리팩토링의 규율

동작은 그대로, 형태는 계속 나아지는 기술

ThakiCloud (Hyojung Han)

리팩토링의 규율

동작은 그대로, 형태는 계속 나아지는 기술

ThakiCloud (Hyojung Han)

Contents

1 1장. 왜 지금 고치는가: 리팩토링의 시점 감각 1

2 2장. 무엇을 고치는가: 냄새의 지도 8

3 3장. 어떻게 고치는가: 작은 걸음의 안전 절차 15

4 4장. 계속 나아지게 하기: 1인 개발자의 습관 22

1 1장. 왜 지금 고치는가: 리팩토링의 시점 감각

이런 경험이 있을 것이다. 어떤 기능을 바꾸려다 파일을 열면, 읽는 시간이 쓰는 시간보다 길게 느껴진다. 조건을 쫓고 변수가 어디서 채워지는지 찾고 겨우 고칠 줄을 찾는다. 그리고 이렇게 생각한다. 이 구조를 먼저 고쳐놨으면 이 수정이 10분이 됐을 텐데.

이 책은 그런 순간에 무엇을, 어디까지 고칠지를 어떻게 판단하는지에 관한 것이다. 독자는 혼자 코드베이스를 오랫동안 유지보수하는 1인 개발자, 또는 레거시가 쌓이기 시작한 스타트업의 엔지니어로 가정한다. 자기가 지은 코드를 수년째 돌리면서 고칠 때마다 더 느려지는 감각을 알아채기 시작한 사람이면, 이 책이 쓴 대상이다.

1.1 먼저, 리팩토링이 무엇인가

리팩토링은 바깥쪽 동작을 조금도 바꾸지 않으면서 코드의 안쪽 형태를 좋게 만드는 작업이다. 바깥쪽은 사용자에게 보이는 전부다. API 응답, 화면 출력, 저장되는 데이터, 로그 줄, 예외 메시지. 안쪽은 자신만 보는 구조다. 함수의 쪼개기, 이름, 로직의 위치, 조건문의 쓰임.

규율이란 단어가 붙은 이유가 있다. 동작은 그대로라는 약속 때문이다. 동작을 살짝 바꾸는 리팩토링은 버그다. 그래서 리팩토링은 늘 검증 수단을 짝으로 삼는다. 테스트가 없으면, 적어도 동작 전후를 기록한 것. 이 전제는 책 전체에서 반복되니, 여기서 먼저 익혀 두길 바란다.

1.2 리팩토링, 버그 수정, 리와이트의 경계

이어붙인 활동 세 가지의 경계도 정리해 둔다. 버그 수정은 동작을 바꾼다. 결과가 틀렸고 이제 맞아야 한다는 것이 목표다. 리팩토링은 동작을 바꾸지 않는다. 형태가 더 고치기 쉬워지는 것이 목표다. 두 활동은 같은 파일을 열기에 겉으로 비슷해 보인다. 하지만 동작이 변하는 방향이라는 점에서 정반대다. 파일을 열다가 결과가 틀렸다면, 먼저 버그 수정을 한다. 동작이 맞는 상태가 된 뒤에야 형태를 논한다. 둘을 섞는 것이 리팩토링 사고의 원인이 되는 일이 많다.

리와이트는 동작을 보존하지 않고 새 코드로 바꾸는 일이다. 금지는 아니다. 다만 기준이 높다. 형태를 고치는 비용이 다시 짓는 비용보다 비싸질 때, 즉 작은 걸음으로 좋게 만들기가 더 이상 경제성이 없을 때다. 1인 개발자에게 다시 짓기의 위험은 검증 공백이다. 기존 동작이 맞았는지를 확인할 사람이 없으니까. 그래서 리와이트는 보통 모듈이 작고 입출력이 뚜렷할 때만 검토한다. 그 첫 단계도 역시 현재의 동작을 기록하는 일이다. 3장에서 이 기록법을 다룬다.

1.3 왜 시점이 기법보다 먼저인가

리팩토링 실패담의 상당수는 시점 탓이다. 실패하는 극단 둘을 이름 붙여 보자.

첫째는 대청소다. 코드가 너무 더러우니 한 주를 내줘서 다시 쓰자고 하는 계획. 1인 개발자에게 이 계획은 거의 성립하지 않는다. 문제의 정확한 경계를 모르다 보니 한 주라는 시간이 현실화되지 않는다. 그 한 주 동안 기능과 버그 수정은 못 하므로, 그 압박이 결국 계획을 포기하는 이유가 된다. 게다가 큰 폭의 수정에서는 동작이 그대로라는 검증을 도저히 못 해, 고친 건지 망가뜨린 건지 알 수 없는 상태로 끝난다.

생각보다 흔한 실패 장면 하나를 덧붙이자. 인증 모듈을 정리하겠다고

하루를 잡았다. 오전에 경계를 이해하려 코드를 다시 읽는다. 세션 로직 옆에 제3자 로그인이 붙어 있고, 그 부분은 동작을 가장 자신 없어하는 구역임을 오후에야 깨닫는다. 결국 변수 이름 몇 개만 바꾸고 끝났다. 다음 날 프로덕션에서 결제 버그가 터져, 계획했던 하루는 전부 날아갔다. 1인 개발자에게는 그 시간을 대신 채워줄 사람이 없다. 대청소 실패의 비용은 다음에는 그런 계획 자체를 믿지 않게 된다는 점이다.

둘째는 아무것도 하지 않는 것이다. 모든 수정이 마찰을 남긴다. 마찰은 쌓인다. 지난 주에는 30분이었던 수정이 이번 달에는 2시간, 그다음에는 하루가 된다. 이 상태를 안정된 레거시라고 부르는 사람이 있지만, 그것은 조용히 나빠지는 코드다.

답은 둘 사이의 어디쯤이다. 작게, 적합한 장소에서, 적합한 시간에. 그 장소와 시간을 어떻게 고르는지가 이 장의 주제다.

1.4 변경 빈도가 잣대다

이 책의 핵심 규칙을 먼저 말하겠다. 리팩토링의 우선순위는 코드가 얼마나 자주 바뀌는지로 정한다. 얼마나 추한지는 아니다.

추하지만 거의 안 바뀌는 코드를 고칠 가치는 낮다. 추함의 비용은 바꿀 때만 내니까, 안 바꾸면 비용도 안 낸다. 반대로, 약간 추하더라도 한 달에 5번은 만지는 코드는 핫스팟이다. 마찰을 한 달에 5번씩 내고 있는 것이다. 형태를 한 번 좋게 만들면 매월 5배의 회수가 들어오고, 계속 만지는 코드라면 그 회수가 복리로 붙는다.

숫자로 셈해 보자. 어떤 함수가 형태 탓에 바꿀 때마다 15분씩 더 읽어야 한다면, 그 함수를 한 달에 4번 고친다는 것은 매달 1시간을 쓰는 것이다. 30분을 들여 형태를 쉽게 만들면, 다음 달부터 매월 30분이 남는다. 회수 기간은 1개월이다. 그리고 2년 더 고친다면, 매월 3시간씩 24개월, 72시간의 순수 이익이다. 반면에 같은 추함인 함수가 3개월에 한 번만

바뀐다면, 한 번에 15분을 아끼는 셈이니까, 30분짜리 투자 회수에 8개월의 일자가 필요하다. 같은 추함이 위치에 따라 다른 가치가 된다.

