행 = 예측 가능하고, 지속 가능하며, 고품질 소프트웨어
-

본질: 지식이 아니라 습관

ROD, TFD, DGTF 는:

- ✗ 원칙이 아니다
 ✗ 규칙이 아니다
 ✗ 프로세스가 아니다
 ✗ 지식이 아니다
- ✓ 사고방식이다
 ✓ 습관이다
 ✓ 훈련이다

이해 ≠ 실천

이해:

- 이 문서 읽음 → "알겠어"
- 고개 끄덕임 → "말이 되네"

- 전부 동의함 → "이렇게 해야지"
- 이건 System 1이 "Got it, next!"라고 하는 것

실천:

- 내일 적용 → 어려움
- 실패 → 배움
- 다시 시도 → 조금 나아짐
- 반복 → 결국 자연스러워짐
- 이게 훈련을 통해 습관을 만드는 것

DGTF 를 10 분 안에 이해할 수 있다. 몇 달의 연습 없이 DGTF 를 실천할 수는 없다.

학습이 아니라 훈련

학습: 지식을 얻음 (일회성)

훈련: 습관을 만들 (지속적)

운전을 잘 하려고 "배우는" 게 아니다.

운전을 잘 하려고 "훈련하는" 것이다.

DGTF 를 "배우는" 게 아니다.

DGTF 를 "훈련하는" 것이다.

허들을 넘는 게 끝이 아니다

✗ "DGTF 를 한 번 성공적으로 적용했어!"

→ 끝났나? 아니다. 내일 압박이 다시 온다.

✓ "오늘 DGTF 를 적용했다. 내일 또 할 거다."

→ 그리고 모레도. 그리고 다음 주도.

→ 더 이상 의식적인 노력이 아닐 때까지.

운전 비유

운전의 기본 원칙: - 파란불이면 간다 - 브레이크를 밟으면 멈춘다 - 엑셀레이터를 밟으면 앞으로 간다

이 원칙을 안다고 좋은 운전자인가?

아니다.

좋은 운전자는: - 지속적인 주의 - 상황 판단 - 다른 운전자에 대한 배려 - 이것들이 **습관**으로
몸에 배어 있음

소프트웨어도 마찬가지다.

SOLID, Clean Code, TDD 를 안다고 좋은 개발자가 아니다. 이것들을 **습관**으로 실천할 수
있어야 좋은 개발자다.

차이:

원칙/규칙:

- "이렇게 해야 한다"
- 외부에서 강제됨
- 압박 오면 무너짐

사고방식/습관:

- "이렇게 생각한다"
- 내면에서 나옴
- 압박 와도 유지됨

가장 어려운 부분

DGTF 에서 가장 어려운 건 이해하는 게 아니다. 가장 어려운 건: - 호기심을 유지하는 것 -
의심을 계속하는 것 ("이게 정말 맞나?") - 계속해서 연습하는 것 - 첫 성공 후에 멈추지 않는
것

이건 기술이 아니라 태도가 필요하다.

핵심 가치

1. 견고한 이론적 기반

- Kahneman (노벨상 수상자): 인간의 사고 방식
- Meadows (시스템 권위자): 시스템 작동 방식

- Altshuller (TRIZ 창시자): 문제 해결 방법

이론 + 실천 = 신뢰할 수 있는 방법론

2. 인간을 이해한다

인간의 약점을 인정한다: - 압박에서 실수한다 (Kahneman) - 부분을 보면 전체를 놓친다 (Meadows) - 타협에 익숙하다 (Altshuller)

그리고 시스템으로 보완한다: - ROD: 설계 단계에서 완전하게 - TFD: 피드백으로 검증 - DGTF: 신중함을 강제

3. 보편적으로 적용 가능

- 언어 독립 (Go, Java, Python 등)
- 도메인 독립 (Web, Mobile, Backend 등)
- 팀 규모 독립 (1 인부터 대규모 팀까지)

4. 상호 보완적

- 각각 독립적으로 적용 가능
- 함께 적용하면 시너지
- 점진적 도입 가능

5. 지속 가능성

- 개인: 번아웃 방지, 전문성 향상
- 팀: 일관된 생산성, 높은 사기
- 비즈니스: 예측 가능한 배포, 경쟁 우위

시작하기

오늘: 1. ROD: 다음 기능 설계할 때, 먼저 서비스 체인 작성 2. TFD: 구현 전에 테스트 케이스 정의 3. DGTF: “빨리” 느낄 때 3 초 멈춤

이번 주: - 작은 기능 하나에 적용 - 무슨 일이 일어나는지 관찰 - 어려운 점 메모

이번 달: - 여러 기능에 적용 - 동료와 공유 - 상세 방법은 실전 가이드 참조

기억하라: > “좋은 프로그래머는 빠른 타이핑이 아니라 올바른 사고에서 나온다”

“품질은 검사가 아니라 프로세스에서 만들어진다”

“지속 가능한 개발은 방법론이 아니라 태도에서 시작한다”

상세 구현 가이드, 예제, 체크리스트, 연습은 실전 가이드를 참조하라.

Part 2: 실전 가이드

이 가이드 사용법

전제조건: - “DGTF++ 핵심 개념” 먼저 읽기 - HOW 를 배우기 전에 WHY 를 이해하기

이 가이드가 제공하는 것: - 실전 규칙과 방법 - 단계별 워크플로우 - 코드가 포함된 실제 예제 - 검증용 체크리스트 - 흔한 함정과 해결책

구조: 1. 빠른 참조 (이론 요약) 2. ROD 실전 3. TFD 실전 4. DGTF 실전 5. 완전한 예제 6. 주니어 개발자를 위해 7. 성공 측정 8. FAQ & 어려운 질문들 9. 실행 계획

빠른 참조: 이론 요약

상세 설명은 “핵심 개념” 문서 참조.

Kahneman (언제 생각할 것인가)

System 1: 빠름, 자동적, 오류 발생 쉬움, 압박에서 지배적

System 2: 느림, 의도적, 정확함, 압박에서 꺼짐

→ System 2로 설계 (차분할 때)

→ 구현 중 System 2 보호 (DGTF)

Meadows (무엇을 생각할 것인가)

- 부품 전에 전체 시스템을 보라

- 빠진 요소는 예측 불가능한 동작을 야기

- 피드백 루프가 자기 교정을 가능하게 함

- 설계 단계 = 가장 높은 레버리지 포인트

→ 완전한 서비스 체인 구축 (ROD)

→ 테스트로 피드백 생성 (TFD)

TRIZ (모순을 어떻게 해결할 것인가)

“빠름” vs “품질”은 트레이드오프가 아니다.

