상태머신의 규율: 주문·결제·문서가 제멋대로 바뀌지 않는 기술

주문, 결제, 문서, 승인처럼 상태가 흐르는 대상을 상태머신으로 명시적으로 모델링해 제멋대로 바뀌지 않게 하고, 테스트와 디버그, 확장이 쉬운 설계로 만드는 오리지널 입문서

ThakiCloud (Hyojung Han)

상태머신의 규율: 주문·결제·문서가 제멋대로 바뀌지 않는 기술

주문, 결제, 문서, 승인처럼 상태가 흐르는 대상을 상태머신으로 명시적으로 모델링해 제멋대로 바뀌지 않게 하고, 테스트와 디버그, 확장이 쉬운 설계로 만드는 오리지널 입문서

ThakiCloud (Hyojung Han)

Contents

1 제1장. 상태가 흐르는 곳, 전부 상태머신이다

주문이 “결제 대기”에서 “결제 완료”로 바뀌는 순간, 당신의 시스템 안에서 어떤 일이 일어나는가. 대부분의 답은 “어딘가에서 if 문이 통과됐다” 정도다. 그런데 그 if 문이 정확히 어디에 있고, 몇 개인지는 아무도 모른다. 한 명만 알았다면 다행인데, 대개는 아무도 모른다. 모른다는 사실 자체가 문제다. 이 장에서는 그 “모름”을 “알음”으로 바꾸는 첫 걸음을 다룬다.

1.1 주문도, 구독도, 문서도 결국 같은 형식이다

주문을 생각해보자. 처음엔 만들어지지 않았다. 사용자가 장바구니를 만들면 “대기”가 된다. 결제를 하면 “결제 완료”. 재고를 확인하고 택배사에 넘기면 “배송 중”. 고객이 받으면 “완료”가 되고 그 사이 어느 시점에 “취소”될 수도 있다. 결제가 실패하면 “결제 실패”로 돌아가 재시도할 수도 있다.

구독은 다르다고 생각하기 쉽다. 그런데 거의 같다. “시범”에서 시작해 “활성”이 되고 결제가 밀리면 “연체”가 되고 한도를 넘으면 “중지”된다. 재결제 성공하면 다시 “활성”으로 돌아온다. 문서는 또 다른 모습인 듯하지만, “작성 중”, “심사 대기”, “승인됨”, “거부됨”의 흐름은 주문과 같은 형식이다. 승인 요청도 마찬가지다. “제출됨”에서 “승인 중”을 거쳐 “완료” 또는 “반려”로 끝난다.

이것들이 공유하는 형식은 하나다. 세 가지로 나누면, 이렇게 된다.

첫째, 이 대상들은 전부 “지금 어디쯤인가”라는 질문에 답할 수 있는 상태가 있다. 대기, 완료, 연체. 이 상태는 동시에 여러 개가 아니다. 주문은 지금 “배송 중”이지, “배송 중”이면서 동시에 “취소”가 아니다. 상태는 한 시점에 하나다.

둘째, 상태를 바꾼다. 스스로 바뀌는 게 아니라, 사건이 일어난다. 결제가 확인되고 사용자가 버튼을 누르고 심사관이 승인을 누른다. 그 사건이 오기를 기다리다가, 조건이 맞으면 다음 상태로 넘어간다.

셋째, 모든 이동이 허용되는 것은 아니다. "결제 대기"에서 "배송 중"으로 바로 갈 수는 없다. 결제가 확인돼야 한다. "완료"된 주문은 더 이상 "취소"되지 않는다. 이 "될 수 없다"를 만드는 규칙이 있다.

이 세 가지, "현재 위치", "사건에 의한 이동", "허용되지 않는 이동"이 바로 상태머신의 전부다. 복잡한 수학 용어가 아니다. 당신이 매일 만들고 있는 것 자체다. 다만 지금까지 이름을 붙이지 않았을 뿐이다. 이름을 붙이는 순간, "어딘가에서 if 문이 통과됐다"가 "이 상태에 이 사건이 오면 이 조건만 충족되면 그 상태로 간다"로 바뀐다. 모르면 알음이다.

1.2 왜 제멋대로 바뀌는가

그러면 왜 주문, 결제, 문서가 "제멋대로" 바뀌는가. 그 이유는 대개 설계의 부재에서 온다. 상태를 명시적으로 정의하지 않은 채, 각 기능마다 흩어뜨려 놓았기 때문이다.

구체적으로 보자. 주문 테이블에 status라는 컬럼이 있다. 값은 "pending", "paid", "shipped", "done", "canceled". 여기까진 괜찮다. 문제는 그 다음이다. 결제 실패를 표현하려고 paid를 만들까, 별도의 payment_status를 만들까. 고민 끝에 payment_status도 추가했다. 재고 확인은 inventory_checked라는 불리언을 넣었다. 환불은 refund_requested도 넣었다.

이제 status 하나만으로 "지금 어디쯤인가"를 알 수 없다. status가 "paid"인데 payment_status가 "failed"이고 inventory_checked가 참인 주문은 정확히 무엇인가. "결제 완료인데, 결제는 실패했고 재고는 확인됐다"는 말인가. 이 조합이 현실에서 일어나는가. 일어나지 않는

조합까지 가능해졌다. 컬럼이 다섯 개가 됐을 때, 그 조합의 수는 다섯 컬럼의 값 곱으로 불어난다. 대부분은 불가능한 조합이고 불가능한 조합에 대한 방어 코드가 여기저기 if 문으로 쌓인다.

더 나쁜 것은, 이 if 문들이 각 서비스에 흩어져 있다는 점이다. 결제 서비스는 결제 서비스를 보고 배송 서비스는 재고를 보고 환불 서비스는 환불 불리언을 본다. 각자 자기 컬럼만 믿는다. 그래서 "이 주문은 지금 도대체 어떤 상태냐"는 질문에 한 목소리의 답을 내기 어렵다. 각 서비스의 if 문이 서로 다른 전제를 깔고 있기 때문이다.

이를 "암시적 상태머신"이라고 부른다. 상태머신은 존재한다. 실제로 흐름이 있기 때문이다. 다만 표로, 규칙으로, 한 데 모아 명시되지 않았을 뿐이다. 암시적인 시스템은 디버깅이 어렵고 테스트가 불완전하고, 한 기능만 바꿔도 다른 곳에 영향을 줄 수 있다. 이 책이 다루는 "규율"이란, 이 암시적 상태머신을 명시적 상태머신으로 바꿔서 한 곳에서, 표로, 읽을 수 있게 만드는 일이다.

1.3 네 단어: 상태, 이벤트, 전이, 가드

명시적 상태머신은 네 단어로 완전히 기술된다. 각각이 무엇인지, 왜 나뉘는지부터 정확히 해두자. 이 구분은 이 책 전체의 뼈대다.

상태는 "지금 어디쯤인가"의 답이다. 대기, 결제 완료, 배송 중, 완료, 취소. 동시에 하나만 참이다. 상태를 나타내는 값은 대개 문자열 상수다. "pending"을 "대기"로, "paid"를 "결제 완료"로. 여기서 중요한 원칙이 있다. 상태는 "현재 위치"다. "결제 완료"는 상태다. 그런데 "결제를 시도했다", "재고를 확인했다"는 상태가 아니다. 이것은 사실, 즉 이벤트의 결과다. 사실을 상태로 섞으면, 지금 본 컬럼 대란이 다시 온다. 위치만 상태에 두자.

