빌려 쓰는 코드도 우리 코드다: 서드파티 의존성의 기술

1인 개발자를 위한 의존성 고르기, 잠그기, 업데이트, 교체, 제거의 규율

ThakiCloud (Hyojung Han)

빌려 쓰는 코드도 우리 코드다: 서드파티 의존성의 기술

1인 개발자를 위한 의존성 고르기, 잠그기, 업데이트, 교체, 제거의 규율

ThakiCloud (Hyojung Han)

Contents

1 1장. 빌림의 대가: 의존성이 설계가 되는 순간

반복되는 장면이 있다. 제품을 만들고 있는데 작은 문제가 생긴다. 날짜 처리, 파일 업로드, 인증, 메일 전송. 이런 것들 중 하나. 레지스트리를 검색하고 별점도 괜찮아 보이는 라이브러리를 하나 골라 5분 만에 설치한다. 문서의 첫 예제가 그대로 돌아간다. 그날 밤은 잘 잔다.

열여덟 달 뒤. 같은 라이브러리가 의존성 트리에 여전히 있다. 이제 그 작은 문제 하나만 맡는 게 아니다. 설정 파일, 배포 스크립트, 심지어 테스트까지 그걸 참조하게 됐다. 바꾸려다 보면 체인지로그를 읽고 어디가 깨질지 추측해야 한다. 몇 시간 아끼려고 빌려온 라이브러리가, 이제는 건드릴 때마다 몇 시간을 갚아야 하는 존재가 된 것이다.

이 두 순간의 차이는 시간이 아니다. 첫 순간에는 결정이 공짜로 느껴졌고, 두 번째 순간에는 대가가 조용히 붙어 있었다. 이 책은 그 대가에 관한 것이다. 그리고 그 대가에 휘둘리지 않는 법에 관한 것이다.

1.1 의존성이란 계약이다

의존성(dependency)의 가장 짧은 정의는 이렇다. 쓰고 있지만 통제하지 못하는 코드. 라이브러리 저자가 다음 버전, 출시 시기, 라이선스, 질문에 답할지 여부를 결정한다. 당신은 그쪽이 내민 것을 그대로 써야 한다.

그러니까 설치라는 행위는 짧은 계약의 서명이다. 조항은 몇 개 없는데, 하나하나가 아프다.

- 상대방은 언제든 우리 빌드를 깨뜨릴 수 있다. 그게 메이저 버전이다.
- 상대방은 유지보수를 그만둘 수 있다. 프로젝트는 방치될 수 있다.

- 상대방은 조건을 바꿀 수 있다. 라이선스가 더 엄해질 수 있다.
- 상대방이 사라지면, 정리 비용은 우리가 낸다.

이 조항이 실제로 발동하는 모습을 상상해 보자. 어느 날 메이저 버전이 올라온다. 체인지로그를 열어 보면 "구버전 API 제거"라고 한 줄 있다. 빌드가 깨지고 고치다 보니 함수 이름 두 개가 바뀐 것을 뒤늦게 발견한다. 예전에는 한 시간이면 끝났던 작업이 반나절이 된다. 그 반나절은 라이브러리 측에서 내는 것이 아니라, 우리가 내는 대가다.

계약은 조항을 안 읽었을 때 문제가 된다. 대부분의 설치는 체인지로그도, 이슈 목록도 안 보고 한다. 설치 순간에는 아무 일도 일어나지 않는다. 하지만 의무는 그 순간 이미 발생했다. 이후 이 라이브러리는 매 릴리스, 매 취약성 고지, 매 리팩터링, 매 배포 때마다 확인해야 하는 존재가 된다.

1.2 빌림의 네 가지 대가

의존성을 하나 추가할 때 한꺼번에 들어오는 대가는 네 가지다.

첫째는 유지보수 대가. 앞으로는 업데이트를 계속해야 한다. 체인지로그의 보안 패치도, 메이저 버전의 깨진 변경도. 버전을 제자리로 두는 비용이 제로인 것은 없다. 다만 그 비용이 미래로 넘어가며 커지는 것일 뿐이다. 둘째는 학습 대가. 안전하게 쓰려면 어느 정도 이해해야 한다. 문서는 어떻게 쓰는지만 알려주고 깨졌을 때 뭘 할 것인가까지는 알려주지 않는다. 그 부분은 이슈 토론이나 소스 코드를 읽으며 직접 습득한다. 셋째는 보안 대가. 공격 표면이 넓어진다. 모든 라이브러리는 문이고 그 열쇠는 우리가 갖지 않는다. 최근 몇 년간, 방치된 패키지를 대체하는 순간에 공격이 들어온 사례가 실제로 있었다. 유명 프로젝트도 그 흐름에 휘말렸다. 넷째는 라이선스 대가. 안 읽은 조항에 묶일 수 있다. 개인 사용은 괜찮아도, 상업적 재배포는 이야기가 다를 수 있다.

이 네 가지 대가는 한꺼번에 보이지 않는다. 학습 대가는 가장 먼저 오고

유지보수 대가는 몇 달 뒤 따라오고, 보안 대가는 수년 뒤에 나타나는 경우도 있다. 그래서 결정 순간에는 대가가 공짜로 느껴진다. 그게 함정이다. 대가를 보이게 하려면, 결정 순간에 대가를 기록해 두어야 한다. 그 기록의 틀은 다음 섹션들에서 하나씩 만든다.

1.3 왜 1인 팀일수록 대가는 큰가

동일한 의존성은 큰 팀과 1인 팀에게 같은 무게로 다가오지 않는다. 팀이 있으면 업데이트는 이슈 하나를 건네는 일이고, 깨진 빌드는 누가 먼저든 고치는 일이다. 그런데 1인 팀에서 같은 일은 전부 "내일"이다.