시간 분리로 해결:

- 설계 단계: 100% 신중
 - 구현 단계: 100% 빠르게 (계획을 따라서)
- 사전 조치: 필요하기 전에 모든 것을 하라
 → 분할: 독립적인 부분으로 나누라
-

Part 1: ROD 실전

규칙들

규칙 1: 서비스 내부에서 생성자 금지

```
// ✗ 나쁨: 숨겨진 의존성
type OrderService struct{}

func (s *OrderService) CreateOrder(userID string) (*Order, error) {
    user := new(User) // User 가 어디서 오는가?
    user.ID = userID // User 가 어떤 데이터를 갖는가?
    // ... 숨겨진 복잡성
}

// ☑ 좋음: 명시적 의존성
type OrderService struct {
    userFinder UserFinder
}

func (s *OrderService) CreateOrder(userID string) (*Order, error) {
    user, err := s.userFinder.FindByID(userID)
    if err != nil {
        return nil, err
    }
    // ... 명확한 흐름
}
```

왜 이 규칙인가? - `new()`는 의존성을 숨김 → Missing 발견 안 됨 - `new()`는 System 1의 탈출구 → “그냥 만들어!” - 명시적 의존성 → 완전한 서비스 체인

규칙 2: Static 필드나 메서드 금지

```

// ✗ 나쁨: 전역 상태
var globalConfig *Config // 누가 설정하나? 언제?

func GetUser(id string) *User {
    db := globalDB // 숨겨진 의존성
    // ...
}

// ☑ 좋음: 주입된 의존성
type UserFinder struct {
    db     Database
    config Config
}

func (f *UserFinder) FindByID(id string) (*User, error) {
    // 모든 의존성이 보임
}

```

왜 이 규칙인가? - Static = 전역 상태 = 숨겨진 연결 - 전역 상태는 격리를 깨뜨림 - 전역 상태 변경은 모든 것에 예측 불가능하게 영향

규칙 3: 구현 가정 금지

```

// ✗ 나쁨: 구현 가정
type PaymentService struct {
    // Stripe 가 항상 사용된다고 가정
}

func (s *PaymentService) Process(amount int) {
    stripe.Charge(amount) // PayPal 로 바꾸면?
}

// ☑ 좋음: 인터페이스 기반
type PaymentGateway interface {
    Charge(amount int) error
}

type PaymentService struct {
    gateway PaymentGateway // Stripe 은, PayPal 이든, 뭐든 될 수 있음
}

```

```
func (s *PaymentService) Process(amount int) error {
    return s.gateway.Charge(amount)
}
```

왜 이 규칙인가? - 가정은 놀람이 됨 - 놀람은 System 1 을 트리거함 - 인터페이스는 안전한
변경을 허용

서비스 체인 만드는 법

Step 1: 요구사항으로 시작

요구사항: "사용자가 이메일과 비밀번호로 로그인할 수 있다"

Step 2: "무엇이 일어나야 하는가?" 물기

1. 입력 형식 검증
2. 이메일로 사용자 찾기
3. 비밀번호 확인
4. 계정 상태 확인
5. 세션 생성
6. 토큰 반환

Step 3: 각 단계를 서비스로 변환

```
ValidateCredentialsFormat(email, password) → ValidationResult
FindUserByEmail(email) → User
VerifyPassword(user, password) → bool
CheckAccountStatus(user) → AccountStatus
CreateSession(user) → Session
GenerateToken(session) → Token
```

Step 4: 빠진 요소 추가

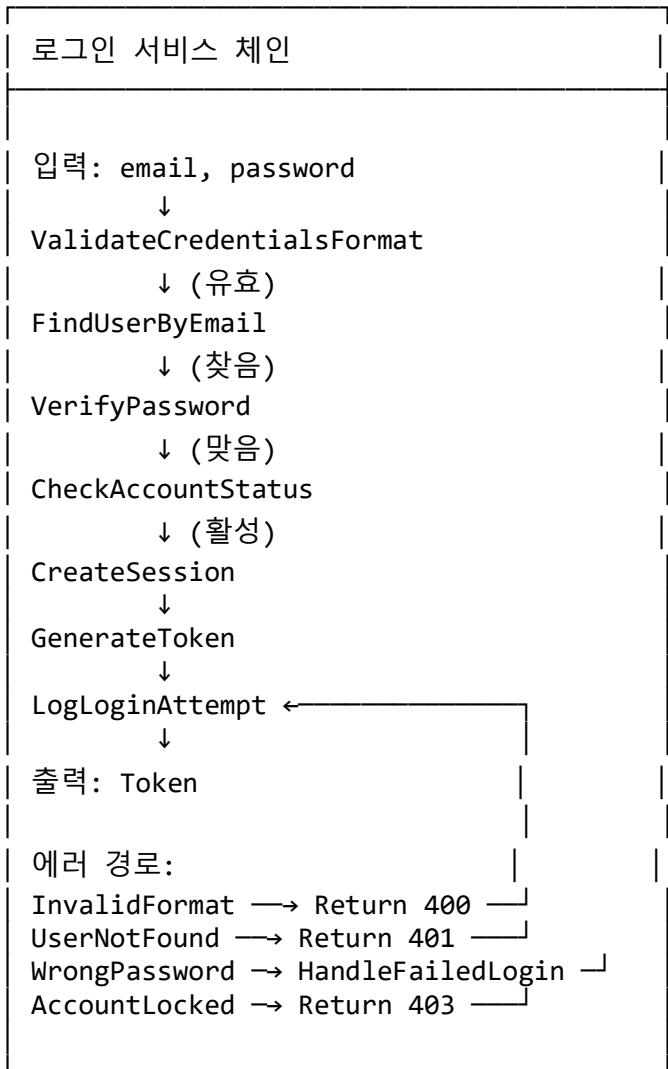
각 서비스에 대해 물기: - “입력이 어디서 오는가?” → 불명확하면 서비스 추가 - “출력이
어디로 가는가?” → 불명확하면 서비스 추가 - “실패하면 어떻게?” → 에러 처리 서비스 추가
- “누가 알아야 하는가?” → 알림/로깅 서비스 추가

추가된 것:

```
LogLoginAttempt(email, success, timestamp) → void
```

```
HandleFailedLogin(email, attemptCount) → void  
NotifySecurityTeam(suspiciousActivity) → void
```