이벤트는 "사건"이다. 결제가 확인되고 사용자가 취소를 누르고 심사관이 승인을 누르고 재고 확인이 끝나고. 이벤트는 외부에서 오거나, 내부에서

발생한 사실을 시스템이 수신한 것이다. 이벤트는 상태를 직접 쓰지 않는다. 이벤트를 "수신"한다. 그리고 그 수신에 따라 전이 여부를 판단한다.

전이란 "이 상태 + 이 이벤트 + 이 조건이면, 그 상태로 간다"는 규칙이다. 상태머신의 핵심이 여기 있다. 전이는 (출발 상태, 이벤트)를 입력받아 (도착 상태)를 내뱉는 함수처럼 볼 수 있다. 다만 모든 (상태, 이벤트) 조합에 도착 상태가 있는 것은 아니다. "완료" 상태에 "결제 확인" 이벤트가 와도 갈 곳이 없다. 그 조합은 전이가 없거나, 무시되거나, 에러다. 전이 테이블은 이 "갈 수 있는 곳"만을 적는다.

가드는 전이하기 전의 조건이다. "결제 완료에서 배송 중으로 갈 수 있다"는 전이가 있다 하더라도, 가드인 "재고가 충분한가"가 참이어야 실제로 간다. 재고가 부족하면 전이는 허용돼도 실행되지 않는다. 가드는 전이를 막을 수도, 통과시킬 수도 있다. 가드는 "판단"이다.

이 네 단어가 각기 다른 책임을 지는 순간, 시스템이 읽히기 시작한다. "지금 어디쯤인가"는 상태가 답한다. "무슨 사건이 왔나"는 이벤트가 답한다. "이 사건에 갈 수 있는 곳인가"는 전이가 답한다. "조건이 충족되었나"는 가드가 답한다. 네 질문에 네 답. 이 질문에 답을 못 하던 시스템이, 이제 네 단어에 답을 가진다.

1.4 자기 제품의 상태를 인벤토리처럼 잡는 4단계

이론을 넘어, 지금 만든 제품의 상태를 명시적으로 잡아보자. 주문, 구독, 문서 중 하나를 골라 4단계로 진행한다.

첫째, 대상 하나를 고른다. 전부 한번에 잡으면 커져서 끝이 없다. 주문 하나만 잡자.

둘째, 지금 존재하는 "위치"를 전부 나열한다. 코드, 데이터베이스, 문서, 그리고 팀원들의 말에서 나온 값을 모은다. "pending", "paid", "shipped", "processing", "done", "canceled", "refund". 그리고 payment_status의

값, inventory_checked의 참/거짓, refund_requested의 참/거짓까지. 일단 전부 쏟아붓는다. 정리하지 말고 모은다.

셋째, 그 값들 중에서 "위치"인지 "사실"인지 구분한다. "paid"는 위치다. "재고를 확인했다"는 사실이다. "환불 요청을 받았다"는 사실이다. 위치는 상태 후보로, 사실은 이벤트 후보로 옮긴다. 이 구분이 이 장의 핵심 연습이다. 헷갈리는 값은, "이 값이 참인 동안, 이 대상이 '지금 여기'에 머무는가"를 물어보자. 머무는 것이면 상태, 지나가는 것이면 이벤트.

넷째, 위치 사이를 잇는 사건을 나열한다. "대기"에서 "결제 완료"로 가는 사건은 "결제 확인". "결제 완료"에서 "배송 중"으로 가는 사건은 "재고 확인 후 출고". 각 이동에 "지금 이 이동이 실제로 일어나는가, 아니면 허용은 되지만 조건이 붙는가"를 확인한다. 조건이 붙는 것이 가드다.

이 4단계로 나온 결과물은 표 한 장이다. 행이 상태, 열이 이벤트, 칸이 도착 상태. 그 표를 만들면, "어딘가에서 if 문이 통과됐다"던 시스템이 "이 표의 이 칸이 이 if 문이다"로 바뀐다.

이 표가 왜 중요한지 하나만 더 말하자. 이 표는 약속이다. 팀의 누구든 이 표를 보면, 이 주문이 어디에 머물 수 있는지, 어떤 사건에 움직이는지, 어디로 갈 수 없는지를 한 눈에 안다. 코드를 읽어야 알리던 정보가, 표를 읽으면 안다. 이 차이가 이 책의 "규율"이다. 다음 장에서는 이 표를 어떻게 지을지를 다룬다. 상태와 이벤트의 짓는 법, 전이 테이블의 읽는 법, 빈 칸이 무엇을 의미하는지.

1.5 한눈에: 주문의 상태를 표로 쓰기

주문의 위치를 표 한 장에 옮겨보자. 아직 전이 규칙은 다 넣지 않고 "위치"만 정리한 것이다.

상태	값 예시	지금 머무는 곳
대기	pending	결제 전
결제 완료	paid	출고 전
배송 중	shipped	도착 전
완료	done	끝
취소	canceled	끝

이 표의 핵심은 "지금 머무는 곳"이라는 마지막 열이다. 상태를 정의할 때 스스로에게 "이 값이 참인 동안, 이 주문은 어디에 머물러 있는가"를 묻는 것. 답이 "결제 전이다", "출고 전이다"처럼 위치로 나오면 그건 상태다. 반면 "재고를 확인했다", "환불을 요청받았다"처럼 답이 사건으로 나오면 그건 이벤트의 흔적이다.

여기서 한 가지 주의할 것이 있다. "배송 중"을 "배송 준비", "택배 접수", "배송 이동"으로 쪼개야 하지 않느냐는 유혹이 생긴다. 아직은 참지 말자. 세분화는 다음 장에서 상태의 짓는 법을 배우고 나서, 실제로 필요할 때 한다. 지금은 "하나의 대상에 하나의 위치 열"을 확보하는 것이 목표다.

1.6 자주 하는 실수 두 가지

상태를 잡다 보면 대개 두 가지 실수를 한다.

첫째, 상태를 지나치게 추상화한다. "active", "inactive"처럼 뭉뚝하게 지으면, 나중에 "active인데 도대체 뭐가 active한 거냐"는 질문을 피할 수 없다. "active"는 구독의 "활성", 주문의 "결제 완료"를 함께 가리킨다. 대상별로, 그 대상이 지금 무엇을 기다리고 있는지 드러나게 지어야 한다. 추상화보다 구체가 먼저다.

둘째, 사실을 상태로 삼는다. "재고 확인됨"을 상태로 만들면, 그 주문은 "재고 확인됨"에 머물러 있는 게 된다. 그런데 재고 확인은 순간적인

사건이지, 머무르는 위치가 아니다. 확인을 마친 직후 바로 다음 단계로 넘어간다. 이런 순간적인 사건을 상태에 넣으면, "재고 확인된 채로 아무 일도 일어나지 않고 며칠간 방치된 주문"이라는 실제로 존재하지 않는 상태가 생기고, 그 상태에 대한 방어 코드가 또 if 문으로 쌓인다.

이 두 실수의 공통점은, "위치"와 "사건"의 경계를 흐리는 것이다. 위치는 머문다. 사건은 지나간다. 이 구분을 지키면, 다음 장에서 전이 테이블을 그릴 때 칸이 자연스럽게 채워진다.

1.7 이 장의 요약

상태가 흐르는 대상은 전부 같은 형식이다. 현재 위치, 사건에 의한 이동, 허용되지 않는 이동. 이 형식이 상태머신이다. 주문, 결제, 문서, 승인, 구독은 모두 상태머신이며 지금껏 이름을 붙이지 못했을 뿐이다.