운영에서 장애가 잦은 서비스를 오래되고 추하되 안정된 서비스보다 먼저 모니터링하는 것과 같은 논리다. 잣대는 경험 빈도다. 추함이 아니다.

변경 빈도를 재려면 git 히스토리를 쓴다. 예를 들어:

```
git log --since="6 months ago" --oneline -- src/billing
git log --since="6 months ago" --name-only --oneline | sort |
↪  uniq -c | sort -rn | head -20
```

두 번째 명령은 지난 6개월 커밋 가운데 가장 자주 바뀌었던 파일부터 나열한다. 그 목록 맨 위가 리팩토링 투자가 가야 할 곳이다. 그 파일을 어떤 관점으로 읽을지는 2장에서 상세히 다룬다.

1.5 시점 규칙 세 가지

지금 알맞은 시간을 구체적으로 만들자.

규칙 1, 두 번째 방문 규칙. 어떤 함수를 두 번째로 고치러 들어가는 데, 지난번에도 다시 이해하는 데 시간이 들었다면, 먼저 정리하자. 첫 번째 방문은 예외다. 앞으로 어떤 방향으로 바뀔지 아직 모르니까. 하지만 두 번째 방문은 정보를 준다. 이 함수는 살아 있고 계속 만져지고 또 다뤄질 것이다. 첫 번째 방문에 낸 마찰은 할부였다.

규칙 2, 곁모임 규칙. 기능 작업을 하다가 지나는 구역의 장애물을 그때 고친다. 예를 들어 폼에 한 필드를 추가하려다 두 파일을 건드려야 하고 그 중간에 의미 없는 이름의 변수가 보이면 그때 바로 이름부터 고친다. 그 순간에 코드를 가장 잘 알고 테스트를 막 돌린 직후이며 구역이 머릿속에 있으니 검증이 가장 싸다. 리팩토링은 기능 작업의 부산물이다.

규칙 3, 앞서가기 규칙. 형태 때문에 어려울 수정을 하려면, 먼저 형태를

쉽게 만들고 그다음 수정한다. 수정을 먼저 하고 정리를 뒤로 하면, 그 수정 자체가 마찰 속에서 일어나고 동작 검증의 전후 기준선이 흐려진다. 순서가 맞으면 수정과 정리가 두 개의 검증 가능한 단위로 나뉘게 된다.

세 규칙은 공통점을 가진다. 리팩토링을 구체적인 작업 상황에서 결정한다는 점. 그리고 함께 쓰인다. 두 번째로 들어가면 규칙 1의 상황이고 거기서 작은 장애물은 규칙 2로 치우고 큰 수정이 필요하면 규칙 3이 형태부터 쉽게 만들라고 한다.

구체적으로 보자. 결제 모듈에 월정액 옵션을 넣으라는 요청이 들어왔다. 2개월째 네 번째 방문이다. add_payment 함수를 열면 매번 다시 읽어야 했다. 규칙 1, 두 번째 방문이 넘었으니 정리 대상이다. 지나는 중간에 tmp라는 변수를 발견하고, pending_invoice로 바꾸면 2줄 일이다. 규칙 2, 곁모임이다. 그리고 월정액 로직 본체를 넣으려니, 세 군데의 조건문을 고쳐야 한다. 지금 넣으면 코드는 더 추해진다. 그래서 먼저 조건문 하나를 pricing_tier 함수로 빼낸다. 규칙 3, 앞서가기. 월정액 요청 자체는 그다음 단계에서, 깨끗해진 코드로 처리한다. 한 요청이 세 규칙을 자연스럽게 거쳤다.

1.6 바로 쓰는 판단 체크리스트

나쁘다고 생각되는 코드를 만났을 때, 망설이는 순간에 다음을 순서대로 자문하라.

1. 이 코드를 다음 3개월 안에 또 만질 것인가. 확신이 서면 그 답으로 충분하다. 확신이 안 서면 git 히스토리를 확인한다.
2. 현재의 마찰은 무엇인가. 한 문장으로 설명해 보라. X를 바꾸려면 먼저 Y를 읽어야 한다는 식으로. 설명이 안 되면 그것은 직관이며 직관의 판단력은 낮다.
3. 동작이 보호되어 있는가. 이 코드가 망가져도 걸러지는 테스트가 있는가. 없으면, 리팩토링 전에 현재 동작을 기록하는 테스트부터

써야 한다. 특화 테스트는 3장에서 자세히 다룬다.

4. 한 걸음으로 끝나는가. 머릿속의 수정이 며칠이 든다면, 그것은 프로젝트다. 대청소의 함정이 바로 앞에 있다. 한 시간 안에 커밋할 크기까지 줄여라.

네 개가 모두 좋으면 지금이 리팩토링할 때다. 하나라도 나쁘면, 답은 대개 지금이 아니다이지, 영원히 아니다가 아니다. 지금이 아님을 회피로 받아들이면 안 된다. 위치와 이유를 로그에 적어두면, 다음 방문 때 다시 올라온다. 그 로그를 쓰는 방식은 4장에서 다룬다.

리스트를 앞의 예에 대입해 보자. 80줄인 결제 함수, 지난 3개월에 11회 변경. 첫 번째 질문, 다음 3개월 안에 또 만질 것인가. 예, 월정액 기능이 대기 중이다. 두 번째, 마찰은 무엇인가. 요율을 바꾸려면 먼저 할인 로직을 읽어야 한다. 세 번째, 동작이 보호되어 있는가. 테스트는 있지만 정상 경로 두 건뿐, 약한 앵커다. 네 번째, 한 걸음으로 끝나는가. 함수를 통째로 쪼개는 것은 하루 일이고 할인율 계산만 빼내는 것은 15분 일이다. 결론: 지금 15분짜리 움직임부터 하고 나머지는 로그에 적는다. 이것이 이 책의 표준적 판단 모양이다.

1.7 손대지 말아야 할 경우

반대로, 고치고 싶어도 고치지 말아야 하는 경우도 나열하겠다.

데드 코드. 이미 대체된 기능, 더 이상 쓰이지 않는 설정. 만지지 않을 코드는 꾸미지 않는다. 대개 정답은 삭제다. 데드 코드 삭제는 1인 개발자에게 회수가 가장 높은 리팩토링이다. 다만 주의할 것은, 증거가 아니라는 점이다. 참조를 확인하거나, 확실해질 때까지 두고 본다.

동작을 검증할 수 없는 코드. 테스트가 없고 쓰기에도 어려우면, 외부 의존이 크거나 타이밍에 의존하는 로직일 때, 큰 폭의 리팩토링은 도박이다. 그런 곳에서는 가장 보수적인 움직임, 이름 바꾸기나 주석이나 작은 추출 정도와,

테스트 영역을 넓히는 일만 한다.