Step 5: 완전한 체인 그리기



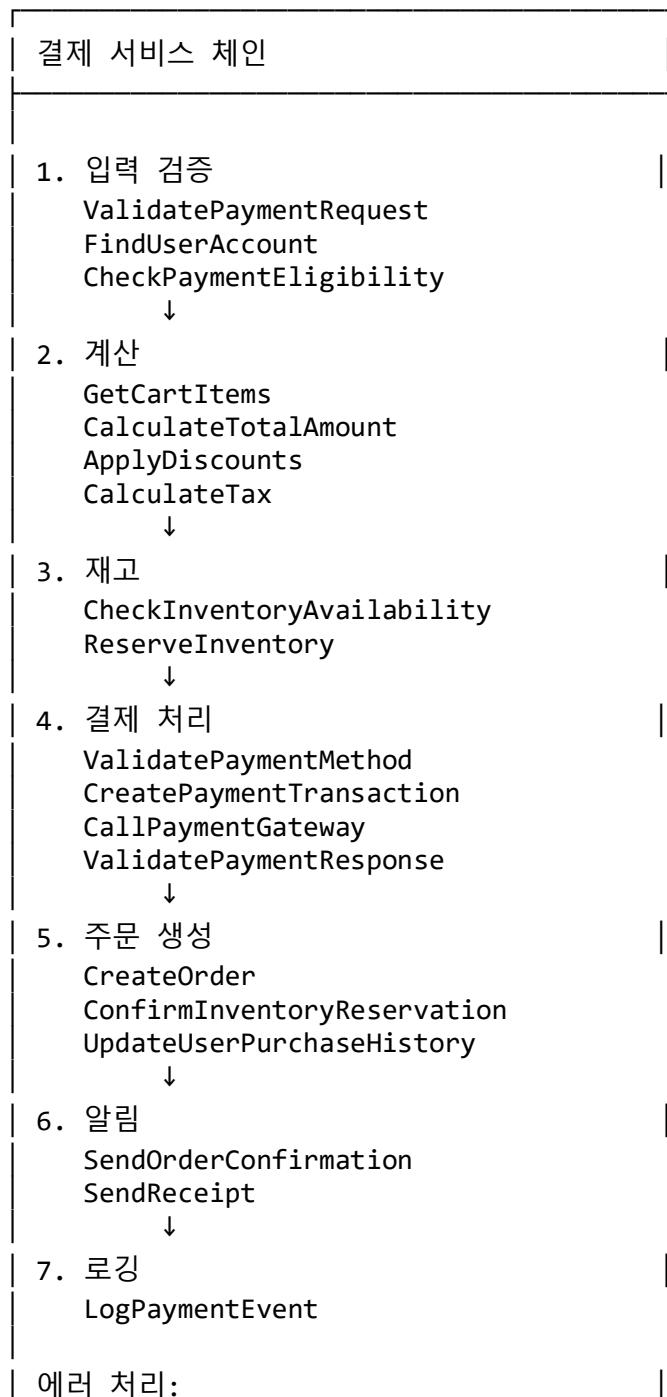
Step 6: 완전성 확인

체인을 통과하는 각 경로에 대해: - 빠진 정보 없이 완료될 수 있는가? → YES = 완전 - 어떤 서비스가 체인에 없는 것을 필요로 하는가? → 추가

예제: 결제 시스템

요구사항: “사용자가 장바구니의 상품을 구매할 수 있다”

서비스 체인:



`ReleaseInventoryReservation`
`RefundPayment`
`NotifyPaymentFailure`

핵심 설계 결정:

1. 결제 전 예약: 결제 실패 시 예약 해제
2. 결제 후 확정: 결제 성공 시에만 확정
3. 로깅 분리: 성공/실패 관계없이 항상 로깅

ROD 체크리스트

설계 단계:

- 모든 요구사항이 서비스 체인에 매핑됐는가?
- 각 서비스가 단일 책임을 갖는가?
- 모든 의존성이 명시적인가 (`new`/`static` 없이)?
- 모든 인터페이스가 정의됐는가?
- 여러 경로가 포함됐는가?

검증:

- 각 요구사항이 체인만으로 완료될 수 있는가?
- 목록에 없는 의존성이 필요한 서비스가 없는가?
- 구현에 대한 가정이 없는가?

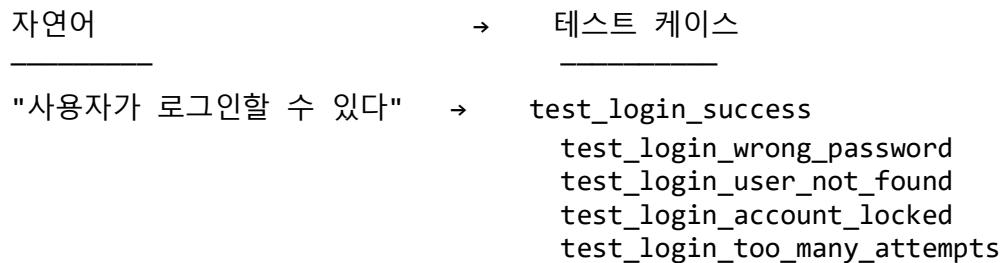
완료 징후:

- 구현이 "그냥 체인을 따라가는 것"처럼 느껴짐
- "이게 어디서 오지?" 질문 없음
- 코딩 중 "뭔가 추가해야 해" 없음

Part 2: TFD 실전

요구사항에서 테스트로

변환:



Step 1: 행복 경로 식별

요구사항: "사용자가 로그인할 수 있다"

행복 경로:

- Given: 유효한 이메일, 올바른 비밀번호, 활성 계정
- When: 로그인 시도
- Then: 유효한 토큰 반환

Step 2: 실패 케이스 식별

묻기: “뭐가 잘못될 수 있는가?”

- 잘못된 이메일 형식
- 등록되지 않은 이메일
- 틀린 비밀번호
- 잠긴 계정
- 미인증 계정
- 너무 많은 실패 시도
- 시스템 오류 (DB 다운)

Step 3: 엣지 케이스 식별

묻기: “경계값은?”

- 빈 이메일
- 빈 비밀번호
- 매우 긴 이메일 (1000 자)
- 특수문자가 있는 비밀번호
- 이메일의 유니코드

- SQL 인젝션 시도
- 동시 로그인 시도

Step 4: 테스트 명세 작성

```
func TestLoginService(t *testing.T) {
    // 행복 경로
    t.Run("ValidCredentials_ReturnsToken", func(t *testing.T) {
        // Given: 유효한 이메일, 올바른 비밀번호, 활성 계정
        // When: Login(email, password)
        // Then: 토큰 반환, 에러 없음
    })

    // 실패 케이스
    t.Run("InvalidEmailFormat_ReturnsError", func(t *testing.T) {
        // Given: 잘못된 형식의 이메일
        // When: Login(email, password)
        // Then: 검증 에러 반환
    })

    t.Run("WrongPassword_ReturnsError", func(t *testing.T) {
        // Given: 유효한 이메일, 틀린 비밀번호
        // When: Login(email, password)
        // Then: 인증 에러 반환, 실패 횟수 증가
    })

    t.Run("AccountLocked_ReturnsError", func(t *testing.T) {
        // Given: 유효한 자격증명, 잠긴 계정
        // When: Login(email, password)
        // Then: 잠금 에러 반환
    })

    t.Run("ThreeFailedAttempts_LocksAccount", func(t *testing.T) {
        // Given: 유효한 이메일, 틀린 비밀번호
        // When: 3 번 로그인 실패
        // Then: 계정 잠김
    })

    // 엣지 케이스
    t.Run("EmptyEmail_ReturnsError", func(t *testing.T) {
```

```

    // Given: 빈 이메일
    // When: Login("", password)
    // Then: 검증 에러 반환
  })

t.Run("VeryLongEmail_HandledCorrectly", func(t *testing.T) {
    // Given: 1000 자 이메일
    // When: Login(longEmail, password)
    // Then: 검증 에러 반환 (최대 길이 초과)
  })
}