제멋대로 바뀌는 이유는 상태가 흩어져 있으면서도 한 데 모이지 않아서다. 다섯 컬럼의 조합은 대부분 불가능한 상태를 만들어내고, if 문은 각 서비스에 흩어져 한 목소리를 내지 못한다. 이를 암시적 상태머신이라 부른다.

명시적 상태머신은 네 단어로 기술된다. 상태는 현재 위치, 이벤트는 사건, 전이는 이동 규칙, 가드는 조건이다. 네 질문에 네 답. 그리고 자기 제품의 상태를 잡는 4단계, 하나를 고르고 위치를 나열하고 위치와 사실을 구분하고 이동을 잇는 사건을 모은다.

다음 장에서 이 네 단어를 하나씩 다룬다. 상태와 이벤트의 짓는 법, 전이 테이블의 읽는 법, 가드와 실패의 구분까지.

2 제2장. 네 단어로 시스템을 읽는다: 상태, 이벤트, 전이, 가드

제1장에서 상태를 모으는 4단계를 돌렸다. 이제 그 재료로 실제 상태머신을 지을 차례다. 같은 재료라도 이름을 어떻게 짓느냐에 따라, 읽히는 시스템과 읽히지 않는 시스템으로 나뉜다. 이 장은 네 단어를 각각 어떻게 설계하는지를 다룬다.

2.1 상태는 "지금 무엇을 기다리고 있는가"로 짓는다

상태 이름은 한 가지 질문에 답해야 한다. 이 대상은 지금 무엇을 기다리고 있는가.

대기 중인 주문은 무엇을 기다리나. 결제 확인을 기다린다. 그래서 awaitingPayment. 이미 돈을 받은 주문은 무엇을 기다리나. 출고를 기다린다. paid라는 이름은 정확히 "결제 확인을 받았고 출고 대기"라는 위치를 지칭한다. 이름은 이 "기다리는 것"을 드러내야 한다.

이 짓는 방식이 주는 이득이 있다. 새 상태가 들어오려 할 때, "그건 무엇을 기다리는 거지?"라고 물을 수 있다. 답이 두 가지면, 두 상태로 나눌 수 있다. "처리 중"이라는 이름 하나에 "결제 승인 대기"와 "관리자 심사 대기"를 함께 넣으면, 버그가 생겼을 때 어떤 대기에 멈춰 있는지 알 수 없다. awaitingPaymentApproval과 awaitingReview로 나누면, 각 대기의 끝을 맺는 이벤트도 달라진다. 결제 승인 웹훅은 전자만, 심사관의 클릭은 후자만 움직인다.

상태의 개수는 몇이 좋은가. 소규모 제품이라면 4에서 7개가 자연스럽고 10개를 넘으면 의심해야 한다. [추정] 넘었다면 두 가지 후보를 본다. 첫째, 사실인 것이 상태를 끼고 있다. "재고 확인됨" 같은 순간 사건의 흔적이

위치로 들어와 있을 수 있다. 둘째, 서로 다른 두 상태머신이 하나로 엉켰다. 주문의 상태와 환불의 상태가 한 컬럼에 섞여 있으면, 상태를 나누는 게 아니라 머신을 둘로 나누는 편이 낫다. 작은 머신 둘이 큰 머신 하나보다 쉽다.

터미널 상태는 따로 챙긴다. 터미널은 나가는 전이가 없는 상태다. 완료, 취소, 환불 완료. 한번 들어가면 그 대상은 읽기 전용이 된다. 터미널 상태를 명확히 해두면 이득이 두 번 온다. 디버깅이 쉬워진다. "완료인데 왜 취소됐다?"는 질문 자체가 사라진다. 코드가 쉬워진다. 터미널 상태에 도착한 이벤트는 전부 "이 상태에서는 움직이지 않는다"로 묶어서 처리할 수 있어, 가드를 줄일 수 있다.

이름이 바뀌면 코드가 달라지는지, 몇 가지를 나란히 놓아보자.

나쁜 이름	좋은 이름	이유
processing	awaitingPayment	무엇을 기다리는지 드러난다
active	paid	대상과 위치가 명확하다
done	completed	"무엇이" 끝났는지 드러난다
failed	paymentFailed	어떤 실패인지 드러난다

나쁜 이름의 공통점은 "어디에나 붙일 수 있다"는 점이다. processing은 어디에서든 processing이다. 좋은 이름은 이 주문, 이 구독의 위치를 특정한다. 이름이 특정할수록, 그 상태를 만드는 전이와 가드도 특정해진다.

2.2 이벤트는 "이미 일어난 사실"만 받는다

이벤트는 이미 일어난 사실이다. paymentConfirmed, 결제가 확인되었다. refundRequested, 환불이 요청되었다. userRejected, 사용자가 거부했다.

요청과 사실은 다르다. 사용자가 "결제" 버튼을 누르는 것은 요청이다. 이벤트가 아니다. 이벤트는 결제 제공사의 웹훅이 도착하는 순간, paymentConfirmed다. 그 사이에 시간이 들고 재시도가 끼고 상태가 바뀔 수 있다. "버튼 클릭"을 이벤트로 모델링하면, 결제가 실제로 완료하기도 전에 시스템이 "결제 완료"로 움직인다. 제1장의 사고 정확히 그 사고다.

이벤트 이름은 과거 시제로 짓자. 이것을 지키는 것이 실용적인 이유가 있다. 현재형의 pay나 approve는 "시스템이 이것을 해야 한다"는 읽혀서, 코드가 이벤트를 처리하는 순간 부수 효과를 실행하게 만든다. 과거형의 paymentConfirmed는 "이미 일어난 것을 시스템이 안다"로 읽혀서, 처리는 기록과 전이 판단에만 집중하게 한다. 이름 하나가 코드의 성격을 바꾼다.

데이터를 이벤트 이름에 넣지 마라. paymentFailed와 paymentFailedInsufficientBalance는 같은 것이 아니다. 전자가 이벤트, 즉 사실이다. 후자의 "잔고 부족"은 사유다. 사유는 가드의 입력이거나, 히스토리에 남기는 이유 문자열이다. 사유를 이벤트 이름에 섞으면, 사유 종류 수만큼 이벤트가 늘어나고 전이 테이블의 열이 두 배로 불어난다. 테이블이 넓어지면 읽히지 않는다.

2.3 전이 테이블: 빈 칸도 규칙이다

상태머신의 몸통은 표다. 행이 상태, 열이 이벤트, 칸이 다음 상태.

주문의 표를 일부만 뽑아보자.

현재 상태	이벤트	다음 상태
awaitingPayment	paymentConfirmed	paid
awaitingPayment	paymentFailed	canceled
awaitingPayment	userCanceled	canceled
paid	shipRequested	inTransit
paid	refundRequested	refundPending

현재 상태	이벤트	다음 상태
inTransit	delivered	completed
refundPending	refundCompleted	refunded

이 표를 읽는 법부터 다져야 한다. 칸에 값이 있는 것은 "여기에서 그 사건이 오면, 그 값으로 간다". 칸이 빈 것은 "그 조합은 존재하지 않는다". 빈 칸은 미완성이 아니다. "이건 일어나서는 안 된다"는 명세다. paid 상태에서 다시 paymentConfirmed가 오는 것은? 빈 칸. 이미 결제가 확인된 주문에 결제 확인이 또 도착하는 건, 중복이거나 이상이다. 시스템은 이 이벤트를 거부하거나, 이미 처리된 것으로 인정하고 조용히 넘긴다. 둘 중 하나를 정해 두는 것, 그 결정이 규율이다.