모양에 대한 불만. 구조가 조금만 다르다면 더 나아질 텐데,는 정당한 이유이지만 우선순위 목록에서는 맨 아래다. 코드가 추하지만 변경 빈도가 0이고 마찰이 낮다면, 얻을 아름다움은 자신에게만 보이는 것이다. 그 만족에는 대가가 있다. 내가 짊어지는 회귀 위험이다.

1.8 한 페이지 요약

첫 장은 세 줄로 압축된다. 리팩토링은 구체적인 작업 상황에서 결정되는 위험 관리 행위다. 잣대는 변경 빈도이지 추함이 아니다. 그리고 지금 알맞다는 것은, 이미 그 구역에 있거나 곧 들어가게 되며, 동작을 검증할 수 있을 때를 말한다.

다음 장에서는 무엇을 고치는가를 보자. git 히스토리로 핫스팟 목록을 손에 넣었으니, 그 안에서 코드의 어떤 형태 문제가 고칠 가치가 있는가. 1인 개발자가 실전으로 쓸 냄새 목록과, 우선순위를 재는 방법을 내어놓겠다.

2 2장. 무엇을 고치는가: 냄새의 지도

1장에서 변경 빈도라는 잣대로 언제를 정했다. 이제 무엇을 고치는가를 정한다. 핫스팟 파일을 열면, 어떤 형태의 문제를 골라야 하는가. 이 장에서는 1인 개발자가 실전으로 쓸 냄새 목록과, 우선순위를 재는 방법을 내어놓는다.

2.1 먼저, 경계: 버그와 냄새

버그는 동작이 틀린 경우. 냄새는 동작은 맞는데, 형태가 다음 수정을 어렵게 만드는 경우다. 리팩토링은 후자만 다룬다. 동작이 틀린 것을 찾았다면, 그것은 버그 수정이라는 다른 절차다. 먼저 동작을 맞게 하고 혹은 현재 동작은 이러니 바꾸지 않는다고 명시한 뒤에, 비로소 형태를 건드린다.

둘을 섞는 것이 리팩토링 사고의 가장 큰 원인이다. 형태를 고치려다 동작을 바꾸고 어느 것이 어느 것이었는지 몰라 차이를 설명하지 못한다. 그래서 코드를 읽는 첫 단계는 늘, 동작이 맞느냐와 형태가 좋으냐를 다른 두 판단으로 나누는 일이다.

테스트를 하나 해 보자. 최근 고친 버그를 떠올려라. 수정 자체에 쓴 시간과, 고칠 곳을 찾는 데 쓴 시간, 어느 쪽이 길었는가. 대개 유지보수자는 찾는 시간이 더 길다. 그 비율이 냄새의 비용이다. 냄새는 작업을 느리게 하지 않는다. 작업의 위치를 찾는 일을 느리게 한다.

2.2 1인 개발자가 가장 쓰는 다섯 가지 냄새

코드 냄새에 관한 책은 많다. 하지만 유지보수 시간을 쓰는 1인 개발자는 그 전부를 다 기억해 둘 수 없다. 아래는 실전에서 실제로 보이고 실제로 개선 효과가 있는 형태로 좁힌 다섯 가지다.

2.2.1 1. 여러 일을 하는 함수

검증, 계산, 저장, 통지를 한 함수가 순서대로 하는 경우. 이 함수는 공구를 네 개 섞어 넣은 상자다. 판단은 간단하다. 한 줄로 이름을 붙일 수 없다면, 하는 일이 많다. process나 handle이나 doWork로 이름 붙여진 함수는 대개 이 냄새다.

핫스팟에서 가장 자주 보이는 냄새이기도 하다. 이유는 기능 추가 때, 가장 손이 잘 가는 곳이 이미 보고 있던 함수이기 때문이다. 함수는 마지막에 고쳤던 곳이라는 방향으로 불어난다.

2.2.2 2. 중복된 로직

같은 형태가 두 군데 이상에 있다. 그러나 중복에도 두 단계가 있다. 똑같은 코드 중복은 우선순위가 높다. 한쪽에서 버그를 고치다, 다른쪽을 놓치기는 거의 예정되어 있다. 형태 중복, 즉 흐름은 같고 세부만 다른 경우는 우선순위가 낮다. 공통 함수로 빼내는 작업이 오히려 코드를 읽기 어려워 만들 수 있으니, 세 번째 사본이 뚜렷이 보일 때까지 기다려도 된다.

1인 개발자에게 중복이 특히 위험한 이유는, 첫 번째 사본을 쓴 사람과 두 번째 사본을 읽는 사람이 동일인이기 때문이다. 전에 썼으니까 대충 알겠다는 것은, 유지보수에서 가장 과대평가된 믿음이다.

2.2.3 3. 파도처럼 퍼지는 수정

논리적인 변화 하나가 파일 N개를 건드리게 만드는 경우. 예를 들어 결제 수단을 하나 추가하려다, 폼 컴포넌트, 검증 모듈, 결제 함수, 설정 파일, 테스트 픽스처, 5개 파일을 고치는 경우. 수정 하나 자체는 작고 안전해 보이지만 그 비용은 매번 납부되고 파일을 하나만 놓쳐도 버그가 생긴다.

찾는 방법은 이 구역의 최근 5개부터 10개 커밋이 어떤 파일들을 건드렸는지 보는 일이다. 비슷한 내용의 커밋이 겹치는 파일 무리를 계속 건드린다면,

그것이 파도다. 고칠 방향은, 함께 바뀌는 로직을 한 곳에 모으는 것.

2.2.4 4. 깊이 파인 조건문

if 안에 if, 그 안에 또 if. 파인 코드는 독자로 하여금 여기 지금 어떤 상황에 와 있다는 목록을 길게 따라오게 한다. 좋은 소식은, 이 냄새는 고치기 가장 쉬운 편에 든다는 점이다. 대개 조건을 뒤집어 미리 return하는 것만으로 형태가 평평해진다. 새로운 추상화가 필요 없고 위험이 낮으며 효과가 즉각적이다. 그래서 초보자의 첫 리팩토링에 잘 맞는다.

2.2.5 5. 마법 숫자와 문자열

계산 속에 떠 있는 0.07, 조건 속에 든 “RETRY_3” 같은 상태 문자열. 그 뜻이 아무데도 없다는 점이 문제의 핵심이다. if (amount > 100000)를 읽을 때, 100000이 기준인지 한도인지 임시 대응인지 모른다. 의미 있는 상수로 빼내는 것은 가장 보수적인 리팩토링이자, 한 칸, 검증, 커밋이라는 습관을 만들 훈련에 가장 좋다.

2.3 파일을 열기 전에 냄새 보는 법

파일을 열기 전에, 신호만으로 냄새를 볼 수 있는 길이 하나 있다.

첫 번째 신호는 파일 이름이다. util.py, helper.py, manager.py, common.js. 이런 이름은 대개 쓰레기장이다. 이름 자체가 뭔가 넣는 곳이라고 한다면, 로직의 집이 없다는 뜻이다. 두 번째 신호는 크기다. 줄 수 자체보다는, 내가 생각했던 줄수와 실제 줄수의 차이가 중요하다. 로그인이 몇 줄이어야 하는지 너는 잘 안다. 그것의 두 배, 세 배라면 냄새다. 세 번째 신호는 커밋 메시지다. 또 고침, 임시 처리, 여기만 건드리지 마. 이런 메시지는 코드가 마찰을 냈다는 증거다. 이 세 신호를 먼저 훑고 파일을 여는 순서로 읽으면 판단이 훨씬 빨라진다.