```

테스트 구조: Arrange-Act-Assert

```

func TestVerifyPassword_CorrectPassword_ReturnsTrue(t *testing.T) {
    // Arrange: 전제조건 설정
    hasher := NewBcryptHasher()
    storedHash := hasher.Hash("correctPassword123")
    service := NewPasswordVerifier(hasher)

    // Act: 동작 실행
    result, err := service.Verify("correctPassword123", storedHash)

    // Assert: 결과 확인
    if err != nil {
        t.Fatalf("에러 없어야 하는데 %v 받음", err)
    }
    if !result {
        t.Error("true 여야 하는데 false 받음")
    }
}

```

체인의 각 서비스 테스트하기

ROD 체인 → TFD 테스트

서비스 체인:

ValidateCredentialsFormat

테스트:

test_valid_format
test_empty_email

		<code>test_invalid_email</code> <code>test_empty_password</code> <code>test_password_too_short</code>
<code>FindUserByEmail</code>	→	<code>test_user_found</code> <code>test_user_not_found</code> <code>test_db_error</code>
<code>VerifyPassword</code>	→	<code>test_correct_password</code> <code>test_wrong_password</code> <code>test_hash_error</code>
<code>CheckAccountStatus</code>	→	<code>test_active_account</code> <code>test_locked_account</code> <code>test_unverified_account</code> <code>test_deleted_account</code>
<code>CreateSession</code>	→	<code>test_session_created</code> <code>test_session_has_expiry</code> <code>test_session_stored</code>
<code>GenerateToken</code>	→	<code>test_token_generated</code> <code>test_token_contains_claims</code> <code>test_token_valid_signature</code>

TFD 체크리스트

테스트 설계:

- 행복 경로 테스트됐는가?
- 모든 실패 케이스 식별되고 테스트됐는가?
- 엣지 케이스 커버됐는가?
- 에러 메시지 검증됐는가?

테스트 품질:

- 각 테스트가 단일 목적을 갖는가?
- 테스트가 독립적인가 (순서 의존 없음)?
- 테스트가 Arrange-Act-Assert 구조를 사용하는가?
- 테스트 이름이 동작을 설명하는가?

커버리지:

- 각 ROD 서비스가 테스트를 갖는가?

- 모든 공개 메서드가 테스트됐는가?
- 통합 테스트가 존재하는가?
- 커버리지 > 80%?

유지보수:

- 테스트가 구현 상세에 의존하지 않는가?
 - 테스트가 문서 역할을 하는가?
 - 실패한 테스트가 문제를 명확히 가리키는가?
-

Part 3: DGTF 실전

5 단계 워크플로우

Step 1: 인식

“지금 서두르고 있나?”

자기 점검: - 심박수 증가? - “빨리, 빨리” 생각 반복? - 테스트 건너뛰고 싶은가? - “일단 되게 만들자” 압박 느끼는가?

YES라면 → System 1 영역에 있다. Step 2로.

Step 2: 멈춤

“잠깐. 멈추자.”

행동: - 키보드에서 손 떼기 - 심호흡 3 번 - 5 까지 세기 - 묻기: “왜 서두르게 하는 거지?”

소요 시간: 5-10 초

Step 3: 확인

“시스템을 확인하자.”

질문: - ROD 설계 확인했나? - TFD 테스트 확인했나? - 이 변경이 다른 부분에 영향을 주나?
- 뭐가 잘못될 수 있나?

소요 시간: 1-5 분

Step 4: 계획

“단계를 계획하자.”

적기: 1. 정확히 뭘 바꿀 건가? 2. 어떤 테스트가 통과해야 하나? 3. 동작 여부를 어떻게
검증하나? 4. 막히면 누구에게 물어볼 건가?

소요 시간: 2-5 분

Step 5: 실행

“신중하게 진행하자.”

방법: - 한 번에 작은 변경 하나 - 각 변경 후 테스트 실행 - 예상 밖의 일이 생기면 → Step
2로 돌아가기 - 자주 커밋

실제 시나리오: “이 버그 지금 당장 고쳐!”

상황:

금요일 오후 4 시 30 분

매니저: “프로덕션 망가졌어! 사용자들 결제 못 해!”

나: (심박수 증가, System 1 활성화)

DGTF 없이:

- 코드로 뛰어들기
- 관련된 것처럼 보이는 거 찾기
- 바꾸기
- 배포

→ 다른 버그 나타남

→ 더 패닉

→ 주말 망침

DGTF 로:

Step 1: 인식

"패닉하고 있다. System 1이 지배하려 한다."

Step 2: 멈춤 (10 초)

심호흡. "좋아, 실제로 뭘 알고 있지?"

Step 3: 확인 (3 분)

- 에러 로그 확인: "PaymentGateway timeout"
- 모니터링 확인: 30 분 전 시작
- 최근 배포 확인: 오늘 없음
- 외부 상태 확인: 결제 업체 다운

Step 4: 계획 (2 분)

- 이건 외부 문제, 우리 버그 아님
- 옵션:
 - a) 업체 기다리기 (30 분이라고 함)
 - b) 백업 업체로 전환
 - c) 사용자에게 "나중에 재시도" 활성화
- 결정: 재시도 활성화, 사용자에게 알림

Step 5: 실행 (10 분)

- 재시도 메커니즘 활성화 (미리 만들어둔 것, 테스트됨)
- 사용자 대상 메시지 추가
- 모니터링
- 매니저에게 타임라인과 함께 알림

총 시간: ~15 분

결과: 전문적으로 처리, 패닉 변경 없음

커뮤니케이션 템플릿

매니저가 “얼마나 걸려?” 물을 때

✗ System 1 응답:

"오늘 끝까지 끝낼게요!"

(분석 없음, 아마 실패)

✓ DGTF 응답:

"범위 확인하고 정확한 추정 드릴게요."

[30 분 후]

"요청 분석했습니다:

- 3 개 서비스 건드림
- 5 개 새 테스트 케이스 필요
- 추정: 2 일

급하게 하면 위험:

- 사용자에게 영향 주는 버그
- 나중에 고치는 데 더 많은 시간

권장 타임라인: 적절한 테스트와 함께 2 일."