먼저, 대가는 내 시간에서 나간다. 팀은 대가를 인원으로 나눈다. 1인 개발자는 나눌 사람이 없다. 의존성 업데이트에 쓰는 시간만큼, 제품 개발에 쓰는 시간이 줄어든다. 둘째, 지식도 나한테만 있다. 이 라이브러리를 왜 골랐는지, 어디까지 건드렸는지, 설정값이 왜 이렇게 됐는지를 아는 사람은 나 하나다. 내가 일주일 자리를 비우면, 그 의존성과 관련된 판단도 일주일 멈춘다. 셋째, 선택의 자유가 줄어든다. 팀이 있으면 "이거 불안하니 다른 걸로 갈아보자"는 말이 나온다. 1인 팀에서는 갈아타기가 프로젝트가 아니라, 다음 기능을 포기하는 결정으로 보이기 쉽다.

그래서 1인 팀에게는 대가의 합보다 대가의 시기가 중요하다. 한 번에 오는 대가는 견딜 수 있다. 매 분기 조금씩 오는 대가는 쌓인다. 이 장의 나머지 섹션은, 쌓이는 대가를 한 장의 표로 가져오는 작업이다.

1.4 직접 만들지, 빌릴지: 결정 라인

"이거 직접 쓰면 될까, 아니면 라이브러리를 쓰면 될까?" 정답은 두 축으로 갈린다. 제품에 얼마나 핵심적인가와, 얼마나 자주 변하는가.

전략	조건
빌린다	공통 문제, 풀이가 안정됐고
빌린다	핵심이나 차별점은 다른 데서
직접 쓴다	제품의 핵심 차별점 자체다
직접 쓴다	몇백 줄이면 쓸 수 있다

예로 하나 잡아 보자. 요청 몸체의 모양을 검증하는 작업이 있다. 이 검증이 우리 서비스의 한 부위라면, 공통 문제고 안정된 풀이가 이미 있다. 빌린다. 그런데 만약 우리 제품이 "검증 규칙 서비스" 자체라면, 검증은 부위가 아니라 전부다. 이럴 때는 직접 쓴다. 같은 기술적 작업이, 어떤 제품에서냐에 따라 라인의 반대편에 떨어진다. 그래서 라인은 기술이 아니라 제품 기준으로 그리어야 한다.

첫 번째 규칙은 핵심성에 관한 것이다. 그 기능이 우리 제품의 차별점이라면, 빌리는 것이 아니라 장점을 남의 라이브러리에 맡기는 셈이다. 모두가 쓰는 라이브러리는 모두의 문제를 푼다. 우리 제품만의 부분은 우리 코드가 되어야 한다. 두 번째 규칙은 변화의 크기에 관한 것이다. 매 분기 수정이 필요한 코드를 남의 코드 위에 얹으면, 매 수정마다 남의 코드를 다시 읽는 비용이 붙는다. 그 비용은 쌓일 뿐 줄지 않는다. 그럴 경우는 우리가 건드는 부분만 직접 쓰는 편이 싼 경우가 많다.

반대로, 날짜나 로깅, 검증 같은 공통 문제는 굳이 직접 만들 필요가 없다. 성숙한 라이브러리는 우리가 밟았을 버그를 이미 다 밟아줬다. 핵심은 이 라인을 미리 정하는 것. "핵심이고 자주 변하면 직접, 공통이면 빌린다." 라인이 없으면 라이브러리 하나 뜰 때마다 제로에서 논쟁을 하고 그 논쟁은 항상 빌리는 쪽으로 기울어진다. 빌리는 쪽이 빠르니까.

1.5 의존성 예산

결정 라인이 있으면, 이제는 한도가 필요하다. 한도를 영어로 하면 예산이다. 프로젝트에 숫자 세 개를 정해 README에 적어두자.

첫째는 직접 의존성 개수의 한도. 출발점으로 20개 이하를 두고 싶다 [추정]. 숫자 자체는 성역이 아니다. 그 역할은, 새로운 설치를 할 때마다 한 번 멈추게 만드는 것. 둘째는 무게의 한도. 웹앱이라면 라이브러리가 붙여 주는 번들 크기, 서버라면 설치 크기와 프로세스 수. 무게가 두 배로 늘었다면 그게 질문이 되어야 한다. 셋째는 변화 빈도의 한도. 이 라이브러리를 1년에 몇 번 건드려야 하는가. 메이저가 매년 바뀌는 라이브러리는 1인 팀에게 매년 이주 작업을 예약해 주는 것과 같다. 그 작업을 예산에 넣어야 한다.

예산은 금지가 아니라 관문이다. 새 의존성을 추가하고 싶을 때 질문이 하나 생긴다. 이건 한도 안에 들어가는가, 아니면 기존 것을 밀어내야 하는가. 밀어내야 한다면, 새 라이브러리는 기존 것보다 더 나은 이유가 있어야 한다. 그 질문이 규율의 핵심이다.

예산이 꽉 찼을 때의 절차도 미리 정해 두자. 새 의존성이 들어오고 싶으면, 재고 시트에서 먼저 빼낼 후보를 찾는다. 없다면, "새것이 기존 20개보다 중요한가"를 물어야 한다. 더 중요하다면, 기존 하나를 4장 절차로 빼고 나서야 새 것을 넣는다. 아니면, 넣지 않는다. 예산은 깨지는 천장이 아니라, 크기가 항상 같은 문이다.