2.4 우선순위를 재는 질문 두 개

냄새 목록이 있어도 한꺼번에 고질 수는 없다. 질문 두 개가 순서를 정한다.

질문 1, 얼마나 자주 경험하는가. 냄새가 들어 있는 코드의 변경 빈도가 높을수록 우선순위는 높다. 1장에서 이미 핫스팟 목록을 만들었으니, 대조하면 된다.

질문 2, 한 번 고치면 얼마를 되돌려 받는가. 어떤 냄새는, 깊은 조건이나 마법 숫자처럼, 회수가 작지만 위험도 작다. 또 어떤 냄새는, 중복 로직이나 파도 수정처럼, 회수가 크고 앞으로의 변경 속도에 미치는 영향도 크다. 작고 안전한 수정을 자주 경험하는 구역에서 시작하고, 크고 효과적인 수정은 빈도가 낮은 구역에서는 뒤로 미룬다.

냄새	신호	우선순위
중복 로직	버그 수정이 2곳을 건드림	높음
여러 일을 하는 함수	한 줄로 이름 못 붙임	높음
파도 수정	커밋 1건에 파일 5개 이상	높음
깊은 조건	조건 3단계 이상	중간
마법 숫자	뜻이 이름에 없음	낮음에서 중간

표는 참고용이다. 최종 판단은 자신의 코드베이스의 실제 변경 빈도를 손에 넣은 상태에서 한다.

예를 pricing 사례에 돌려 보자. 중복된 형태(레벨 할인과 계절 할인)는 14회 변경된 파일에 있다. 경험 빈도가 높다. 한 번 고치면 되돌려 받을 것도 크다. 할인 케이스 하나를 추가하는 일이, 한 줄을 추가하는 일로 바뀌니까. 깊은 조건과 마법 숫자도 같은 파일 안에 있으니, 함께 우선순위가 오른다. 반면, 같은 정도로 추한 세션 모듈은 3개월에 2회 변경에 불과하다. 경험 빈도가 낮다. 모양이 같게 추해도, 우선순위는 뒤에 온다.

그래도 시작할 곳이 안 정해지면, 이렇게 고르라. 테스트가 가장 많으면서 변경 빈도가 가장 높은 파일부터. 앵커가 이미 있는 곳이라, 첫 걸음이 가장 안전하다. 그런 파일이 없다면, 최근 마찰을 가장 크게 낸 파일, 즉 지난주에 버그를 내고 고치던 파일이 대상이다. 그리고 먼저 특화 테스트를 쓴다. 시작 지점의 선택은 앵커의 위치를 확인하는 일이다.

2.5 3줄 수정 규칙

찾은 냄새 가운데 3줄이면 고칠 수 있는 것은, 백로그에 적지 말고 그 자리에서 고친다. 변수 이름 바꾸기, 상수 추출, 조기 return 추가. 이유는 경제다. 작은 과제를 목록에 적고 나중에 다시 읽고 맥락에 다시 들어가는 비용이, 그냥 하는 비용보다 큰 일이 자주 있다. 백로그는 3줄이 넘는 것을 위한 것이다.

그 반대 규율도 있다. 3줄이었던 수정이 중간에 30줄로 불어난다면, 멈춘다. 그때 그것은 더 이상 작은 수정이 아니다. 3장의 절차로 돌아가는 경우다. 이 규칙은, 조용히 대청소로 넘어가는 살짝 고치다 가드를 세워 준다.

3줄 규칙의 또 다른 이름은 유지보수와 프로젝트의 경계다. 3줄 이내는 유지보수이며 그 자리에서 하는 것이 의무다. 3줄을 넘으면 프로젝트고 프로젝트는 앵커와 작은 걸음과 검증이라는 3장의 절차가 필요하다. 대부분의 실패는 고치는 것 자체에서 나는 것이 아니다. 이것이 프로젝트인지 아닌지를 결정하는 지점에서 난다.

2.6 리팩토링 로그: 두 번째 기억

1인 개발자의 약점은 기억이다. 기술이 아니다. 어제 냄새를 보고 다음에 고치자고 한 것이, 다음에 와서는 다시 찾아내야 한다. 그 대책으로 작은 로그를 두자. 마크다운 파일 하나면 충분하다. 형식은 3칸: 날짜, 위치(파일과 함수), 이유(한 줄).

```
2026-09-07 src/billing/pricing.py::calculate 40퍼센트 할인율
↪ 마법 숫자, 뜻 불분명
2026-09-07 src/auth/session.py 함수가 너무 많이 함, 둘로 분리
```

이 로그의 역할은 둘이다. 첫째, 냄새가 사라지지 않게 해 주는 알림. 둘째, 두 번째 변경 빈도 지도: git 히스토리가 조용한 곳이라도, 로그에 같은 구역이 반복되면, 그 곳이 반복 마찰을 만드는 곳이라는 것을 안다. 이 로그를 주간 리듬에서 어떻게 쓰느냐는 4장에서 다룬다.

위치도 정해 두자. 리포지토리 안에. 프로젝트 루트의 REFACTOR.md 같은 파일. 리포 안에 있으면 코드가 바뀌는 대로 같이 바뀌고, 메모 앱이나 머릿속에서 찾을 필요가 없다. 항목이 쌓이면 매주 정리한다. 그 정리의 형태는 4장의 주 15분 리뷰다.

2.7 예: 핫스팟 파일을 읽는 일

위 내용을 구체적인 파일에 적용해 보자. git log가 src/billing/pricing.py를 지난 3개월에 14번 건드리고 그 가운데 5번이 할인 중복 수정 같은 커밋 메시지였다고 하자. 파일을 열면, 다음과 같은 것이 보인다.

```
def calculate(base, user, today):
 discount = 0
 if user.level == "gold":
 discount = 0.1
 if today.month == 12:
 discount = 0.15
 if user.coupons:
 for c in user.coupons:
 if c.valid(today):
 discount += 0.3
 tax = 0.07
```

```
    total = base * (1 - discount) * (1 + tax)
    if total > 1000000:
    total = total * 0.95
    return total
```

냄새 목록을 손에 들고 읽자면: 여러 일을 하는 함수(레벨 할인, 계절 할인, 쿠폰, 세금, 대량 주문 할인을 한 곳에서 처리), 마법 숫자(0.1, 0.15, 0.07, 1000000 어느 것도 뜻이 없음), 중복된 형태(할인율이 레벨로, 계절로 각각 배정되고 구조가 같음), 숨은 위험(골드와 쿠폰이 동시에 적용될 때 할인 최대치는 얼마인지, 테스트가 없으니 모름).

커밋 히스토리의 할인 중복 수정 5건과 코드의 형태가 일치하는 것은 우연이 아니다. 할인 로직이 이 한 함수 안에 뿌려져 있어서, 케이스가 하나씩 추가될 때마다 전체를 다시 이해해야 한다. 이 파일은 두 질문 모두에서 우선순위가 높다.