테스트 건너뛰라고 할 때

✗ System 1 응답:

"네, 빨리 가려고 테스트 건너뛸게요"

(기술 부채, 미래의 고통)

✓ DGTF 응답:

"급한 거 이해합니다. 트레이드오프 설명드릴게요:"

테스트로 (2 일):

- 95% 신뢰도로 동작
- 주말 긴급사태 없음
- 나중에 유지보수 가능

테스트 없이 (1 일):

- 50% 버그 확률
- 주말 수정 필요할 가능성 높음
- 향후 변경 위험

2 일 접근법 권장합니다.

하지만 1 일 내로 배포해야 한다면,
위험 문서화하고 즉시 후속 테스팅
계획 세워야 합니다."

피해야 할 안티패턴

안티패턴 1: "빠른 수정 하나만"

증상: "이 한 줄만 바꿀게..."

현실: 한 줄이 열 줄이 됨, 테스트 없이

결과: 버그 배포, 신뢰 약함

해결: "한 줄"도 5 단계 적용

안티패턴 2: "나중에 테스트 추가할게"

증상: "먼저 동작하게 만들고, 그 다음 테스트"

현실: "나중"은 오지 않음

결과: 테스트 없는 코드가 프로덕션에

해결: 코딩 전에 테스트 명세 작성

안티패턴 3: "내 컴퓨터에서는 동작해"

증상: "로컬에서 테스트 통과, 배포해"

현실: 환경 차이로 실패

결과: 프로덕션 버그

해결: 일관된 환경의 CI/CD

안티패턴 4: "마감이 모든 걸 정당화해"

증상: "오늘 무조건 배포해야 해, 뭐가 됐든"

현실: 망가진 코드 배포는 자연보다 나쁨

결과: 긴급 수정, 신뢰 잃음, 더 많은 자연

해결: 품질이 아니라 범위를 협상

DGTF 체크리스트

작업 시작 전:

- 차분한가? (아니면 먼저 멈춤)
- ROD 설계 있는가?
- TFD 테스트 있는가?
- "완료"가 뭘 의미하는지 이해했는가?

작업 중:

- 서비스 체인을 따르고 있는가?
- 각 변경 후 테스트 실행하는가?
- 자주 커밋하는가?
- 서두르는 느낌인가? (그렇다면 멈춤)

압박받을 때:

- 응답 전 멈췄는가?
- 정직한 추정을 줬는가?
- 트레이드오프를 설명했는가?
- 결정을 문서화했는가?

끝날 때:

- 모든 테스트 통과?
 - 코드 리뷰 받음?
 - 문서 업데이트?
 - 다음 사람이 유지보수할 준비 됨?
-

Part 4: 완전한 예제 - 이커머스 장바구니

요구사항

“사용자가 장바구니에 상품 추가, 장바구니 보기, 결제할 수 있다”

Step 1: ROD - 서비스 체인 구축

하위 요구사항 1: 장바구니에 추가

AddToCartService 체인:

- ValidateAddToCartRequest
- FindUser
- FindProduct
- CheckProductAvailability
- GetOrCreateCart
- AddItemToCart
- CalculateCartTotal
- SaveCart
- LogCartEvent

하위 요구사항 2: 장바구니 보기

ViewCartService 체인:

- ValidateViewCartRequest
- FindUser
- FindCart
- GetCartItems
- EnrichCartItems (상품 상세 추가)
- CalculateCartTotal
- ApplyPromotions
- ReturnCartView

하위 요구사항 3: 결제

```
CheckoutService 체인:  
└── ValidateCheckoutRequest  
└── FindUser  
└── FindCart  
└── ValidateCartItems (아직 가능?)  
└── CalculateTotal  
└── ApplyDiscounts  
└── CalculateTax  
└── CalculateShipping  
└── ReserveInventory  
└── ProcessPayment  
    └── ValidatePaymentMethod  
    └── ChargePayment  
    └── HandlePaymentResult  
└── CreateOrder  
└── ConfirmInventoryReduction  
└── ClearCart  
└── SendConfirmationEmail  
└── LogCheckoutEvent
```

Step 2: TFD - 테스트 설계

AddItemToCart 테스트:

```
func TestAddItemToCart(t *testing.T) {  
    // 행복 경로  
    t.Run("ValidItem_AddsToCart", func(t *testing.T) {})  
    t.Run("ItemAlreadyInCart_IncreasesQuantity", func(t *testing.T) {})  
  
    // 실패 케이스  
    t.Run("ProductNotFound_ReturnsError", func(t *testing.T) {})  
    t.Run("ProductOutOfStock_ReturnsError", func(t *testing.T) {})  
    t.Run("UserNotFound_ReturnsError", func(t *testing.T) {})  
    t.Run("InvalidQuantity_ReturnsError", func(t *testing.T) {})  
  
    // 엣지 케이스  
    t.Run("QuantityExceedsStock_ReturnsError", func(t *testing.T) {})  
    t.Run("MaxCartItems_ReturnsError", func(t *testing.T) {})  
}
```

결과

소요 시간: 4 일

테스트: 25 개 통과

커버리지: 87%

프로덕션 버그: 0

급하게 했을 때 (추정):

소요 시간: 2 일 코딩 + 3 일 수정

테스트: 5 개 (버그 후 추가)

커버리지: 40%

프로덕션 버그: 8 개

고객 불만: 12 건

망친 주말: 예

Part 5: 주니어 개발자를 위해

어디서 시작할까

1-2 주차: 문제 이해

방법론부터 시작하지 마라. 왜 좋은 실천이 존재하는지 이해부터.

연습: 1. “빠른 수정” 코드를 작성했던 때를 떠올려봐 2. 그 다음에 뭐가 일어났어? 3. 고치는 데 얼마나 시간 썼어?

이것이 DGTF++가 존재하는 이유다.

1 개월차: DGTF 부터 시작

ROD 나 TFD 배우기 전에, 서두름을 인식하는 법 배워.

매일 연습: - “빨리” 느낄 때마다 5 초 멈춤 - 묻기: “왜 서두르게 하는 거지?” - 적어두기

이것이 인식 습관을 만든다.

2 개월차: TFD 추가

코드 전에 테스트 쓰기 시작.

매일 연습: - 함수 작성 전에 테스트 하나 쓰기 - 딱 하나. 완벽한 커버리지 아니어도 됨. - 습관 만들기.

3 개월차: ROD 추가

서비스 체인에 대해 생각 시작.

매일 연습: - 코딩 전에 서비스 스케치 - 종이에 박스와 화살표만 - 묻기: “뭐가 빠졌지?”