표를 그릴 때 가장 흔한 실수는 칸을 다 채우려는 것이다. 모든 상태 곱하기 모든 이벤트에 다음 상태를 억지로 넣으면, 불법이 합법으로 바뀐다. completed에 userCanceled가 와도 canceled로 가나. 아니다. 표는 틈이 많아야 한다. 틈이 많을수록, "이건 없어야 할 것인데 있는데"를 찾는 일이 쉬워진다. 빈 칸이 규율이 된다.

2.4 가드: 전이에 붙는 조건

가드는 이벤트가 도착했을 때 확인하는 불리언이다. 표에 나오지 않는다. 표의 칸에 달린 주석이다.

"paid에서 shipRequested가 오면 inTransit으로 간다"는 칸에 가드가 하나 있다. 재고가 확인되었는가. 창고가 재고 확인을 안 했으면, shipRequested가 와도 가드가 거짓이고, 전이는 일어나지 않는다. 주문은 paid에 머문다. 여기서 중요한 구분이 있다. 이벤트가 사라지는 게 아니다. "받아들이지 않은 것", 거절된 것이다. 시스템은 에러를 내거나, 그 이벤트를 대기시키거나, 무시하고 기록한다. 어느 쪽이든 "가드가 거짓이라 움직이지

않았다"는 사실을 남긴다.

가드 세 가지 규칙을 정하자.

첫째, 가드는 순수 함수다. 컨텍스트를 읽고 참/거짓을 반환한다. 데이터베이스에 쓰지 않는다. 외부 API를 호출하지 않는다. 가드가 API를 호출하고 2초 뒤에 반환한다면, 전이의 타이밍이 불확정적이 된다. 같은 입력에 다른 결과를 낼 수 있다. 테스트도 어려워진다. 가드가 "지금은 괜찮은가"를 답해야 한다면, "괜찮아질 때까지 기다리는" 일은 가드의 몫이 아니다.

둘째, 가드 실패는 이벤트 실패가 아니다. "재고 확인 안 됨"은 합법적인 세계다. 예외가 아니다. 로그에는 "가드 거짓, 전이 보류"로 남는다. 둘을 구분하지 않으면, 진짜 예외가 가드 거짓에 묻혀 찾지 못한다.

셋째, 가드의 이유를 히스토리에 남긴다. 참/거짓만 남기면, 나중에 "왜 움직이지 않았지?"에 답할 수 없다. 가드가 확인한 값, 예를 들어 재고 수와 요청 수를 스냅샷으로 남기면, 논쟁이 시간 순서로 읽히는 기록이 된다.

2.5 상태는 한 컬럼, 사실은 한 줄

설계를 마친 상태머신을 어디에 담느냐도 설계의 일부다. 원칙은 하나다. 현재 위치는 한 컬럼, 일어난 사실은 한 줄.

주문 테이블을 보자. status라는 컬럼이 하나 있다. 값은 이 장에서 지은 이름들, awaitingPayment, paid, inTransit, completed, canceled. 이 컬럼 하나만 보면 "이 주문은 지금 어디쯤"을 안다. 두 번째 컬럼을 보며 추측할 필요가 없다.

그 옆에는 사실 컬럼들이 있다. paidAt, 언제 결제가 확인됐는가. shippedAt, 언제 출고됐는가. refundReason, 환불 사유. 이 사실들은 변하지 않는다. 한번 쓰이면 그대로다. 상태는 변하지만 사실은 누적된다. 이

구분을 데이터베이스 구조로 가져가면, "상태를 다른 컬럼에서 추론하는" 코드가 사라진다. paidAt이 비어 있으면 아직 결제가 안 됐고, shippedAt이 있으면 출고가 됐으니, 이런 if가 필요 없어진다. status를 보면 된다.

반대로, 사실에서 상태를 추론하는 코드는 경보다. "shippedAt이 있으면 배송 중"이라고 쓰는 순간, shippedAt이 있는 배송 완료 주문을 "배송 중"으로 읽는 버그가 생긴다. 사실은 "언제 일어났는가"에 답한다. "지금은 어디인가"에는 답하지 못한다. 두 질문에 두 컬럼.

한 줄을 더 보태자. 상태가 바뀔 때, 그 변화 자체도 한 줄로 남기는 것이 좋다. 어떤 상태에서, 어떤 이벤트가, 언제, 누구의 판단으로, 어느 상태로 옮긴 기록. 이 기록이 히스토리 테이블이다. 제3장에서 코드로 옮길 때, 이 한 줄이 디버깅과 논쟁의 전부임을 보게 된다. 여기서는 "변한 순간을 남긴다"는 원칙만 기억해 두자.

2.6 구체적인 연습: 구독

구독은 돈이 흐르는 상태가 있어서 좋은 연습이다.

상태는 trial, 시범. active, 정상. pastDue, 연체. canceled, 취소. 이벤트는 trialEnded, 시범 종료. paymentSucceeded, 결제 성공. paymentFailed, 결제 실패. gracePeriodEnded, 유예기 종료. userCanceled, 사용자 취소.

현재 상태	이벤트	다음 상태
trial	trialEnded	active
trial	userCanceled	canceled
active	paymentFailed	pastDue
pastDue	paymentSucceeded	active
pastDue	gracePeriodEnded	canceled

pastDue라는 상태가 왜 필요한지도 여기서 말해두자. 결제가 실패했다고

바로 취소하면, 카드 유효기간이 만료되거나 일시적으로 결제가 안 되는 고객이 전부 이탈한다. 그래서 "연체"라는 대기를 넣는다. pastDue에서는 고객이 다시 결제할 시간을 준다. 그 시간, 유예기가 끝나면 비로소 canceled로 간다. 이 상태가 없으면, paymentFailed가 곧 userCanceled와 같아지고, 사업적으로 원하지 않는 결과가 코드의 기본값이 된다. 상태 하나는 사업 정책을 코드에 반영하는 자리다.

이 표에 한 칸을 더 보자. active에 paymentSucceeded가 오면, active로 돌아간다. 자기 루프다. 자기 루프는 합법이다. 다만 경고이기도 하다. 상태를 바꾸지 않는 이벤트를 왜 표에 넣느냐는 질문이다. 구독의 "갱신"은 사업적 사실, 즉 청구가 일어났다는 것이다. 그런데 상태머신에는 상태 변화가 없다. 이럴 땐 두 가지가 있다. 표에 자기 루프로 넣고 가드를 달거나, 전이 체계를 벗어나 사실만 더하는 이벤트로 취급하거나. 둘 다 괜찮다. 다만 하나를 골라 일관하게 쓰는 것이 규율이다.

2.7 설계 질문 다섯 개

표를 그리기 전에, 혹은 다 그린 뒤 한 번씩 되묻자.

1. 상태가 몇 개인가. 8개를 넘으면, 끼어든 사실을 찾거나, 머신을 둘로 나누어 보자.
2. 터미널 상태가 몇 개인가. 없다면, 이 대상은 끝나지 않는다. 대개 누락된 상태가 있다.
3. 이벤트가 전부 사실인가. 현재형 이름이 있다면, 그건 요청이다. 요청과 사실을 나누자.
4. 자기 루프가 있는가. 있다면, 그건 상태머신의 이벤트인가, 사실만 더하는 이벤트인가. 하나를 골라 결정하자.
5. 가드가 전부 순수 함수인가. API를 부르는 가드가 있다면, 그건 가드가 아니라 단계다. 이벤트를 분리하자.