그럼에도 먼저 할 일은? 함수 전체가 아니라, 가장 작은 움직임: 마법 숫자를 이름 있는 상수로 빼낸다. 각 1줄, 위험은 낮고 그러면 각 분기의 뜻이 조금 분명해지고 다음 움직임(할인 계산 추출)이 쉬워진다. 3장에서는 이 파일을 한 칸, 검증, 커밋 절차로 실제로 통과시켜 본다.

2.8 한 페이지 요약

무엇을 고치는가는, 냄새 목록을 변경 빈도와 대조해 정한다. 1인 개발자가 기억해 둘 냄새 다섯 가지: 여러 일을 하는 함수, 중복 로직, 파도 수정, 깊은 조건, 마법 숫자. 그리고 규율: 3줄이면 그 자리에서 고치고, 아니면 리팩토링 로그에 적는다. 다음 장은 어떻게.

3 3장. 어떻게 고치는가: 작은 걸음의 안전 절차

대상은 2장에서, 시점은 1장에서 정했다. 남은 것은, 동작을 깨뜨리지 않고 실제로 코드를 움직이는 절차. 이 장이 책의 핵심. 절차는 짧다. 앵커, 한 칸, 검증, 커밋, 반복.

3.1 전제의 전제: 테스트 앵커

동작은 그대로라는 약속은 검증 수단이 없으면 공염불이다. 테스트 없이 코드를 옮기면, 동작이 바뀌었는지를 아는 유일한 방법은 앱을 돌리고 눈으로 보는 것뿐. 여러 기능을 동시에 돌리는 1인 개발자에게 그 검사 방식은 믿을 수 없다. 그래서 리팩토링의 첫 단계는 늘, 앵커를 설치하는 일이다.

테스트가 있으면 앵커는 쉽다. 전체를 돌려 다 초록이 되면, 한 번 커밋해서 기준선으로 삼는다. 그 다음에 무언가를 깨뜨리면, 테스트가 알려준다.

테스트가 없으면, 특화 테스트를 쓴다. 특화 테스트의 목표는 현재 동작을 기록하는 것이다. 현재 동작에 버그가 있어도, 테스트는 그 버그를 있는 그대로 기록한다. 고치지 말고 기록한다. 순서는 다음과 같다:

1. 입력을 고른다. 대표적으로. 정상 케이스, 경계 값(0, 빈 값, 최대값), 에러 케이스. 가격 함수라면 입력 4건에서 6건이면 충분하다.
2. 현재 코드를 돌리고 출력을 적는다. 출력하고 복사하기만 하면 된다.
3. 적어둔 출력을 기대값으로 어설션을 쓴다.
4. 테스트가 초록이 되는 순간, 그게 앵커. 이 시점부터, 형태를 바꾸고 출력이 달라지면 테스트가 멈춰 세운다.

목표가 I/O나 데이터베이스나 네트워크에 엉킨 함수로, 테스트가 어렵다면, 범위를 줄인다. 모듈 전체를 앵커하지 말고 그 안의 순수 리프 함수, 같은 입력이 같은 출력을 내는 함수부터 앵커한다. 앵커는 사다리다. 한 칸씩 올라가면 된다.

pricing 함수의 경우를 들어 보자. 특화 테스트의 입력으로 다음 네 가지를 골랐다.

```
(User("gold"), date(2026, 6, 10))
(User("gold"), date(2026, 12, 10))
(User("silver"), date(2026, 12, 10))
(User("silver", 쿠폰 1건), date(2026, 6, 1))
```

옛 코드를 돌려 얻은 출력을 적으면: 0.1, 0.15, 0.0, 0.3. 이 값을 기대값으로 어설션을 쓰면, 할인 계산의 현재 동작이 네 건으로 고정된다. 이 네 건이 없어도 버그는 고칠 수 있다. 하지만 네 건이 있으면, 형태를 바꾸는 동안 아무도 이 값을 건드릴 수 없다. 특화 테스트의 목적은 여기 있다.

3.2 원자적 리팩토링 루프

리팩토링의 본체 절차는 다섯 단계다.

단계 1, 앵커. 테스트가 초록이고 작업 트리가 깨끗한 것을 확인한다. 미완성 작업 위에 리팩토링을 시작하면, 어떤 변경이 무엇을 깨뜨렸는지를 구분할 수 없는 상황. 진행 중이 있다면, 먼저 커밋하거나 stash한다.

단계 2, 한 칸. 가장 작은 변경 단위를 골라, 그것만 한다. 변수 이름 바꾸기, 함수 추출, 한 줄 이동. 이 단계에서 잠시 이것도 하고는 금지다. 잠시 이것도가 대청소의 입구다.

단계 3, 검증. 테스트를 돌린다. 테스트가 없으면, 앵커 단계에서 약속한 수동 검사를 한다. 불안하면 함수에 로그를 남기고 전후 출력을 비교한다.

테스트조차 쓰기 어렵다면, 수동 앵커를 쓴다. 움직이기 전에 앱을 한 번 돌려 출력을 저장해 두고 움직인 뒤에 한 번 더 돌려 비교한다. 출력에 시각이나 랜덤 값이 섞여 있어 그대로 비교할 수 없다면, 그 부분만 고정하고 비교한다. 테스트 데이터나 고정된 시점을 사용하는 식으로. 이 앵커의 품질은 테스트보다 낮다. 하지만, 아무것도 없는 것보다 낫다. 수동 앵커만 있는 움직임은, 검증이 어려운 부분부터 나중에 돌린다.

단계 4, 커밋. 작은 커밋. 메시지에 refactor: 접두어를 붙이면, 나중에 역사에서 형태를 바꾼 것을 찾을 때 편하다. 커밋은 체크포인트다. 다음 칸에서 넘어지면, 여기까지 되돌아 올 수 있다.

메시지에 한 줄 더 하라. 어떤 형태를 바꿨고 동작은 안 바꿨다는 형식으로. 예를 들면 refactor(billing): discount_rate 추출, 동작 불변. 동작 불변이라는 반절은 장식이 아니다. 미래의 너에게 대한 계약서다. 나중에 역사에서 이 커밋을 보면, 동작에 손이 닿았는지를 한눈에 알 수 있다.

단계 5, 반복 또는 정지. 형태가 아직 좋다면, 다음 작은 칸으로 단계 2로 돌아간다. 충분하면, 멈추고 리팩토링 로그에 적는다.

처음이라면, 다섯 단계를 한 세션에 다 할 필요는 없다. 앵커와 한 칸만으로도 충분히 의미 있는 세션이다. 반나절에 2에서 3칸, 그것이 첫 리팩토링의 적당량이다.

이 루프는 너무 느리게 보인다. 하지만 속도는 안전에서 온다. 한 칸은 보통 5분에서 15분이고 검증 비용은 거의 0이다. 테스트가 손에 있고 변경이 작으니까. 실수를 해도 한 커밋 되돌아가면 된다. 전체 소요 시간은, 큰 폭의 변경에서 길을 잃고 되돌아 하는 것보다 짧은 경우가 오히려 많다.