Part 6: 성공 측정

개인 지표

DGTF++ 전:

- 기능당 도입 버그: 5-10 개
- 디버깅 시간: 40%
- 주말 작업: 빈번
- 배포 시 자신감: 낮음

DGTF++ 후 (3 개월):

- 기능당 도입 버그: 1-2 개
- 디버깅 시간: 15%
- 주말 작업: 드물
- 배포 시 자신감: 높음

진전의 징후

2 주차: - 서두를 때 알아챔 - 가끔 멈춤 (항상은 아님)

1 개월: - 멈춤이 습관이 됨 - 코드 전에 일부 테스트 작성 - “긴급” 수정 줄어듦

3 개월: - 서비스 체인이 자연스러움 - 테스트가 자동적인 워크플로우 일부 - 동료들이 코드 품질 알아챔

6 개월: - 다른 사람들이 어떻게 차분한지 물어봄 - 왜 이렇게 일하는지 설명할 수 있음 - 주니어 개발자들이 배우려 함

Part 7: FAQ & 어려운 질문들

Q: “선행 시간이 너무 오래 걸려”

A: 총 시간을 비교해보자:

DGTF++ 없이:

- 설계: 0.5 일
- 코드: 2 일
- 디버그: 3 일
- 프로덕션 버그 수정: 2 일
- 총: 7.5 일

DGTF++로:

- ROD: 0.5 일
- TFD: 0.5 일
- 코드: 2 일
- 디버그: 0.5 일
- 프로덕션 버그: 0
- 총: 3.5 일

선행 시간이 그 이상을 절약한다.

Q: “매니저가 당장 결과를 원해”

A: 비즈니스 용어로 소통하라:

“테스트로 2 일 또는 테스트 없이 1 일에 배포할 수 있습니다.”

테스트 없이:
- 50% 프로덕션 버그 확률 - 버그 수정에 1-2 일 걸림 - 고객 영향: 부정적 리뷰
- 예상 총 시간: 3 일

테스트로:
- 95% 신뢰도로 동작 - 최소 고객 영향 - 예상 총 시간: 2 일

어떤 게 낫나요?”

Q: “레거시 코드에 어떻게 적용해?”

A: 점진적으로.

1 개월: 새 기능만

- 새 코드에 ROD + TFD 적용
- 레거시 건드리지 않음
- 팀 스킬 향상

2-3 개월: 보이스카우트 규칙

- 레거시 수정 시 테스트 추가
- 작은 개선만
- 재작성 안 함

4-6 개월: 전략적 개선

- 가장 위험한 모듈 식별
- ROD 재설계 적용
- 포괄적 테스트

절대 안 됨: 빅뱅 재작성

- 너무 위험
- 너무 비쌈
- 보통 실패

어려운 질문들 (악마의 변호인)

완벽한 방법론은 없으며, 솔직한 비판은 이해를 깊게 한다. 이 질문들은 실제 한계와 우려를 다룬다. 회의적이라면 그래야 한다—회의주의는 System 2 가 제대로 작동하고 있다는 증거다.

Q1: “System 1 이 활성화됐을 때 어떻게 System 1 을 인식하나?”

역설: DGTF 1 단계는 “서두르고 있음을 인식하라”고 한다. 하지만 System 1 의 본질은 무의식적이다. 진짜 패닉 상태에서는 자신이 패닉인지 모른다. “나 지금 괜찮아”라는 생각이 가장 위험할 수 있다.

솔직한 답:

그 순간에는 종종 인식할 수 없다. 이것은 진짜 한계다. 그래서 DGTF 가 강조하는 것:

1. 자기 인식이 아닌 외부 트리거

- 마감 발표 → 자동 멈춤
- 버그 리포트 → 자동 멈춤
- 메시지에 “급함” → 자동 멈춤

2. 의지력이 아닌 습관

- 패닉 인식에 의존하지 않음
- 패닉 전에 발동하는 반사 구축

3. 환경 설계

- 테스트 강제하는 pre-commit 혹
- 리뷰 강제하는 PR 요구사항
- 물리적 단서 (포스트잇, 타이머)

목표는 완벽한 자기 인식이 아니다. 그것이 필요한 상황을 줄이는 것이다.

Q2: “ego depletion 은? 의지력은 고갈된다.”

현실: 자기 통제는 유한한 자원이다. 하루 끝, 일주일 끝, 프로젝트 끝—의지력이 바닥났을 때 DGTF 를 어떻게 실천하나?

솔직한 답:

이것은 진짜 한계다. DGTF 는 피로를 해결하지 못한다. 하지만:

1. 의지력 의존 줄이기

- DGTF 를 자동으로 만들기 (습관)
- 외부 강제 기능 사용
- 고갈되면 일 멈추기—진지하게

2. 지속 가능한 속도

- DGTF 는 “과로하지 마”를 포함

- 주 60 시간은 DGTF 를 불가능하게 만듦
- 이건 조직 실패의 증상

3. 불완전함 수용

- 지쳤을 때 DGTF 실패할 것
- 그건 정보다: 너무 피곤함
- 집에 가라

Q3: “환경이 독성적이면?”

현실: “천천히 신중하게”가 해고 사유가 될 수 있다. 일부 조직은 품질을 벌한다.

솔직한 답:

DGTF 는 최소한의 조직적 합리성을 가정한다. 진짜 독성 환경에서:

1. 단기: 생존

- 안전한 곳에서 DGTF 적용
- 품질 작업 문서화
- 포트폴리오 구축

2. 중기: 결정

- 이 환경이 바꿀 수 있는가?
- 품질을 원하는 동맹이 있나?
- 당신의 영향력은?

3. 장기: 탈출 또는 변화

- 독성 환경은 스스로 고쳐지지 않음
- 당신의 커리어는 이 직장보다 김
- DGTF 스킬은 더 좋은 곳으로 이전됨

DGTF 가 해결할 수 없는 것: 체계적 조직 기능 장애. 개인 방법론으로 망가진 회사를 고칠 수는 없다. 이것은 방법론의 실패가 아니라 범위의 인식이다.

일부 환경에서는 DGTF 가 외부적으로 보상받지 못할 수 있다. 그러나 개인적으로는 여전히 혜택을 받는다: 덜한 스트레스, 깨끗한 코드, 더 나은 스킬, 그리고 다음 역할을 위한 강력한 포트폴리오.

Q4: “DGTF 는 검증 불가능. 잘하고 있는지 어떻게 아나?”

현실: TFD 는 pass/fail 이 있다. ROD 는 “Missing 발견”이 있다. DGTF 는 내면 상태를 측정한다—측정 불가.

솔직한 답:

직접 측정은 불가능하다. 대리 지표 사용:

DGTF 대리 지표:

- 기능당 버그 (감소해야 함)
- “긴급” 수정 (감소해야 함)
- 코드 리뷰 반복 (감소해야 함)
- 개인 스트레스 수준 (감소해야 함)
- 후회스러운 커밋 (감소해야 함)

또한: **DGTF 실패는 눈에 보인다.** 건너뛰면 결과가 나타난다. 버그가 발생한다. 재작업이 생긴다. DGTF 의 부재는 측정 가능하다.