1.6 빌리지 말라는 말이 아니다

이 책이 계속 대가를 기록하고 예산을 갖고 라인을 그리라고 말한다. "빌리지 말라"는 뜻으로 읽힐 수 있다. 아니다. 1인 팀에게는 빌리지 않는 쪽이 더 위험한 길이다. 공통 부분을 직접 만들며 허비할 시간이, 우리 제품의 차별점에 쓰였어야 할 시간 그 자체다.

규율은 빌리지 않는 것이 아니라, 빌림을 아는 것이다. 대가가 뭔지, 계약이 어떻게 깨지는지, 빠져나가는 데 얼마가 드는지 아는 것. 모르는 채 빌리면 부채다. 아는 채 빌리면 투자다.

1.7 첫 번째 단계는 재고

전략에 앞서, 지금 무엇을 빌리고 있는지부터 알아야 한다. 재고 조사는 1시간이면 끝난다.

먼저 직접 의존성을 나열한다. 대부분의 생태계에서 매니페스트(package.json, pyproject.toml, Cargo.toml)가 직접 의존성을 기록한다. 다음으로 의존성 트리 명령의 출력에서 간접으로 끌려온 것을 확인한다. 숫자에 놀랄 수 있다. 간접 의존은 직접 의존의 몇 배인 경우가 흔하다.

각 직접 의존성에 대해 네 가지를 적는다. 어떤 문제를 푸는지, 언제 추가했는지, 통합에 얼마가 들었는지, 제거하면 얼마가 들는지. 마지막 항목이 가장 중요하다. 제거 비용을 모르면, 진짜 가격을 모른다는 뜻이다.

항목	적을 내용
문제	우리 서비스에서 뭘 하는가
통합 비용	들여올 때 든 노력
제거 비용	빼고 나서 들 예상 노력
마지막 갱신	버전을 마지막으로 건드린 때

행을 채운 예시를 하나 보자. 날짜 처리 라이브러리 한 줄이면, “문제” 칸에 시간대 규칙 처리라고 적고, “제거 비용” 칸에는 반나절이라고 적는다. 인증 라이브러리 한 줄이면, “문제” 칸에 세션 토큰 검증, “제거 비용” 칸에는 나흘이다 [추정].

재고를 돌리다 보면, 한 가지가 항상 보인다. 기록이 없는 의존성. 들여온 이유도, 들여온 것도, 전부 비어 있는 것. 그 빈칸 자체가 신호다. 기록

없이 결정했다는 뜻이다. 기억나는 만큼만 적고 기억이 나지 않으면 "알 수 없음"이라고 적는다. 재고는 알 수 없음을 숨기는 곳이 아니라, 드러내는 곳이다. 빈칸을 한 번 써 보면, 다음 선택에서 그 빈칸을 남기지 않으려 한다.

중요한 것은 제거 비용 칸에 실제 단위가 들어가는 것이다. "적다", "많다"가 아니라, "반나절", "나흘" 같은 우리 손으로 재는 단위. 칸에 단위가 없으면, 그 의존성은 아직 재고에 오르지 않은 셈이다.

이 시트는 이 책 후반부의 원자재다. 4장에서는 이 시트로 분기 리뷰를 돌린다. 재고가 끝날 때까지, 전략 이야기는 전부 탁상공론이다.

1.8 오늘 할 일

책 전체를 읽고 나서 움직이려고 하지 말자. 오늘 하나만 한다. 매니페스트를 열어 직접 의존성 개수를 세고, README에 그 개수를 적는 것. 정한 한도를 넘고 있다면, 그게 대화의 시작점이다. 안 넘고 있다면, 그게 앞으로의 좋은 습관이다.

재고에서 한 가지를 더 골라 보자. 제거 비용이 가장 큰 의존성은 무엇인가. 그 답을 알면, 예산의 질문이 반으로 줄어든다. "새것을 넣을 자리가 있는가"가 아니라, "이 무거운 것을 뺐을 때 새것이 들어올 자리가 생기는가"가 되니까.

빌림의 대가는 설치 순간에 치르는 것이 아니다. 변화 순간에 치른다. 그리고 변화를 만드는 사람은, 언제나 우리다.

2 2장. 고르는 눈: 라이브러리 심사 프레임워크

레지스트리는 같은 일을 하는 라이브러리로 가득하다. 다운로드가 제일 많은 후보, API가 가장 깔끔해 보이는 후보, 내가 따라다니는 사람이 만든 후보, 동료 하나가 권해주는 후보. 네댓 개쯤이면 선택지가 끝난다. 어느 것을 골라야 할까.

가장 흔히 쓰는 기준. 다운로드 수가 가장 믿을 수 없는 기준. 다운로드 수는 얼마나 많은 사람이 써 봤는지를 재고, 얼마나 많은 사람이 지금도 안전하게 쓰고 있는지 재지 못한다. 한때 인기 폭발이었다가 방치된 라이브러리도 다운로드 수는 그대로 높다. 그런데 오늘부터의 설치는, 다음 업데이트가 언제 올지 모르는 것에 대한 베팅이다. 그래서 우리는 다른 것을 보아야 한다. 이 장에서는 설치 전에 라이브러리를 볼 기준을 만들어 둔다.

2.1 유지보수 신호를 읽는다

라이브러리는 살아 있는 존재다. 사람처럼 건강 지표가 있고 그 지표를 보려면 소스를 읽을 필요는 없다. 이 여섯 가지 신호를 보면 된다.