다음 장에서 이 표를 코드로 옮긴다. 전이는 데이터, 실행은 한 함수, 그리고

동시성과 영속화를 어떻게 다룰지.

3 제3장. 규율을 코드로 옮긴다: 전이 테이블과 가드 함수

제2장에서 표를 그렸다. 이제 그 표를 코드로 옮긴다. 옮기는 방식이 전부다. 같은 표라도, if 문으로 풀면 제1장의 암시적 상태머신으로 돌아간다. 전이를 데이터로, 실행을 한 함수로 모아야 표가 살아 있는 규율이 된다.

3.1 핵심: 전이는 데이터, 실행은 한 함수

가장 중요한 결정 하나. 전이 테이블은 데이터다.

나쁜 형태는 전이가 if 문으로 흩어진 것이다.

```
if event == 'paymentConfirmed' and state == 'awaitingPayment':
 state = 'paid'
if event == 'userCanceled' and state in ('awaitingPayment',
↪  'paid'):
 state = 'canceled'
```

요구사항이 바뀔 때마다, 이 if 문들을 전부 찾아야 한다. 그리고 다 찾았다는 보장이 없다. 결제 서비스를 고치다 배송 서비스의 if 문을 빠뜨리면, 두 서비스가 같은 이벤트를 다른 결과로 처리한다. 표가 두 개가 된다.

좋은 형태는 apply 함수 하나와 테이블이다.

```
TRANSITIONS = {
 'awaitingPayment': {
 'paymentConfirmed': 'paid',
 'paymentFailed': 'canceled',
 'userCanceled': 'canceled',
```

```
 },
 'paid': {
 'shipRequested': 'inTransit',
 'refundRequested': 'refundPending',
 'userCanceled': 'canceled',
 },
 'inTransit': {
 'delivered': 'completed',
 },
 'refundPending': {
 'refundCompleted': 'refunded',
 'refundFailed': 'paid',
 },
}
```

이제 전이는 "여기"에 있다. apply 함수는 판단하지 않는다. 실행만 한다.

```
def apply(order, event, ctx):
 current = order.status
 if current in TERMINAL:
 return reject(current, event, '터미널 상태')
 next_state = TRANSITIONS.get(current, {}).get(event)
 if next_state is None:
 return reject(current, event, '이 조합에 규칙 없음')
 guard = GUARDS.get((current, event))
 if guard is not None and not guard(order, ctx):
 return reject(current, event, '가드 거짓',
 ↪  snapshot=ctx.guards())
 version = update_with_lock(order.id, next_state,
 ↪  current_version)
 if version is None:
 return reject(current, event, '동시 충돌, 재시도')
```

```
    insert_history(order.id, current, event, next_state, ctx)
    return accept(current, next_state)
```

apply의 내부는 늘 이 순서다. 터미널인지 묻고 표를 찾고 가드를 확인하고 잠금으로 쓰고 히스토리에 남긴다. 순서가 중요한 이유는, 각 단계가 각기 다른 종류의 거부를 만들기 때문이다. "터미널 상태"와 "규칙 없음"과 "가드 거짓"은 다른 버그다. 셋을 하나의 에러로 묶으면, 제4장에서 디버깅할 때 무엇을 고쳐야 할지 모른다.

이 분리가 규율의 핵심이다. 판단은 데이터, 실행은 함수. "무엇이 변할 수 있는가"를 한 곳에서 읽을 수 있게 된다. 표를 바꿀 때 코드를 고치지 않고 코드를 고칠 때 표를 건드리지 않는다.

3.2 순서가 왜 그 순서인가

apply의 검사 순서가 자의적인 게 아니라는 것을, 각 단계를 빠뜨렸을 때의 사고로 보자.

터미널 확인을 없애면, 완료된 주문이 userCanceled에 응답하기 시작한다. 표에 틈이 남아 있어도, 터미널은 움직이지 말아야 하는 위치다. 표를 믿기 전에 터미널을 먼저 확인하는 이유는, 표는 "일반적인 규칙"이고 터미널은 "항상 예외"이기 때문이다.

표 찾기를 가드보다 뒤에 하면, 존재하지 않는 전이에 대한 가드를 계산한다. completed에 userCanceled가 와도, 그 가드를 실행하고 나서야 "규칙 없음"을 안다. 가드가 비싸다면, 이건 비용 낭비뿐 아니라 버그의 원인이다. 가드가 "이 상태에서는 참"을 반환하는 순간, 없는 전이가 있는 것처럼 보인다. 표가 먼저다.

잠금 없이 쓰면, 앞에서 말한 동시 충돌이 일어난다. 두 전이가 같은 상태를 읽고 둘 다 쓰기에 성공한다. 히스토리가 갈라진다.

마지막에 한 가지만 더. 상태 갱신과 히스토리 기록은 같은 트랜잭션에서 해야 한다. 상태를 쓰고 히스토리를 쓰기 전에 멈추면, "변했다"는 기록이 없는 상태 변화가 남는다. 반대로 히스토리를 쓰고 상태를 쓰지 못하면, "일어났다"는 기록만 있고 세상은 그대로다. 둘 중 어느 쪽도 원하지 않으므로, 둘을 하나로 묶어 원자적으로 만든다. 트랜잭션 하나가 이 문제를 푼다.

이 네 가지를 보면, "가장 싼 거절부터, 원자적 쓰기까지"의 원리가 적용된 결과다.

3.3 가드 함수는 순수하게

가드는 제2장에서 정한 규칙 그대로, 컨텍스트를 읽고 참/거짓을 반환하는 순수 함수다.

```
GUARDS = {
 ('paid', 'shipRequested'): lambda o, ctx:
  ↪  ctx.inventory_confirmed(o.id),
 ('paid', 'userCanceled'): lambda o, ctx: not
  ↪  ctx.refund_in_progress(o.id),
}
```

여기서 ctx는 읽기 전용 스냅샷이다. 가드가 네트워크를 가져오지 않는다. 데이터베이스에 쓰지 않는다.

"재고 확인에 시간이 걸리면 가드에서 기다리면 되잖아"라는 유혹이 온다. 기다리지 마라. 가드가 오래 걸리면, 전이의 타이밍이 불확정적이다. 같은 입력에 다른 결과가 나올 수 있다. 테스트가 불가능해진다. 재고 확인에 시간이 걸린다면, "재고 확인 완료"를 별도의 이벤트로 만들자. shipRequested는 확인이 끝난 뒤에 온다. 가드는 "이미 일어난 사실"을 묻기만 한다.

이것을 일반화하면, 단계는 이벤트로 분리한다. 가드는 “지금은 허용되는가”를 답하고 이벤트는 “일어났다”를 답한다. 경계를 지킬수록 apply는 짧아지고, 가드는 작아진다.

3.4 동시성: 동시에 한 번만

paymentConfirmed 웹훅이 두 번, 동시에 온다고 하자. 둘 다 state가 awaitingPayment인 것을 읽고 둘 다 next가 paid라고 계산한다. 조심 없이 쓰면, 히스토리에 두 줄이 생기고, 버전은 두 번 뛴다. 주문은 한 건인데, “결제 완료”가 두 번 일어난 셈이다.