Q5: “선순환은 합리적인 매니저를 가정한다”

현실: DGTF → 신뢰 → 자율성 → 압박 감소 → DGTF 쉬워짐... 매니저가 품질을 보상한다고 가정한다. 많은 매니저가 그렇지 않다.

솔직한 답:

맞다, 이 순환은 최소한 합리적인 관리를 필요로 한다. 매니저가: - 품질을 벌하고 - 속도만 보상하고 - 결과를 무시하면

순환이 끊어진다. 옵션: 1. 다른 관리 찾기 2. 그들의 생각을 바꿀 증거 쌓기 3. DGTF 가 외부적으로 보상받지 못함을 수용—여전히 당신의 성장에는 가치있음

모든 환경이 품질 관행을 지원하지는 않는다. 하지만 당신이 쌓은 스킬은 당신의 것으로 남는다.

Q6: “ROD 는 경험이 필요하다. 주니어는 Missing 을 못 찾는다.”

현실: “완전히 설계”는 뭐가 빠졌는지 알아야 한다. 주니어는 빠진 요소를 예측할 경험이 없다.

솔직한 답:

맞다. ROD 혼자는 시니어 의존적이다. 하지만:

1. 주니어는 패턴을 배울 수 있다

- 흔한 missing 요소는 문서화됨
- 여러 처리, 로깅, 검증, 정리
- 이것들은 예측 가능

2. 주니어는 멘토가 필요하다

- 시니어와 설계 리뷰
- “뭘 놓쳤나요?” 세션
- 시간이 지나며 Missing 학습

3. TFD 가 주니어를 돋는다

- 테스트가 Missing 을 더 일찍 드러냄
- 실패 피드백이 더 빠름
- 학습이 가속됨

ROD 는 주니어에게 더 어렵다. 그래서 DGTF++에 “주니어 개발자를 위한” 섹션이 있다—어려움을 인정하고 비계를 제공.

Q7: “급하게 느끼는 것과 진짜 급한 것을 어떻게 구분하나?”

현실: 프로덕션 다운은 돈이 듈다. 매 초가 중요하다. 선은 어디인가?

솔직한 답:

진짜 긴급 예시:

- 진행 중인 보안 침해 → 빨리 행동 (하지만 여전히 생각)
- 확산되는 데이터 손상 → 즉시 억제
- 초당 매출 손실 → 무자비하게 우선순위

가짜 긴급 예시:

- 매니저가 "ASAP" 말함 → 아마 생사 문제 아님
- 고객이 크게 불만 → 중요하지만 긴급 아님
- 마감이 내일 → 몇 주가 있었음; 계획 실패

진짜 긴급상황에서도 DGTF 적용—압축만 됨: - 5 초 멈춤이 잘못된 서버 종료를 막음 - 1 분

진단이 데이터 손실을 막음 - “빠른”은 “무생각”을 의미하지 않음

가장 위험한 환경(수술, 항공)에 멈춤과 체크리스트가 더 적은 게 아니라 더 많다.

Q8: “DGTF 는 개인적이다. 팀 압박은?”

현실: - 짹 프로그래밍 파트너가 “그냥 해” - 코드 리뷰에서 “왜 이렇게 오래 걸려?” - 스탠드업에서 진행 보고 압박

솔직한 답:

팀 DGTF 는 필요:

1. 팀 합의

- 팀으로 방법론 논의
- 표준에 합의
- “품질을 중시한다”를 공유 가치로

2. 명시적 소통

- “이것을 생각해보려고 멈추고 있어요”
- “제대로 하고 싶어요”
- 하는 것에 이름 붙이기

3. 압박 다루기

- “긴급함 이해해요. 한 가지만 확인할게요.”

- “더 빨리 갈 수 있지만 버그 있을 거예요. 결정하세요.”
- 트레이드오프 명시화

4. 문화 변화 (어려움)

- 한 사람이 행동 모델링 가능
- 결과가 결국 말해줌
- 항상 가능하지 않음

Q9: “DGTF 실패 시 복구 전략은?”

현실: 패닉했다. global variable 썼다. 테스트 건너뛰었다. 그 다음은?

솔직한 답:

DGTF 실패 복구:

1. 인정 (부정하지 않기)
 - “서둘렀네”
 - 부끄러움 없이, 인식만
2. 억제
 - 문제를 키우지 않기
 - 지금 서두르기 멈춤
 - 확산 방지
3. 평가
 - 뭘 건너뛰었나?
 - 피해는?
 - 위험은?
4. 수정 또는 수용
 - 지금 고칠 수 있으면: 제대로 고침
 - 아니면: 기술 부채로 문서화
 - 교정 계획
5. 학습

- 뭐가 서두름을 유발했나?
- 다음 번 어떻게 방지?
- 환경/습관 업데이트

핵심 통찰: DGTF 실패는 정보다. 환경, 상태, 또는 트리거에 대해 뭔가를 알려준다.
사용하라.

Q10: “이 문서 읽기가 System 1 을 발동시킨다”

메타 문제: DGTF 를 개념적으로 이해하면 배운 것처럼 느껴진다. “알았어, 다음!” 하지만
읽기는 System 1 이다. 실제로 배운 게 없다.

솔직한 답:

완전히 맞다. 읽기 ≠ 학습. 그래서 이 가이드가 강조하는 것:

1. 읽기가 아닌 연습

- “실행 계획” 섹션
- 주간 연습 목표
- 의도적 적용

2. 시간 요구사항

- “1 개월: 의식적 무능”
- “3 개월: 의식적 유능”
- “6 개월+: 무의식적 유능”

3. 교사로서의 실패

- DGTF 실패할 것
- 실패는 읽기가 가르칠 수 없는 것을 가르침
- 이것은 예상되고 필요함

이 가이드를 읽고 “이해했다”고 느끼는 것은 시작일 뿐이다. 진짜 이해는 실패한 시도, 교정,
수개월에 걸친 점진적 개선에서 온다. 문서는 지도이고, 영토를 걷는 것은 다르다.

Part 8: 실행 계획

1 주차: 이해

1-2 일차:

- 핵심 개념 문서 읽기
- System 1/2 이해
- 서비스 체인 이해

3-4 일차:

- 이 실전 가이드 읽기
- 공감되는 것 메모
- 연습할 기능 하나 선택

5 일차:

- 멘토/동료와 논의
- 답하기: "왜 이걸 시도하고 싶은가?"
- 4 주 목표 설정

2 주차: DGTf 연습

매일:

- 서두를 때 알아채기 (최소 3 번)
- 멈추기 (잠깐이라도)
- 서두르게 한 것 일지에 쓰기

주 끝:

- 일지 검토
- 공통 트리거 파악?
- 멈추기 쉬워지는가?