첫째, 릴리스 주기. 마지막 릴리스는 언제이고 릴리스가 어떤 리듬으로 오는가. 성숙한 라이브러리는 드물게 올 수도 있다. 그런데 그 사이 이슈가 쌓여 가고 아무도 답이 없다면, 멈춤이다. 둘째, 이슈 응답. 유지자가 답을 하는가. 답을 한다면, 닫을 때 이유를 적어 주는가. 이슈 목록이 몇 달째 조용한 저장소는, 아카이브로 표시되지 않았더라도 사실상 방치된 프로젝트다. 셋째, 체인지로그의 질. 무엇을 바꿨고 무엇을 깨뜨렸고 어떻게 이주하는지 적혀 있는가. 체인지로그가 커밋 링크 목록이라면, 이주 비용을 우리가 다 낸다. 넷째, 릴리스 기강. 버전 태그가 제대로 붙어 있는가. 롤백, 즉 철회된 릴리스가 있었는가. 자주 롤백하는 라이브러리는 릴리스 프로세스 자체가 흔들리고 있다는 뜻이다. 다섯째, 유지자 구조.

한 사람인가, 팀인가. 한 사람의 라이브러리가 나쁜 것은 아니다. 다만 리스크가 한곳에 몰려 있고 버스 팩터가 하나라는 뜻이다. 여섯째, 전달 의존 개수. 이 라이브러리가 다른 것을 몇 개 끌고 오는가. 작은 유틸리티가 생태계의 절반을 끌고 온다면, 제거 대가도 작지 않다.

여섯 가지를 매번 전부 확인하라는 것이 아니다. 익숙한 범주라면 가벼이 훑어도 된다. 다만 핵심 경로에 닿는 라이브러리는, 하나만 건너뛰어도 안 된다.

2.2 신호의 함정

신호를 읽다 보면 세 가지 함정이 나온다.

첫 번째 함정은 숫자 착시. 별 수, 다운로드 수, 포크 수. 이 숫자들은 "얼마나 많은 사람이 봤는가"를 재지, "얼마나 건전하게 유지되는가"를 재지 않는다. 특히 별 수는 한때의 감탄이다. 프로젝트가 방치된 뒤에도 별은 그대로 남아 있다. 두 번째 함정은 최신성 편향. 릴리스가 최근이면 건강해 보이는 데서 끝나지 않는다. 릴리스가 잦으면 관리가 잘되는 것처럼 보이지만 실제로는 작은 변경까지 버전 번호를 올리는 습관일 수 있다. 버전이 잘 바뀌는 라이브러리가 우리 코드를 자주 흔드는 라이브러리인 경우도 있다. 세 번째 함정은 지인 편향. 따라다니는 사람이 만들었다는 사실은, 그 사람의 평판을 대용으로 쓴 것이다. 유지자가 좋은 사람이라는 것과 프로젝트가 유지된다라는 것은 별개의 사실이다. 사람이 바뀔 수 있고, 관심이 바뀔 수 있다.

함정은 알고 있어 피하는 것이다. 숫자가 좋으면 신호 여섯 가지로 한 번 더 확인하고 최신이면 릴리스의 내용까지 확인하고, 지인이면 유지자 구조 항목을 특히 찬찬히 본다.

2.3 선택 전 다섯 질문

신호는 바깥에서 보는 검사다. 다음은 안쪽에서 보는 검사다. 설치 전에 다섯 질문의 답을 쓴다.

질문 하나: 이건 정확히 무엇을 대체하는가. 범위를 정확히 적는다. "날짜 처리"와 "스케줄링"은 다른 일이다. 범위를 좁힐수록, 우리가 들여와야 하는 라이브러리의 표면이 줄어든다. 라이브러리의 기능을 절반만 쓴다면, 대가도 절반이다. 반대로 "우리 서비스의 시간 관련 일을 전부"라고 적으면, 대가는 전부다. 질문 둘: 이게 무엇을 끌고 오는가. 전달 의존을 설치 전에 확인하고 설치 후에 확인하면 안 된다. 매니페스트의 의존 목록이 계약서다. 열 줄짜리 유틸이 30개를 끌고 온다면, 그 열 줄은 열 줄짜리가 아니다. 질문 셋: 라이선스는 무엇이고 우리 것과 맞는가. MIT나 Apache 2.0 계열은 대부분 안전하다. GPL 계열은 배포할 때 의무가 우리에게로 이어진다. 그리고 가장 잘 놓치는 경우: 라이선스가 없는 라이브러리. 라이선스 파일이 없으면, 법적으로는 "모든 권리 보유"다. 자유롭게 쓸 수 없다.

라이선스 계열	우리 쪽 의무
MIT, Apache 2.0	고지 포함
GPL 계열	배포 시 변경부 전달
라이선스 없음	원칙상 사용 불가
상업 라이선스	가격과 범위 확인

질문 넷: 제거 비용은 얼마인가. 이게 다섯 질문 중에 가장 중요한 질문이다. 라이브러리의 API 표면을 본다. 쓰는 API가 함수 몇 개라면, 제거는 쉽다. 라이브러리의 타입이 우리 도메인 모델로 흘러들고, 앱의 라이프사이클을 라이브러리가 관리한다면, 제거는 프로젝트다. 제거 비용이 작을수록, 빌림은 안전하다. 질문 다섯: 무언가 고장 나면 누가 책임지는가. 활발한 커뮤니티가 있는 라이브러리는 물을 사람이 있다. 혼자 하는 프로젝트는

유지자 하나다. 이건 지원 문제다. 품질이 같다면, 지원이 있는 쪽이 1인 팀에는 더 낫다.

2.4 다섯 질문을 한 라이브러리에 걸어 본다

서비스에서 요청 몸체의 모양을 검증해야 한다고 하자. 레지스트리에는 후보가 두 개 있다. 다섯 질문을 두 후보에 차례로 걸어 본다.