표준 해법은 낙관적 잠금이다.

```
UPDATE orders
SET status = 'paid', version = version + 1
WHERE id = 1042 AND version = 7
```

버전이 매칭되는 행 하나만 갱신된다. 두 번째 쓰기는 버전이 이미 8이라 매칭이 안 되어 실패한다. 실패한 쪽은 “이미 처리됨”을 반환하거나 재시도하거나 한다.

shipRequested와 userCanceled가 같은 순간에 도착하면, 순서를 정해야 한다. 표준은 이벤트 큐다. 대상 하나당 큐를 두고 이벤트를 순서대로 하나씩 apply한다. 큐가 없다면 둘 중 하나를 정해 두자. 이벤트에 시각이 있으면 시각이 큰 것부터, 없으면 도착 순서대로. “우연히 먼저 처리된 것”이 순서라면, 그 순서는 재현할 수 없다. 재현할 수 없는 순서는 디버깅 불가다.

외부 이벤트의 중복은 별도다. 결제 제공사는 웹훅을 두 번 보낸다. 타임아웃, 재시도, 네트워크. 이건 예상 트래픽이다. 히스토리 테이블에 event_uid 컬럼을 넣고, 같은 uid가 이미 있으면 “중복, 무시”로 끝낸다. idempotent, 같은 입력에 같은 결과,가 목표다. 중복이 버그처럼 보이면, 그건 중복을 다루지 않았기 때문이다.

3.5 컨텍스트: 가드가 보는 것은 읽기 전용 스냅샷

apply의 세 번째 인자, ctx가 무엇인지 정하자. ctx는 가드가 판단에 쓸 수 있는 모든 것의 읽기 전용 묶음이다.

```
ctx = {
 'inventory': {'sku_101': 3, 'sku_102': 0},
 'refund_in_progress': False,
 'guards': lambda: {'inv_confirmed': True},
}
```

ctx를 스냅샷으로 만드는 규칙이 두 개 있다.

첫째, apply 진입 시점에 한 번만 만든다. apply 도중에 ctx를 다시 채우면, 가드와 쓰기가 보는 세상이 달라진다. 가드가 "재고 충분"을 보고 통과했는데, 쓰기 전에 재고가 바뀌었다면, 그 전이는 거짓 전제 위에 서게 된다. 한 번의 스냅샷으로 판단과 쓰기를 같은 세계에 묶는다.

둘째, ctx 안에 쓰기가 들어가지 않는다. ctx는 가드가 읽을 자료다. ctx를 만드는 과정에 데이터베이스 조회가 들어갈 수 있지만 그 결과는 값이어야 하고 함수여서는 안 된다. 값이 아니라 함수를 ctx에 넣으면, 가드가 부를 때마다 조회가 다시 일어나고, 타이밍 불확정이 돌아온다. 제2장의 가드 규칙이 여기까지 이어진다.

3.6 예제로 한 바퀴: 주문의 일생

apply가 실제로 어떻게 돌는지, 주문 하나를 처음부터 끝까지 따라가보자.

1. 주문 1042 생성. status는 awaitingPayment, version 0.
2. paymentConfirmed가 도착. ctx의 재고는 충분. apply는 표를 찾는다. awaitingPayment에서 paymentConfirmed는 paid. 가드가 없다. version 0에서 1로 쓰고 히스토리에 한 줄. from

awaitingPayment, event paymentConfirmed, to paid.

3. paymentConfirmed가 또 도착. 이번엔 중복. event_uid가 이미 히스토리에 있다. "중복, 무시". 전이 없다. 버그도 없다. 예상 트래픽.
4. shipRequested가 도착. ctx의 재고는 충분, 가드 참. paid에서 shipRequested는 inTransit. version 1에서 2로. 히스토리에 한 줄.
5. userCanceled가 도착. inTransit에서 userCanceled는 표에 없다. "규칙 없음"으로 거부. 배송 중인 주문은 여기서 취소되지 않는다. 이 거부가 규율이다. 없었으면, 취소된 주문이 배송되는 사고가 났을 것이다.

이 다섯 걸음을 히스토리 테이블로 읽으면, 2, 4에 줄이 있고 3, 5는 "받았으나 움직이지 않았다"는 기록이다. 움직인 것과 움직이지 않은 것을 모두 남기는 것이 포인트다. 움직이지 않은 것만 남기면, "왜 안 움직였지"에 답할 수 없다.

3.7 영속화: 현재 상태는 한 컬럼, 전체 경로는 히스토리

두 가지를 남긴다.

현재 상태. 대상 행에 status와 version이 있다. 한 값, 한 컬럼. paidAt이나 shipAt으로 status를 추론하지 않는다. 사실은 사실, 상태는 상태. version은 낙관적 잠금의 숫자다.

히스토리. 부가 전용, append-only 테이블이다. 한번 쓰면 지우지 않는다.

열	역할	예
from_state	출발 상태	paid
event	도착 이벤트	shipRequested
to_state	도착 상태	inTransit
guard_snapshot	가드 평가값	inv: 3, req: 1

열	역할	예
event_uid	중복 제거용 식별자	pay_8f2a
actor	누가, 어떤 것	user, webhook
created_at	시각	2026-09-10T03:12:00Z

히스토리의 첫 번째 이득은 디버깅이다. 제4장에서 자세히 보겠지만 "이 주문이 어떻게 여기까지 왔나"를 한 SELECT로 읽는다. 두 번째는 논쟁이다. "나는 분명 취소 요청을 했는데"라는 클레임에, 누가 언제 무엇을 했는지를 시간 순서로 보여줄 수 있다. 세 번째는 미래다. 나중에 새 상태를 추가해도, 과거 데이터는 그대로 읽힌다. 히스토리는 과거를 변경하지 않으므로, 과거의 행위가 그때의 규칙으로 기록되어 있다.

부가 전용이라는 점이 중요하다. 히스토리를 갱신하거나 삭제하면, "그때 실제로 무엇이 일어났는가"의 증거가 사라진다. 그래서 부기만 한다. 실수가 있으면, 새 이벤트로 정정한다. 예를 들어 잘못된 취소를 되돌리려면, cancelReversed라는 이벤트가 히스토리에 한 줄을 더한다. 지우지 않고 정정한다.

3.8 리뷰 질문 다섯 개

상태를 건드리는 PR을 리뷰할 때, 이 다섯 가지를 묻자.

1. 표 밖의 전이가 있는가. status를 직접 대입하는 if가 있다면, 그건 표를 우회하는 경로다.
2. 비순수 가드가 있는가. 가드가 데이터베이스를 쓰거나, API를 부른다면, 그건 가드가 아니라 단계다.
3. 히스토리를 쓰지 않는 경로가 있는가. 전이가 일어나는데 기록이 없다면, 그 전이는 디버깅 불가다.
4. 동시 쓰기가 보호되는가. 버전 확인이 없다면, 중복 전이가 일어난다.

5. 중복이 처리되는가. event_uid가 없다면, 외부 웹훅의 재전송이 버그가 된다.

이 다섯 질문에 “없다”를 말할 수 있으면, 그 코드는 규율을 지킨다. 다음 장에서 이 규율이 테스트, 디버그, 확장에서 어떻게 되돌아오는지를 본다.

4 제4장. 테스트, 디버그, 확장이 쉬워지는 순간

제3장에서 규율을 코드에 박았다. 이 장은 그 규율이 어디에서 값을 하는지를 본다. 테스트, 디버그, 확장. 세 곳.