3 주차: TFD 연습

매일:

- 코드 전에 최소 1 개 테스트 쓰기
- 테스트 단순하게 유지
- 자주 테스트 실행

주 끝:

- 작성한 테스트 수 세기
- 이전 주와 버그 수 비교
- "전"과 "후" 테스트 느낌?

4 주차: ROD 연습

매일:

- 코딩 전 서비스 체인 스케치
- 박스와 화살표만
- 묻기 "뭐가 빠졌지?"

주 끝:

- 스케치 검토
- 체인이 혼란 방지했나?
- 구현 더 쉬웠나?

2-3 개월: 통합

매주:

- 세 가지 모두 하나의 기능에 적용
- 지표 추적 (버그, 시간, 자신감)
- 검토하고 조정

매월:

- 1 개월과 지표 비교
 - 팀에 배운 것 공유
 - 경험 기반으로 접근법 조정
-

마지막 노트

기억하라

DGTF++는:

- 완벽해지는 것이 아니다
- 느리게 하는 것이 아니다

- 규칙을 맹목적으로 따르는 것이 아니다

DGTF++는:

- 더 나은 결정을 내리는 것
- 지속 가능한 페이스
- 전문적인 결과

여정

- 1 일차: "이거 많은데"
- 1 주차: "서두르고 있는 거 알아챘어!"
- 1 개월: "오늘 테스트 먼저 썼어"
- 3 개월: "왜 진작 안 배웠지?"
- 6 개월: "이게 그냥 내 방식이야"

클릭할 때

DGTF++가 동작하고 있다는 걸 알게 되는 순간: - 압박 속에서도 차분함 - 코드가 디버깅 덜 필요함 - 동료들이 추정을 신뢰함 - 배포가 기대됨

행운을 빈다. 천천히 가라. 그게 포인트다.

저자 소개

김백면 (Bakmeon Kim)

소프트웨어 엔지니어 & 개발 철학 실천가

김백면은 25년 넘게 소프트웨어를 만들어왔다.

2000년에 개발자로 시작하여, 연구소장과 CTO를 거쳤다. 그의 경험은 기업 솔루션, AI 플랫폼, 스마트팩토리, AR/VR 애플리케이션, 보안 시스템에 걸쳐 있으며, 한국, 대만, 일본, 미국에서 프로젝트를 수행했다.

김백면을 차별화하는 것은 넓이보다 깊이에 대한 헌신이다. 개발자들이 보통 2-3년마다 이직하는 산업에서, 그는 한 회사에서 19년을 보냈다—만들고, 다듬고, 자신의 기술을 완성하면서. 15년 넘게 연구소장으로 팀을 이끌며 10년 이상 소프트웨어 공학 스터디를 진행했다.

그의 철학은 단순하지만 심오하다: **좋은 소프트웨어는 기능이 많은 소프트웨어가 아니라, 기능을 쉽게 추가할 수 있는 소프트웨어다.** 유지보수성에 대한 이 집중—그가 “변화 대응력”이라 부르는 것—이 그의 작업의 원동력이었다.

주요 경력

- 소프트웨어 개발 25년 이상
- 한 회사에서 19년 근무, 장기적 헌신의 증거
- 15년 이상 연구소장 역할 수행
- 10년 이상 소프트웨어 공학 스터디 리딩
- 여러 산업에 걸쳐 100개 이상 프로젝트 수행
- 대만, 일본, 미국 해외 프로젝트 경험

경험한 산업 분야

- SCM (공급망 관리) 시스템
- MES 기반 스마트팩토리 플랫폼
- AI 기반 수요예측/품질검사/예지보전

- AR/VR 산업용 메타버스
- AI 기반 보안 솔루션 (현재)

전문 분야

기업 솔루션:

- SCM (공급망 관리) 시스템
- MES (제조 실행 시스템)
- 스마트팩토리 플랫폼

AI & 신기술:

- 수요 예측을 위한 딥러닝
- 품질 검사를 위한 컴퓨터 비전
- LLM/sLLM 애플리케이션
- AR/VR 산업 메타버스 솔루션

소프트웨어 공학:

- 테스트 우선 개발 문화
- 클린 아키텍처와 디자인 패턴
- 팀 교육과 멘토링
- 개발 프로세스 최적화

현재 (2025~)

AI 기반 보안 솔루션을 개발 중이다. 네트워크 보안에 ROD, TFD, DGTF 원칙을 적용하고 있다.

AI 를 잘 다루면 3-4 명의 개발자와 함께 일하는 것과 유사한 생산성을 확보할 수 있다는 것을 직접 경험하고 있다. 하지만 이것이 DGTF 의 중요성을 줄이지 않는다—오히려 더 중요하게 만든다.

원칙의 시작

1992년 대학 입학 당시, 이미 프로그래밍을 할 줄 아는 동기들 사이에서 뒤처짐을 느꼈다.
그때 다짐한 두 가지:

“절대 생각함을 멈추지 말자” “절대 궁금해함을 멈추지 말자”

이 다짐이 30년 후 이 책의 기반이 되었다.

학력

- 컴퓨터공학 석사(소프트웨어 공학 전공) 숭실대학교, 서울 (2002) GPA: 4.0/4.0
- 컴퓨터공학 학사 숭실대학교, 서울 (1999)

철학

“같은 일을 두 번 하는 것을 싫어한다. 세 번 이상 해야 한다면, 자동화한다.”

“좋은 소프트웨어는 옵션이 많은 소프트웨어가 아니다. 옵션을 쉽게 추가할 수 있는 소프트웨어다.”

“생각을 멈추지 마라. 호기심을 멈추지 마라.”

대학 시절 확립되고 수십 년의 실천을 통해 다듬어진 이 원칙들이 이 책에서 소개하는 ROD, TFD, DGTF 방법론의 기반이다.

개인

김백면은 고등학교 영어 회화 동아리에서 만난 아내, 그리고 딸과 함께 서울에 살고 있다.

코딩하지 않을 때는:

- 집에서 커피 로스팅
- 클라리넷 연주 (2015년부터 배움)
- 지역 공방에서 목공
- 가죽 공예
- 3D 프린팅
- 보컬 레슨 (2023년부터)

- 방송대에서 심리학 공부

2013년에 담배를 끊었고, 지금은 가끔 딸과 술을 즐긴다.

연락처

- GitHub: <https://github.com/bakmeon/dont-go-too-fast>
 - Email: stillblueist@naver.com
-

이 책은 25년간의 배움, 실천, 가르침을 담고 있다. 이론만이 아니라, 지속되는 소프트웨어를 만드는 일상의 현실에서 쓰여졌다.

Remember: Don't Go Too Fast 