왜 한 칸이 최대 단위인가. 이유는 실패를 설명하는 데 있다. 한 칸을 하고 테스트가 빨강이 되면, 원인은 거의 분명히 그 칸이다. 세 칸을 한꺼번에 하고 빨강이 되면, 셋 중 어디인지를 찾지 못한다. 디버깅이 시작된다. 한 칸은, 실패를 설명 가능한 범위 안에 남긴다. 그것이 리팩토링과 디버깅의

경계다.

3.3 작은 움직임 메뉴, 안전한 것부터

가장 안전한 것부터 위험한 것으로 줄 세웠다.

이름 바꾸기. 동작은 바꾸지 않고 이름만 바꾼다. 가장 안전한 움직임이자, 이것은 무엇인지 라는 질문을 강제한다. data2 같은 이름이 pending_orders가 되는 식이다. 현대 편집기의rename 명령은 모든 참조를 한꺼번에 바꾸므로 위험이 최소화된다.

상수 추출. 마법 숫자를 이름 있는 상수로 빼낸다. 영향 범위는 그 숫자가 쓰이던 곳뿐이다.

중첩 평탄화. 만약 아니라면 빠져라 조건을 앞에 두고, 들여쓰기를 낮춘다. 대개 구조만 바뀌고 로직은 같다. 평탄화 전에 확인할 것은, 조건 안에 값이 매겨지던 변수의 기본값. 이것이 이 움직임에서 가장 많이 범하는 실수다.

함수 추출. 연속된 블록을 이름 있는 함수로 빼낸다. 블록이 변수를 여러 개 읽고 여러 개 쓴다면, 인자와 반환값으로 넘긴다. 그 수가 너무 많다면, 빼고 있는 함수가 아직도 일을 너무 한다는 신호다.

클래스 사이 이동. 함수를 가장 많이 의존하는 클래스로 옮긴다. 이 이상의 움직임은 구조를 바꾸므로, 작은 움직임 가운데에서 가장 위험하다. 참조와 import를 전부 바르게 바꿔야 하니까.

이 메뉴에서 흔히 범하는 실수는 둘이다. 첫 번째, 한 번에 여러 개. 이름 바꾸기, 추출, 이동은 각각 한 칸인데, 셋을 한 세션에 몰아넣으면 그것은 하나의 큰 변경이 된다. 그중 하나가 틀리면, 어느 칸에서 깨졌는지 알 수 없다. 두 번째, 검증 생략. 당연히 맞을 것 같으니까. 작은 걸음의 검증 비용은 거의 0이고 실수의 비용은 크다. 비율은 항상 검증 쪽이다.

3.4 pricing 함수를 실제로 통과시켜 보기

2장에서 진단한 src/billing/pricing.py를 루프로 실제로 움직여 보자. 출발 코드는 전 장의 calculate 함수다.

칸 1: 상수 추출.

```
GOLD_RATE = 0.1
DECEMBER_GOLD_RATE = 0.15
COUPON_RATE = 0.3
TAX_RATE = 0.07
LARGE_ORDER_THRESHOLD = 1000000
LARGE_ORDER_RATE = 0.95
```

각각 1줄, 테스트는 그대로, 전부 초록. 커밋. 이 칸에서 0.95가 뜻하는 바를 모른다는 것을 알았다. 그래서 LARGE_ORDER_RATE라고 이름 붙이고, 이건 할인인가 라는 질문을 로그에 남긴다. 모르는 것을 이름 붙이는 것도 진전이다. 모를 것을 보이게 하니까.

칸 2: 할인 계산 추출.

```
def discount_rate(user, today):
 rate = 0
 if user.level == "gold":
 rate = DECEMBER_GOLD_RATE if today.month == 12 else GOLD_RATE
 for c in user.coupons:
 if c.valid(today):
 rate += COUPON_RATE
 return rate
```

discount_rate를 위한 특화 테스트를, 옛 코드를 돌려 얻은 기대값으로 추가한다. 초록, 커밋.

칸 3: calculate 평탄화.

```
def calculate(base, user, today):
 rate = discount_rate(user, today)
 total = base * (1 - rate) * (1 + TAX_RATE)
 if total > LARGE_ORDER_THRESHOLD:
 total *= LARGE_ORDER_RATE
 return total
```

함수는 이제 5줄이고 각 줄의 뜻이 이름 속에 있다. 중간에 한 경계를 확인했다. rate가 1을 넘으면 총액이 음수가 되지 않나. 조합을 전부 세 봤다. 골드 플러스 쿠폰 최대 0.4, 12월 골드 플러스 쿠폰 0.45. 1에 닿지 않는다. 이 질문은 주석 한 줄과 테스트 케이스 한 건으로 닫혔다. 이것이 작은 걸음의 전형적인 이익이다. 큰 폭 리와이트라면 모호하게 남을 질문이, 하나씩 정확한 질문이 된다.

세 칸에 들인 시간은 테스트 포함 약 40분. 동작은 같고(테스트 초록), 형태는 좋아졌다(18줄 한 함수가 상수들과 두 함수가 됐다). 한 주는 필요 없었다.

세 칸의 순서를 돌아 보면, 우연이 아니다. 상수가 먼저라야 분기의 뜻이 분명해진다. 뜻이 분명해져야 빼낼 블록이 보인다. 블록이 빠져야 calculate가 자연스럽게 평평해진다. 각 칸이 다음 칸을 쉽게 만들었다. 이것이 작은 걸음의 진짜 힘이다. 각 칸이 다음 칸을 준비해 준다는 것.

3.5 멈출 때: 중단 기준 네 개

안전한 절차여도, 멈춰야 하는 순간이 있다.

설명하지 못하는 초록이 아닌 테스트가 세 번 연속. 첫 번째 빨강은 무언가를 깨뜨렸다는 신호고 고치면 된다. 두 번째, 세 번째도 빨강이고 원인을 말하지 못한다면, 이미 작은 걸음 상황에 있지 않다. 멈추고, 마지막 초록 커밋으로 돌아가라. git reset -hard가 친구다. 더 작은 칸으로 다시 접근한다.

한 시간이 될 움직임이 리와이트가 될 때. 움직임 도중 이렇게는 안 되고 전체를 다시 짜야 한다면, 그 움직임은 작지 않았다. 중단하고 의도를 둘로 나눠 리팩토링 로그에 적고, 이번 세션은 완성된 부분만으로 끝낸다.

목표의 동작 자체가 불분명해질 때. 리팩토링 도중에, 원 동작이 무엇인지 모른다는 것을 알았다면, 그것은 앵커 공백을 만난 것이다. 리팩토링을 멈추고 단계 1로 돌아가 테스트를 설치한다. 동작을 모르고 코드를 옮기는 것은 리팩토링이 아니다.

컨텍스트가 다른 구역으로 밀릴 때. 움직임 중에 목표 밖 파일을 열고 있다면, 움직임이 작지 않았다. 참조를 확인하는 것만은 예외지만 거기서 무언가를 바꿔야 한다면, 멈춘다.