4.1 테스트: 표의 크기만큼만 쓰면 된다

암시적 상태머신의 테스트는 코드 경로를 테스트한다. 코드가 많아질수록 테스트도 많아지고 "다 짰다"는 확신은 없다. 한 if 문을 빠뜨리면, 그 경로의 테스트도 없다. 빠뜨린 줄도 모른다.

명시적 상태머신의 테스트는 표를 테스트한다. 표는 유한하다. 상태 여섯 개, 이벤트 여덟 개인다면, 조합은 48개다. 테스트도 48개. 하나를 만들면 빠뜨릴 것이 없다.

전체 전이 테스트는 이렇게 생긴다.

```
for state in STATES:
 for event in EVENTS:
 result = apply(make_order(state), event, fake_ctx)
 expected = TRANSITIONS.get(state, {}).get(event)
 if expected is None:
 assert result.rejected
 else:
 assert result.accepted
 assert result.next_state == expected
```

이 테스트의 아름다움은, 표가 바뀌면 테스트가 자동으로 따라온다는 점. 상태를 하나 추가하면, 그 상태 곱하기 모든 이벤트의 줄이 추가된다.

이벤트를 하나 추가하면, 모든 상태 곱하기 그 이벤트의 줄이 추가된다. "케이스를 하나 더 해야 하나"를 묻지 않아도 된다. 표를 바꾸면, 테스트가 알아서 늘어난다.

거절 이유도 테스트한다. "코드가 안 죽었다"는 성공이 아니다. "불법 조합이 이유를 가진 거절로 돌아왔다"가 성공이다.

```
r = apply(make_order('completed'), 'userCanceled', fake_ctx)
assert r.rejected
assert r.reason == '터미널 상태'
```

"터미널 상태", "규칙 없음", "가드 거짓"은 다른 버그다. 이유를 테스트하지 않으면, 셋이 하나의 에러로 합쳐져, 나중에 어떤 것이 고쳐져야 할지 모른다. 제3장에서 순서를 정한 이유가 여기 있다.

가드 테스트는 따로 한다. 가드는 순수 함수이므로, fake_ctx로 참/거짓을 만든다.

```
ctx_true = fake_ctx(inventory_confirmed=True)
ctx_false = fake_ctx(inventory_confirmed=False)
assert GUARDS[('paid','shipRequested')](order, ctx_true)
assert not GUARDS[('paid','shipRequested')](order, ctx_false)
```

가드가 불리언 함수이므로, 이 테스트는 비즈니스 로직이 변해도 변하지 않는다. 가드의 정의, 즉 어떤 값을 보는지가 변할 때만 고친다.

4.2 불변식: 어떤 경로로도 깨지면 안 되는 것

전이 테스트가 "이 상태에서 이 이벤트가 오면 이렇게 간다"를 확인한다면, 불변식 테스트는 "어떤 경로로도 참이어야 하는 것"을 확인한다.

```
def invariants(order):
 assert not (order.status == 'refunded' and order.refund_total
 ↪  is None)
 assert order.version >= 0
 if order.status == 'completed':
 assert order.completedAt is not None
 if order.status == 'awaitingPayment':
 assert order.paidAt is None
```

불변식은 경로와 무관하다. awaitingPayment에서 paid를 거쳐 completed가 됐든, 다른 경로를 거쳤든, "completed이면 completedAt이 있다"는 사실은 변하지 않는다. 전이 테스트가 표의 칸을, 불변식 테스트가 상태와 사실의 관계를 확인한다.

불변식이 유용한 이유는, 표에서 빠진 칸을 잡아주기 때문이다. 표에 "paid에서 paymentFailed가 오면 refunded"라는 칸을 실수로 넣었다고 해보자. paymentFailed는 환불이 아니지만, refunded는 refund_total을 가진다. 불변식이 그 조합을 잡아낸다. 표를 다시 볼 이유가 생긴다.

불변식은 프로덕션에서도 돌리자. 배치로, 하루 한 번, 전체 대상의 불변식을 확인하고 깨진 것만 보고한다. "깨진 것이 없다"가 정상. 깨진 것이 나오면, 그것은 존재하지 않아야 할 상태, 즉 앞에서 말한 사고다. 테스트에서 잡은 사고는 없다. 프로덕션에서 잡은 사고는 있다. 둘을 둘 다 돌리는 것이 규율이다.

4.3 디버그: "어떻게 여기까지 왔나"를 읽는다

프로덕션 사고의 흔한 형태는 "존재하지 않아야 할 상태"다. 완료인데 환불이 진행되는 주문. 취소됐는데 다시 승인된 문서.

암시적 상태머신에서 이걸 조사하는 것은 고고학이다. status를 쓰는 곳을

전부 grep하고 로그에서 "어떻게 여기까지 왔나"를 재구성한다. 오후가 사라진다.

명시적 상태머신에서는 히스토리를 읽는다.

```
SELECT from_state, event, to_state, created_at
FROM transitions
WHERE object_id = 1042
ORDER BY created_at
```

타임라인이 나온다. awaitingPayment에서 paymentConfirmed가 오면 paid, paid에서 shipRequested가 오면 inTransit, inTransit에서 delivered가 오면 completed. 이 타임라인에 틈이 있으면, 그 틈이 버그다. 그리고 틈은 "코드"에 있는 게 아니라, "표"나 "가드"에 있다. 검색 공간이 작다.

더 유용한 습관이 있다. 존재하지 않아야 할 상태가 보이면, 먼저 "표의 경로인가"를 묻는다. 표에 있으면, 표가 틀린 것이다. 설계 버그다. 표에 없으면, 코드가 apply를 우회한 것이다. 구현 버그다. 두 종류, 두 개의 수정 방법. "if를 하나 더 넣겠다"는 수정이 아니다. 표를 고치거나, 우회하는 경로를 없앤다.

실제로 하나를 풀어보자. 고객이 말한 것은 "나는 분명 환불을 요청했는데, 왜 아직 처리가 안 됐어?"다. 히스토리를 읽는다.

```
03:12 paid refundRequested refundPending (actor: user)
03:12 refundPending refundCompleted refunded (actor: webhook)
05:40 refunded userCanceled [거절, 터미널] (actor: user)
```

고객의 환불은 03:12에 이미 refunded로 끝났다. 05:40의 userCanceled는 터미널 상태에서 거절된 것이다. "처리 안 됐어"가 아니라, "처리된 뒤에 취소까지 시도했어"다. 답은 한 줄이다. "03:12에 환불 완료됐습니다." 이 대화, 히스토리가 없으면 로그를 뒤지며 추측해야 한다.

히스토리가 있으면, 시간 순서로 읽히는 증거다.

히스토리가 없으면 이 습관을 못한다. 제3장에서 히스토리를 부기 전용으로, 움직이지 않은 것까지 남기라고 한 이유가 여기 있다. 디버깅은 "변한 것"만으로는 풀리지 않는다. "변하지 않은 것"까지 있어야, 왜 안 변했는지 안다.

4.4 확장: 새 요구사항은 테이블의 한 칸

새 요구사항이 왔다. "배송 후 48시간 내에는 취소할 수 있다."

암시적 상태머신에서: "취소할 수 있는가"를 판단하는 곳을 전부 찾아서 조건을 추가하고, 다 찾았기를 바란다. 회귀 테스트는 추측이다.