범위는 같다. 둘 다 "JSON을 스키마대로 검증한다." 전달 의존은 다르다. 후보 A는 네 개를 끌고 오고 후보 B는 없다. 라이선스도 다르다. A는 MIT다. B는 라이선스 파일이 없다. 여기에서 B는 떨어진다. 쓸 수 있느냐의 문제다. 제거 비용은 이렇게 나온다. A의 검증 함수는 세 군데에서 호출되고, 스키마는 데이터 파일에 있다. B는 검증 타입이 요청 핸들러 안으로까지 흐른다. 책임 소재는 A가 유지자 두 명에 열려 있는 이슈 열 건, B가 유지자 한 명에 여든 건이다.

걸어보기에서 주의할 것은 하나다. B가 더 편했을 수 있다는 점이다. B를 안 고른 이유는 이유다. 라이선스, 제거 비용, 책임 소재. 그리고 그 이유는 뒤에, 제거할 때 다시 확인할 수 있다. 선택의 기록이 남아 있어야, 다음 선택이 빠진다.

2.5 30분 실험 프로토콜

다섯 질문에 답하는 가장 확실한 방법은 써 보는 것이다. 다만 메인 브랜치에서 쓰면 안 된다. 격리된 곳에서 30분 실험 프로토콜을 돌린다.

단계 하나: 스크래치 브랜치나 스크래치 프로젝트를 만든다. 라이브러리는 거기서만 들인다. 단계 둘: 실제로 쓸 핵심 용례만 구현한다. 문서를 전부 읽지 않는다. 우리 서비스가 쓸 함수 두세 개를 골라, 그걸 돌린다. 핵심 용례가 돌아가지 않으면, 나머지 API가 아무리 예뻐도 상관없다. 단계 셋: 출력을 재본다. 빌드 시간과 번들 크기를 먼저, 로그 소음을 뒤에.

라이브러리 없는 상태와 비교한다. 단계 넷, 가장 중요한 것: 제거를 해 본다. 라이브러리를 지우고 무엇이 깨지는지 확인한다. 임포트 몇 개만 지우면 끝난다면, 그 라이브러리는 잠금이 낮다. 타입 이름과 설정 키까지 함께 바뀌었다면, 그 라이브러리는 잠금이 높다.

이 프로토콜의 가치는, 실제 빌림 전에 제거 비용을 드러내는 데 있다. 제거 비용을 내는 사람은 언제나 우리다. 그래서 첫 대가 검증은 30분짜리, 그리고 버려도 되는 곳에서 한다.

실험이 끝나면 판정을 내린다. 핵심 용례가 돌아갔고 제거가 쉬웠다면, 들여온다. 핵심은 되지만 제거가 어렵다면, 속도를 늦춘다. 이때 질문은 "기능이 오래된 잠금을 값으로 한다면 그만한 가치가 있는가"다. 기능의 좋고 나쁨은 그 다음 문제다. 핵심 용례 자체가 제대로 안 되면, 문서를 아무리 정성 들인 후보도 배제한다. 실험의 30분은, 뒤의 수개월을 사고 있는 시간이다.

2.6 잠금의 척도

잠금은 API를 얼마나 쓰느냐의 문제가 아니다. 라이브러리의 개념이 우리 코드로 새어나오느냐의 문제다.

새어 나오는 곳은 몇 군데 있다. 가장 흔한 곳은 이름이다. 우리 타입 이름을 라이브러리 접두사로 붙이기 시작하면, 도메인 한가운데에 라이브러리 이름이 앉게 된다. 더 큰 곳은 데이터 모양이다. 데이터베이스 테이블 구조가 라이브러리의 기대에 맞춰지면, 데이터에 라이브러리의 지문이 남는다. 가장 깊은 곳은 프로세스다. 앱의 라이프사이클, 시작과 종료, 라이브러리가 움직이면 우리가 움직이고 라이브러리가 멈추면 우리가 멈추는 상태. 설정도 흔한 새는 곳이다. 라이브러리에서 온 파일 이름, 라이브러리 맥락에서만 의미가 있는 키.

척도의 아래쪽은 순수 유틸리티 라이브러리다. 함수를 호출하고 결과를

받고 라이브러리 이름은 우리 도메인 어디에도 나오지 않는다. 위쪽은 프레임워크와 런타임이다. 앱의 모양을 그쪽이 정하고 우리 코드가 프레임워크의 구현 세부 사항이 된다. 중간은 ORM이나 HTTP 클라이언트 같은 것들이다. 데이터와 프로세스를 건드리지만 경계를 아직 둘 수 있다.

여기서 잠금과 의존성의 차이를 한 줄로 정리하자면 이렇다. 의존성은 쓸 수 있는 것이고 잠금은 안 쓸 수 없는 것이다.

위쪽을 피해야 한다는 뜻은 아니다. 프레임워크에 가치는 있다. 다만 위쪽을 고를 때는, 교체 비용이 주 단위로 재는다는 것을 알아야 한다. 시간 단위와는 차원이 다르다. 척도는 두 후보를 비교할 때 쓰인다. 핵심 용례가 같은 두 라이브러리라면, 잠금 한 단계를 아래로 고르는 것이 안전한 선택이다.

2.7 무시하면 안 되는 빨간 깃발

신호가 좋아도 빨간 깃발이 있는 라이브러리가 있다. 이런 것을 보면 멈춘다.