1인 개발자에게 중단 비용은 실제로 낮다. 조율할 사람이 없고 보고할 사람이 없고 한 칸 되돌림은 자신의 결정이다. 중단 비용이 낮으므로, 중단 기준은 깐깐해야 한다.

네 기준의 모양을 눈여겨 보라. 전부 같은 형태다. 내가 지금 무엇을 하고 있는지 설명을 잃는 순간. 설명이 사라지면 멈추는 것. 작은 걸음이 안전한 이유는, 설명할 수 있어서다.

3.6 한 페이지 요약

절차는 다섯 단계: 앵커, 한 칸, 검증, 커밋, 반복. 앵커는 테스트, 없으면 특화 테스트. 움직임은 메뉴의 작은 움직임 하나. 그리고 빨강을 설명하지 못할 때, 리와이트가 될 때, 동작이 불분명해질 때, 컨텍스트가 밀릴 때, 멈춘다. 다음 장에서는 이 일회성 움직임을 습관으로 연결해, 코드의 형태가 계속 나아지게 만든다.

4 4장. 계속 나아지게 하기: 1인 개발자의 습관

앞의 세 장에서 왜 지금, 무엇을, 어떻게를 답했다. 하지만 의지 있을 때만 돌아가는 기법은 썩는다. 유지보수는 리듬이다. 이 장에서는 앞의 기법을 장기적으로 돌아가는 시스템으로 만든다.

4.1 별도 시간이 아니라, 20퍼센트 몫

리팩토링을 주말에 하자고 하는 계획은 살아남지 않는다. 주말은 오지 않거나, 오면 버그에게 통째로 먹힌다. 배분 방식은 기능 작업 안의 몫이다.

구체적으로: 기능을 추정할 때, 5분의 1 정도를 형태 몫으로 더하자. 3일짜리 기능 추정에는, 지나는 구역에서의 6시간에서 8시간짜리 작은 움직임이 포함된다. 그 시간의 쓰임은 제한된다. 이미 그 기능 때문에 건드리고 있는 파일 안에서만 리팩토링을 한다. 1장의 곁모임 규칙이다. 이 제한의 이익은, 대청소로 흘러가지 않고 테스트와 컨텍스트가 갓 만든 상태라는 점이다.

형태 몫을 안 쓰면, 그것도 신호다. 지나는 구역이 고치기 쉬웠다는 뜻이며 그 자체를 기록할 가치가 있다. 쉬운 구역에는 투자를 넣지 않는다.

한 가지만 더. 형태 몫도 3장의 루프를 따른다. 한 시간 넘을 움직임은 세션 안에 넣지 않고 로그로 내려보낸다. 형태 몫이 살짝 고치다의 탈출구가 되지 않게 하는 잠금이다.

예를 들어 보자. 지난주에 청구서 기능을 추가하던 중, 결제 모듈을 건드려야 했다. 그곳에 tmp 변수를 다시 보았다. 청구서 기능의 형태 몫은 약 6시간으로 잡혀 있었고, 다음과 같이 썼다. tmp를 pending_invoice로 이름 바꾸기(15분), 청구서 발행 함수에 섞여 있던 검증을 함수로 빼내기(1시간), 테스트가 없던 금액 계산 함수에 특화 테스트 쓰기(4시간).

그 뒤에 청구서 기능 본체를 붙이자, 계획보다 2시간 단축됐다. 코드를 돌아다녀야 할 필요가 없어졌다. 형태 몫은 번 것이었다.

4.2 매주 15분: 나와의 리뷰 미팅

1인 개발자에게는 스탠드업이 없고 코드 리뷰가 없고 이번 주에 무엇을 했느냐고 물어줄 사람이 없다. 그래서 스스로 만든다. 주에 한 번, 금요일 오후가 좋고 15분, 액션 셋.

액션 1, 이번 주 커밋을 되돌아본다. 그중 어려운 커밋을 찾는다. 메시지를 쓰는 데 시간이 걸린 것, 파일을 많이 건드린 것, 또 고침이 뒤따른 고침. 그것들이 마찰의 자국이고 자국이 있는 곳이 이번 주 핫스팟이다.

액션 2, 리팩토링 로그를 대조한다. 로그의 항목 중 이번 주에 건 것은 무엇이고 건지 못한 것은 무엇인가. 건지 못한 항목을 지우지 않는다. 재결정한다. 다음 주, 혹은 더 이상 안 함. 더 이상 안 함의 결정은 고치는 것만큼 중요하다. 아무것도 해소되지 않는 로그는 불안이다.

액션 3, 다음 주 항목 하나를 고른다. 단 하나. 기준은 2장의 질문 두 개다. 경험 빈도와, 한 번 고쳤을 때 회수. 항목의 크기는 3장의 루프로 한 시간 안에 끝나는 것이어야 한다.

이 미팅의 산출물은 한 줄이다. 위치, 움직임, 예상 크기. 그것을 리팩토링 로그의 맨 위에 쓴다. 이것이 다음 주의 나와 이번 주의 나를 만나는 방법이다.

15분을 지키지 못한 주가 연속 두 번이면, 리듬이 끊어진 것이다. 최소 회복책은 5분: 이번 주 커밋만 보고 로그의 맨 위 항목 하나를 다음 주로 넘기는 것. 완벽보다 최소를 지킨다.

4.3 그래도 더러울 때

주간 리듬이 있어도, 빚이 갚는 속도보다 빨리 올라오는 코드베이스가 있다. 기능이 급하고 버그가 꼬리를 물고 형태 몫이 가장 먼저 먹히는 경우다. 규율을 버리는 것이 아니라, 속도를 바꾸는 일이다.

1인 개발자에게 있는 특권이 하나 있다. 자신의 속도를 고를 수 있는 것. 기능을 시키는 사람도 너이고 개발하는 사람도 너이니까. 빚이 무거운 주에는, 의식적으로 이번 주 기능을 하나 작게 만든다. nice-to-have를 하나 떨어뜨리거나, 다음 주로 미룬다. 손해처럼 느껴진다. 하지만 그것은 상환이다. 대안, 즉 지난주와 같은 작업량을 받는 것은 차입이고 차입에는 마찰 이자가 붙는다.

판단 기준은 간단하다. 이번 주 15분 리뷰를 하고 나서, 불안이 남는가. 리뷰를 하고도 불안하다면, 너의 속도는 아직 너무 빠르다.

4.4 레거시 스타터 루틴: 5일

코드베이스가 너무 오래되어, 시작할 곳을 모르겠다면, 다음 5일 루틴을 돌린다. 매일의 산출물이 다음 날의 입력이 되는 설계이고, 완주하지 못해도 부분 산출물이 버리지 않는 설계다.

1일차, 지도. 1장의 명령으로 지난 6개월의 변경 빈도 목록을 만들고 상위 5개 파일을 열어 각각 한 번씩 훑어 읽는다. 목표: 핫스팟을 이름 부를 수 있게 되는 것. 모든 줄을 이해할 필요는 없고, 이 파일은 이것을 한다는 말만 나오면 된다.