명시적 상태머신에서: 표에 칸 하나를 추가한다.

```
'inTransit': {
 'delivered': 'completed',
 'userCanceled': 'cancelRequested',
}
```

가드 하나. "배송 시각으로부터 48시간 이내인가." 그리고 새 상태 하나, cancelRequested, 환불 처리 대기. cancelRequested에서 refundCompleted가 오면 refunded, refundFailed가 오면 inTransit으로 돌아간다.

여기서 이득은, 요구사항의 비용이 보인다는 점이다. "if 3개 수정, 컬럼 1개 추가"가 아니라, "상태 1개, 칸 1개, 가드 1개, 테스트 N개". 후자는 추정 가능하다. 테스트 수는 표의 크기로 나온다. 추정 가능하다는 것이 소규모 팀에게는 큰 이득이다. 큰 요구사항인지, 표가 잘못된 건지, 코드를 쓰기 전에 안다.

반대 사례도 하나. "프로모션 코드"를 추가한다고 하자. 이건 상태를 안 바꾼다. 주문은 그대로 awaitingPayment에 있고 할인 여부는 사실이다.

promoCodeAt이라는 컬럼에 값을 넣으면 끝이다. 표에 칸이 하나도 늘지 않는다. "새 요구사항이 왔으니 표를 고치자"가 아니라, "새 요구사항이 왔으니, 이건 상태인가, 사실인가"를 먼저 묻는 것이다. 상태이면 표가 커지고 사실이면 컬럼이 커진다. 둘은 다른 비용이다. 흐름은 바뀌지 않고, 데이터만 늘어나는 요구사항까지 "상태머신을 고친다"고 착각하면, 표가 불필요하게 불어난다.

4.5 상태머신이 답이 아닌 때

이 책을 읽으며 "모든 것을 상태머신으로 해야 하는가"를 물을 수 있다. 답은, 아니다. 상태머신이 잘 맞는 것은, 동시에 하나인 위치가 있고 그 위치가 소수로 셀 수 있는 대상이다. 주문, 구독, 문서, 승인. 이런 대상.

그런데 몇 가지는 다른 모양이다. 편집 도구의 되돌기와 다시 실행은, 위치가 아니라 행위의 순서다. 상태가 아니라 스택이다. 여러 하위 상태가 동시에 진전하는 대상, 예를 들어 "결제"와 "재고"가 각각 진행되면서 둘 다 끝났을 때만 완료되는 주문은, 하나의 단일 상태열보다, 둘의 조합이다. 이런 것을 억지로 하나의 상태열로 붙이면, 상태가 곱으로 불어난다. 앞 장에서 말한 "상태 폭발"이다.

이런 경우의 판단 기준은, "위치 하나가 이것을 설명하는가"다. 설명하면, 단일 상태머신. 설명하지 못하고 둘 이상이 동시에 움직여야 하면, 둘의 상태머신을 따로 갖고 조합을 명시한다. "주문은 inTransit이고 환불은 진행 중"이라고 두 머신을 읽을 수 있다면, 하나의 머신으로 합칠 필요는 없다.

한 가지 더. 상태머신은 도메인을 대신하지 않는다. 이벤트를 올바르게 받아야, 올바른 전이로 이어진다. 결제 제공사가 잘못된 webhook을 보낸다면, 상태머신은 그 이벤트를 "있는 것"으로 처리할 뿐, "옳은 것"으로 만들어 주지 않는다. 상태머신이 고치는 것은, 한 번 들어온 사건에 시스템이 제멋대로 반응하는 일이다. 사건 자체의 품질은, 통합의 문제다. 이 경계를 알면, 상태머신에 기대지 않게 되고 그래서 상태머신도 과하게

믿지 않게 된다.

4.6 팀: 표는 회의 자료

간과하기 쉬운 이득이 하나 더 있다. 상태머신에서는 기획, 개발, 운영이 같은 대상을 본다. 기획은 "여기서 취소 가능해?"를 묻지 않고 칸을 가리킨다. 운영은 "pastDue가 뭐야?"를 묻지 않고, 가드를 읽는다.

실용적인 습관을 하나만 남긴다. 새 기능을 코딩하기 전에, 표를 화이트보드에 그린다. 그림이 10분이면, 기능은 작다. 1시간이면, 기능이 크거나, 상태머신이 틀린 것이다. 둘 중 어느 쪽이든, 코드를 쓰기 전에 안다.

새 상태가 들어오려 할 때, "이건 어떤 상태의 일부지?"를 묻는다. 답이 "새로운 거야"라면, 이건 상태인가, 사실인가. "할인 적용됨"은 사실이다. discountAt이라는 컬럼이다. "할인 승인 대기"는 상태다. 이 구분이, 표가 커지는 건지, 행이 커지는 건지를 정한다.

4.7 시간이 없다면, 최소 셋만

"이 책을 다 적용하기 전에 먼저 출시를 해야 해"라는 사람도 있다. 맞다. 그렇다면 최소 셋만 하자.

첫째, 표를 하나 그려라. 지금 만든 대상의 상태와 이벤트를, A4 한 장에. 그림을 그리는 것만으로 "빈 칸"이 보인다. 빈 칸은 "여기가 문제다"를 말한다.

둘째, status를 직접 쓰는 if를 한 군데로 모어라. 전부 apply로 안 바꿔도, "status =…"이 흩어진 것을 한 함수에 모은다. 전이가 "여기"에 있음을 만든다. 표를 데이터로 바꾸는 것은 그다음 일이다.

셋째, 히스토리 테이블을 만들어라. 부기 전용. from, event, to, 시각.

이 셋만 있어도, “어떻게 여기까지 왔나”를 읽을 수 있다. 디버깅의 8할이 이것이다.

이 셋은 하루 안에 끝난다. 그리고 다음 요구사항이 올 때, “if를 하나 더 넣자”가 아니라, “표를 하나 더 고치자”로 바뀌어 있다. 그 한 문장의 차이가, 이 책의 전부다.

4.8 닫는 질문 열 개

출시 전에, 상태가 흐르는 대상에게 이 열 가지를 물어보자.

1. “상태가 몇 개인가”에 한 번에 답할 수 있는가.
2. 여러 컬럼을 봐야 하는 상태가 있는가. 있다면, 그건 상태가 아니다.
3. 이벤트가 전부 사실, 과거 시자인가.
4. 전이 테이블이 한 곳에 있는가.
5. 표를 우회해 status를 직접 쓰는 if가 있는가.
6. 가드가 전부 순수 함수인가.
7. 동시 쓰기에 버전이 있는가.
8. 중복 이벤트에 event_uid가 있는가.
9. “이 대상이 어떻게 여기까지 왔나”를 히스토리로 답할 수 있는가.
10. 새 상태를 추가하면, 테스트 수를 추정할 수 있는가.

열 개에 전부 “예”라면, 그 대상은 규율을 지킨다. 아니라면, 답하지 못하는 것부터 시작하자. 가장 큰 것이 아니라, 가장 아픈 것부터. 아픈 것은 대개, 고객이 보는 것이고 돈이 흐르는 것이다.

이 책의 전부다. 주문, 결제, 문서, 승인. 당신의 제품에도, 상태가 흐르는 대상이 하나쯤은 있다. 그 대상을 “어딘가에서 if 문이 통과됐다”에서 “이 표의 이 칸”으로 옮기는 일. 그 차이가, 제멋대로 바뀌지 않는 시스템이다.