하나는, 사용법을 배우려면 앱을 구조 조정해야 하는 문서다. "먼저 A를 설치하고 B를 설정하고 C 미들웨어를 붙여라" 식의 온보딩은, 라이브러리가 강한 의견을 갖고 있다는 신호다. 강한 의견은 때로 특징이다. 그런데 1인 팀에게는 유지비용이 된다. 둘은, 검증할 수 없는 마법이다. "그냥 됩니다"로 설명되는 기능에서, 어디서 어떻게 동작하는지 보이지 않으면, 깨졌을 때 디버깅할 수가 없다. 셋은, 규칙을 안 지키는 버전 번호다. 체인지로그 없이 1.2에서 2.0으로 뛰는 릴리스, API가 깨진 마이너 버전. 넷은, 갑작스러운 유지자 변경이다. 새 주인과 새 라이선스로 넘어간 라이브러리는, 우리 동의 없이 계약이 다시 쓰인 상황이다.

이 중 하나만으로는 배제 이유가 되지 않는다. 다만 둘 이상 모이면, 후보 하나 더 보라. 레지스트리는 넓다.

2.8 한 개만 남았을 때

심사 프레임워크를 돌리면, 후보가 둘이 안 남는 경우가 있다. 하나만 남거나, 전부 떨어지거나.

하나만 남았으면, 포크를 확인한다. 원본이 방치된 뒤 포크가 그 역할을 이어가는 경우가 많다. 다만 포크에 대해서는 신호 여섯 가지를 포크 자체로 다시 확인해야 한다. 라이선스가 다를 수 있고 유지자는 다른 사람이다. 원본의 평판은 포크의 평점이 아니다.

전부 떨어졌으면, 출구가 두 개다. 하나는 직접 만드는 것. 범위가 좁다면, 1장의 결정 라인에서 우리 쪽이다. 열두 줄이면 열두 줄로 쓴다. 다른 하나는, 얇은 인터페이스를 먼저 쓰는 것. 직접 의존을 들여오되, 3장의 어댑터 규율대로 우리 이름의 함수 뒤에 둔다. 들여온 것이 맞지 않는 날이 와도, 파일 하나를 바꾸는 비용으로 갈아탈 수 있다.

후보가 하나뿐이라는 사실은, 더 신중히 보라는 뜻이다. 고를 필요가 없다는 뜻은 아니다. 대체제가 없으면, 그 계약의 조항 하나하나가 우리에게 더 무겁게 떨어진다.

2.9 이 장에서 만든 눈

선택의 기준은 인기가 아니다. 유지보수의 건강, 범위의 명확함, 라이선스의 호환, 제거 비용, 잠금의 높이. 이 다섯 가지다. 그리고 이 다섯 가지는 전부, 설치 전 30분 안에 확인할 수 있다.

레지스트리를 검색할 때 보는 것이 달라진다. “이 계약은 어떤가”를 보게 된다. 인기의 문제가 아니다. 오늘 고른 라이브러리는, 내년 리팩터링의 모양을 정한다. 그래서 선택은 설계의 일이다.

3 3장. 잠그고 따라가기: 고정, 업데이트, 교체의 규율

고르는 것으로는 반이 끝난다. 다른 반은 설치한 순간부터 시작된다. 라이브러리가 다음 버전을 올리고 우리는 다시 선다. 업데이트할 것인가, 안 할 것인가. 안 한다면, 언제까지. 이 장은 설치 뒤의 일, 고정과 업데이트와 교체를 다룬다.

3.1 SemVer, 현실에서

대부분의 생태계는 의미를 담은 버전 체계, SemVer를 쓴다. 버전 번호는 세 자리다. 앞자리가 메이저, 그 뒤가 마이너, 마지막이 패치.

약속은 간단하다. 패치는 버그만 고친다. 마이너는 기능을 더하지만 기존 것을 깨뜨리지 않는다고 약속한다. 메이저는 무엇이든 깨뜨려도 된다. 그래서 "마이너와 패치는 마음 편히 올리고, 메이저는 신중하게"가 표준적인 충고다.

약속과 현실 사이에는 간극이 있다. 실제로는 마이너에 깨진 변경이 숨어 들어오는 경우가 있고, 메이저라면서도 이주하면 금방 끝나는 경우가 있다. 버전 번호는 힌트다.

왜 마이너에 깨진 변경이 숨는가. 이유가 두 가지다. 첫 번째는 정직한 실수. 유지자가 동작 변경을 개선했다고 여겼고 깨짐이라고는 보지 않았다. 두 번째는 회색 지대. 문서가 "이렇게 쓸 수 있다"는 것은 말해 주면서 "이 동작에 기대지 말라"는 것은 말하지 않는 경우. 우리 코드는 기대했고 그 기대가 버그가 됐다. 두 번째 케이스에서는 버전번호가 우리를 지켜 주지 못한다. 지켜 주는 것은, 우리 동작을 코드로 만든 테스트다. 그래서 마이너 업데이트의 실질적 작업은 "테스트 스위트를 돌리고, 실패의 diff를

읽는 것"이다. 테스트가 빈약하다면, 마이너 업데이트 시점에 테스트를 먼저 늘린다. 깨진 뒤 디버깅보다 값이 싸다. 그래서 규율은 "버전 번호를 확인하는 것"이다. 업데이트 전에는 체인지로그를 읽고, 업데이트 후에는 테스트를 돌린다. 이 두 개의 비용은 몇 분이다. 이 두개로 잡는 것은 몇 주다.

하나 더 볼 것이 있다. 어떤 라이브러리는 마이너를 너무 빠르게 올려 따라가기가 어렵다. 마이너 다섯 개를 뒤처졌다면, 그건 라이브러리의 문제가 아니라 우리 업데이트 리듬의 문제다. 리듬이 없다는 뜻이다. 다음 섹션은 그 리듬을 정한다.