2일차, 앵커. 가장 뜨거운 파일 하나에 테스트 앵커를 설치한다. I/O에 엉켜 있으면, 순수 리프 함수를 뽑아 특화 테스트를 쓴다. 목표: 이 파일은 움직여도 된다는 자격을 얻는 것.

3일차, 첫 움직임. 루프로 그 파일에서 2에서 3개의 작은 칸을 한다. 가장

보수적인 칸만. 이름 바꾸기, 상수 추출, 중첩 평탄화. 목표: 커밋과 검증의 손 감각.

4일차, 로그와 질문. 이번 주에 발견한 것을 리팩토링 로그에 적고 답하지 못한 것을 질문으로 남긴다. 이 0.95는 무엇인지 모른다는 식으로. 질문 목록은 다음 주 앵커의 지도가 된다.

5일차, 계획. 15분 리뷰를 한 번 돌려, 다음 주 항목 하나를 고른다. 목표: 스타터에서 주간 리듬으로 넘어가는 것.

5일의 목표는 리듬을 설치하는 것이다. 리듬이 설치되면, 향상은 스스로 이어간다. 설치되지 않으면, 오늘 조금 좋게 만든 것이 한 달 뒤에는 원상복귀다.

4.5 리듬을 깨는 함정

잘 돌아가도, 리듬을 부수는 함정이 있다.

함정 1, 도구 함정. 새로운 린터나 포맷터를 도입하는 데 일 주일을 쓰지만 코드 자체는 한 칸도 움직이지 않은 경우. 도구는 이미 리듬이 있을 때 쓰면 가치가 있고 검증 비용을 줄여 줄 때만 도입한다. 도입이 하루를 넘기면 그것은 프로젝트이며, 로그로 내려간다.

함정 2, 완벽 함정. 모듈 하나를 고치면서 이 모듈은 표준에 맞추자고 마음먹는 순간. 리팩토링의 목표는 더 잘 고쳐지는 상태이지, 최고 상태가 아니다. 구역에 표준을 세우는 순간, 3줄 규칙은 무너진다.

함정 3, 기억 함정. 코드를 너무 잘 알아서 테스트가 필요 없다고 느끼는 경우. 코드는 바뀌었고 너도 바뀌었다. 앵커는 기억을 잃어도 믿을 수 있는 것이라서 필요한 것이다.

4.6 성공을 알리는 방법: 측정

허영 지표는, 지운 줄 수나 추출한 함수 수를 잴 때다. 그 숫자가 올라가는 동안 코드가 나빠질 수 있다. 대신, 다음을 잰다.

이해하고 수정하기까지의 시간. 변경 요청을 받았을 때, 코드를 읽기부터 커밋할 자신감까지의 시간. 처음에는 대략적으로 적어도 된다. 1시간이라는 감각이 30분이라는 감각이 됐는가, 몇 달 뒤에 비교한다. 이것이 리팩토링 회수의 핵심이다.

커밋 크기. 평균 커밋 크기가 작아진다면, 형태가 고치기 쉽게 됐다는 뜻이다. 작은 커밋은 되돌리기 쉬울 뿐 아니라, 일이 터졌을 때 원인을 찾는 데도 쉽다.

수정 후 회귀 빈도. A를 고쳤더니 B가 망가졌다는 일의 빈도. 알맞은 곳을 리팩토링하면 이것은 줄고 줄지 않는다면, 앵커 영역과 리팩토링 영역이 다름을 의심한다.

기록은 한 곳에. 리팩토링 로그의 맨 아래가 좋다.

```
2026-08-01 결제 함수 수정: 읽기 40분 + 쓰기 20분
2026-09-01 같은 함수: 읽기 15분 + 쓰기 20분
```

한 줄이면 충분하다. 숫자 자체보다, 적어야 한다는 점에서 가치가 있다. 적다 보면, 길게 느꼈던 시간이 실제로 짧아졌다는 것을 알게 된다.

이 모든 것을 잴 필요는 없다. 월에 한 지표면 충분하다. 측정의 목적은 보정이다. 돈이, 변경 빈도 잣대와 작은 움직임과 주간 리듬이, 알맞은 데로 가고 있는지를 보는 일이다.

4.7 1인 개발자의 이점

마지막으로 비대칭에 대해 말하겠다. 1인 개발자는 유지보수에 불리하다고 한다. 리뷰가 없고 페어프로그래밍이 없고 물어볼 사람이 없으니까. 그런데 리팩토링에서는 이점이 불리보다 크다.

불리는 조율이고 이점은 자유다. 리팩토링 계획을 설명할 사람이 없고 예산이 깎이는 미팅이 없고 해보고 안 되면 돌아가자고 하는 결정이 즉시 난다. 3장의 중단 비용은 1인 상태에서 가장 낮다. 즉, 작은 걸음, 검증, 빠른 중단이라는 절차는 어떤 팀 형태보다 1인 개발자에게 잘 맞는다.

그런데 위험 목록도 있다. 자유의 이점이 곧 위험의 원인인 부분이다. 1인 개발자가 지식의 유일한 소스라면, 그 사람이 오래 자리를 비우는 순간 프로젝트가 멈춘다. 리팩토링 규율이 이것을 완전히 해결하지는 못하지만, 부분적으로는 해결한다. 리팩토링 로그와 질문 목록은 문서화의 최소 형태이고, 작은 커밋은 누군가에게 어디가 잘못된지를 찾게 해 주는 보험이며, 이름 붙이는 규율은 코드를 외부인이 읽을 수 있게 만든다.

이 장의 습관은 코드를 위한 것만으로 끝내지 말자. 너의 부재가 생존 가능해지는 습관이기도 하다. 코드의 형태가 분명하면, 너의 시간도 분명해진다. 리팩토링을 계속하는 사람에게 돌아가는 조용한 이익이다.

이 책이 다루는 규율은 결국, 팀이 대신하던 역할의 대체물이다. 팀의 리뷰가 냄새를 찾아 주었고 동료의 질문이 네이밍을 시험했고 매니저의 일정 압박이 언제를 결정해 주었다. 1인 개발자는 그 세 역할을 스스로 한다. 로그가 냄새를 찾고 질문 목록이 네이밍을 시험하고 변경 빈도가 언제를 결정한다. 도구는 가볍지만 역할은 똑같다.

4.8 마지막 페이지

전 책을 한 흐름으로 요약해 닫겠다. 코드를 바꾸려 할 때, 변경 빈도를 확인한다. 1장. 자주 만지고 마찰이 큰 곳이면, 냄새를 찾는다. 2장. 우선순위가 가장 높은 것을 고르고 3줄이면 그 자리에서 고치고 아니면 로그에 적는다. 때가 되면, 테스트 앵커를 설치하고 작은 걸음 루프로 형태를 나아게 한다. 3장. 그리고 매주 15분 리뷰를 돌려, 리듬을 잇는다. 4장.

다음 수정을 쉽게 만드는 작업이다. 그리고 다음 수정은 생각보다 빨리 온다. 그래서 오늘 내딛는 작은 걸음은, 다음 주의 작업에서 회수된다. 코드의 형태는 계속 쌓이는 자산이며 자산 모두 그렇듯, 그것을 지켜 주는 것은 규율이다.