3.2 락파일은 계약이다

매니페스트, package.json 같은 파일은 허용하는 버전의 범위를 적는다. "2.x라면 무엇이든" 같은 것. 락파일, lock file은 설치 당시 실제로 쓴 정확한 버전을 적는다. 둘은 다른 것이며 이 둘을 혼동하는 데서 대부분의 사고가 시작된다.

매니페스트는 의도이고 락파일은 계약이다. 같은 매니페스트로, 같은 날의 빌드가 서로 다르게 나올 수 있다. 락파일은 그것이 안 되게 한다. 그래서 첫 번째 규율: 락파일을 버전 관리에 커밋한다. 1인 프로젝트여도, 내부 도구여도. "나중에 다시 생성하면 되지"가 빌드가 깨지는 시작이다.

두 번째 규율: 프로덕션에서 새 버전을 해석하게 하지 않는다. 배포 환경은 락파일에 적힌 것을 그대로 설치한다. 배포할 때마다 최신이 해석되면, 언제 무엇이 바뀌었는지 알 수 없게 된다. 세 번째 규율: 락파일과 매니페스트가 불일치하면, 락파일을 믿고 매니페스트를 고친다. 락파일을 한 해 방치했다면, 그건 이행되지 않은 계약 한 묶음이다.

두 파일의 차이를 구체적으로 보자.

```
// 매니페스트: 의도
"date-utils": "^3.0.0"

// 락파일: 계약
"date-utils": "3.2.1"
```

매니페스트의 caret 기호는 “3.x면 무엇이든”이라고 뜻한다. 새 기계에서 설치하면 3.2.1일 수도 있고 3.5.0일 수도 있다. 락파일은 3.2.1로 못한다. 우리가 테스트한 빌드는 3.2.1의 빌드이지, 3.5.0의 빌드가 아니다. 그래서 락파일이 계약인 것이다.

생태계마다 느슨한 정도가 다르다. npm은 기본 caret 범위로 마이너 업데이트를 허용하기 때문에, 락파일의 역할이 특히 무겁다. Cargo와 Go도 락파일이 있고 Python은 가장 느슨하다. requirements 파일은 하한만 적으므로, 고정된 상한을 원하면 별도 pin 파일이 필요하다. 생태계가 느슨하면, 규율은 우리 몫으로 더 커진다. 도구가 해 주지 않으니까.

한 가지만 덧붙이자. 라이브러리 프로젝트를 만드는 경우에도 락파일을 커밋한다. 애플리케이션이든 라이브러리가든, “내가 테스트한 빌드”라는 개념은 같다. 다만 정책이 다를 뿐이다. 애플리케이션은 락파일을 정답으로 쓰고 라이브러리는 사용자의 매니페스트와 충돌을 피하기 위해 허용 범위를 넓게 주는 쪽이다. 어느 쪽이든, 우리 머신의 빌드와 프로덕션의 빌드가 같다는 보장은 락파일에서 온다.

3.3 고정이 영원한 답은 아니다

영원히 고정한 것에도 대가가 있다. 업데이트하지 못한 버전은, 보안 패치를 못 받는 버전이다. 그리고 마침내 업데이트할 때, 거리가 너무 벌어져서 한 시간 일이 하루 일이 된다. 락파일의 가치는 “우리 때에 바꾼다”다. 한 해를 방치한 버전은 고정이 아니라 방치다.

그러니 고정과 다음 섹션의 업데이트 리듬은 한 쌍이다. 하나가 거리를 유지해 주고 다른 하나가 그 거리를 얼마나 가져갈지 정한다. 둘 중 하나만 하면, 둘 다의 대가를 내는 셈이 된다.

3.4 업데이트의 배치 리듬

업데이트는 매 릴리스마다 할 필요가 없다. 다만 리듬은 필요하다. 리듬은 변경의 크기로 정한다.

패치와 마이너는 배치한다. 시간대를 정해 둔다. 매주든, 배포마다든. 그 시간에 소규모 버전들을 함께 올리고, 전체 테스트 스위트를 돌려, 초록인 것만 들여온다. 포인트는 배치다. 매일 하나씩 올리면, 컨텍스트 전환 비용이 업데이트 비용보다 커진다. 한 타임 슬롯으로 몰면, 이전과 이후를 한꺼번에 비교할 수 있다.

배치를 쉽게 만들 수 있는 모양은, 월요일 아침 15분이다. 업데이트 확인 명령을 돌려, 대기 중인 것을 본다. 패치와 마이너뿐이면, 한꺼번에 올리고 전체 테스트 스위트를 돌려, 초록이면 하나의 커밋으로 뉘는다. 커밋 메시지는 한 줄이면 된다: “의존성 일괄 업데이트 (X월 X일 배치).” 메이저가 섞여 있으면, 배치에서 빼내고 별도 브랜치를 열어 시간 상한을 둔다. 15분 칸은 불어나지 않는다. 불어나지 않는 것이 배치의 포인트다.

배치를 올리기 전에, 한 가지만 더 확인한다. 그 업데이트가 우리 서비스의 핵심 경로를 건드리는지. 핵심 경로를 건드리는 의존성의 업데이트는, 배치에 섞지 않고 한 단위로 올린다. 결제, 인증, 데이터 저장 같은 경로. 그건 배치 리듬이 아니라, 배포 리듬이다. 업데이트는 고정된 가사일이고 주를 삼키는 프로젝트가 아니다.

메이저는 프로젝트로 다룬다. 배치에 섞지 않는다. 별도 브랜치를 열고 이주 자체를 그 브랜치의 일로 만든다. 체인지로그를 처음부터 끝까지 읽고 작업에 시간 상한을 둔다. 반나절에 끝나지 않으면, 한 박자 물러서서 질문을

바꾼다. “이주할 가치가 있는가.” 매년 이주에 반나절을 쓰는 라이브러리는, 1인 팀에게는 무겁다. 이때 싼 길이 교체가 되기도 하고 우리가 쓰는 부분만 직접 쓰기로 바뀌기도 한다. 그건 교체 섹션의 주제다.

교체를 언제 결정하는지, 숫자로 보는 방법이 있다. 대충이지만 쓰는 법은 이렇다. 1년에 그 의존성에 쓰는 시간. 업데이트, 디버깅, 보안 고지 확인까지 더한다. 이걸 2년 치로 두면, 그게 교체의 상한 비용이다. 새 라이브러리로 옮기는 비용이 그보다 작다면, 교체다. 그보다 크다면, 유지다 [추정].

이 산식이 주는 것은 기준선이다. “연 16시간이면, 상한은 32시간, 새 것을 들여오는 데 2주면 교체”라고 말할 수 있게 된다. 말할 수 있으면, 논쟁이 끝난다. 그리고 1인 팀에게 논쟁의 끝남은, 시간이다.

볼 것 하나 더: 업데이트의 비용 자체다. 업데이트가 손쉬우면, 코드는 건강하다. 매 업데이트마다 디버깅이 붙으면, 우리 코드가 라이브러리의 내부 동작에 결합되어 있다는 뜻이다. 업데이트가 그 결합을 드러낼 뿐이다. 그럴 때 고쳐야 할 것은 결합이다. 업데이트의 난이도는, 우리 코드의 진단 도구다.

자동화는 도움이 된다. 업데이트 풀 리퀘스트를 대신 열어 주는 도구들, Dependabot이나 Renovate 같은 것들은 분류 보조 도구로 쓸만하다. 다만 자동 병합기가 아니다. 봇이 올려 준 것을 보지 않고 들여오면, 배치 속에 숨은 깨진 변경을 나중에 발견한다. 자동화의 역할은 업데이트를 제때 우리 눈에 데려오는 것. 판단은 여전히 우리 몫이다.

3.5 라이브러리가 죽을 때: 교체의 트리거

교체가 필요한 경우는 네 가지다. 라이브러리가 방치되었을 때. 유지자가 고칠 수 없거나, 고치지 않는 보안 문제가 왔을 때. 라이선스가 바뀌어 조항이 불리해졌을 때. 그리고 업데이트 비용이 재작성 비용을 넘었을 때.

이 중 방치가 가장 흔하다. 일 년 넘게 릴리스가 없고 이슈에 답이 없고 유지자가 소식이 없는 라이브러리. 그 계약은 이미 깨진 것이 사실이다. 남은 질문은 하나다. 대가가 언제 청구되느냐.

교체의 문제는, 대부분의 코드가 라이브러리를 직접 쓰고 있다는 것이다. 임포트가 여기저기 흩어져 있고 라이브러리의 타입이 함수 서명에 있다. 교체를 하려면, 쓴 모든 곳을 건드려야 한다. 그래서 교체를 싸게 만드는 일은, 라이브러리를 고를 때나 설치할 때가 아니라, 통합할 순간에 한다.

3.6 어댑터로 격리한다

기법은 단순하다. 우리 코드와 라이브러리 사이에, 우리 것인 얇은 인터페이스를 하나 둔다. 이름은 아무거나. 어댑터든, 래퍼든, facade든. 중요한 것은 규칙이다. 라이브러리의 타입은, 우리의 인터페이스를 넘지 못한다.

로깅을 예로 하자. 로깅 함수를 여기저기서 직접 호출하면, 로깅 라이브러리가 잠긴다. 우리 작은 모듈 한 개를 사이에 두면, 이야기가 달라진다.

```
# 애플리케이션 코드는 이 파일만 안다
log.py

def log_event(kind, msg):
 # 구체적인 라이브러리는 여기서만 등장한다
 library_logger.info(msg)
```

구체적인 라이브러리 호출은 이 파일 한곳에만 있다. 다른 모든 파일은 우리의 log.py만 임포트한다. 라이브러리를 바꾸려면, 건드릴 파일이 하나다. 제거 비용이 상수로 고정된다.

같은 규칙은 결제, 메일 전송, 파일 저장, 데이터베이스 접속에도 적용된다.

기억해 둘 규칙 하나: 인터페이스는 우리 도메인의 언어를 말한다. 라이브러리의 언어가 아니라. 인터페이스에 라이브러리의 클래스 이름과 메서드 이름이 붙어 있으면, 격리가 위조다. 라이브러리가 API를 바꾸는 순간, 인터페이스도 바뀌어야 하고 격리의 이익이 사라진다. 인터페이스의 서명은, 우리 서비스가 푸는 문제의 모양이어야 한다. "주문 확인을 보낸다"이지, "notifyAsync를 호출한다"가 아니다.

이 기법의 비용은, 한 겹 더 생긴 우회다. 잠금 척도의 아래쪽, 순수 유틸리티에는 과다 투자다. 핵심 경로에 닿거나, 우리 도메인 데이터를 건너가는 라이브러리에는 값하다. 2장의 척도를 여기서 다시 쓴다. 잠금이 높을수록, 격리도 단단하게.

3.7 교체의 절차

앞에서 격리를 해 두었다면, 교체 자체는 루틴이다.

단계 하나: 같은 인터페이스 뒤에서, 새 라이브러리를 위한 두 번째 어댑터를 만든다. 단계 둘: 둘을 돌려 비교한다. 인터페이스에 대해 테스트를 먼저 써 두면, 새 어댑터가 같은 행동을 하는지 확인할 수 있다. 단계 셋: 설정이나 빌드 플래그를 바꿔, 새 어댑터를 가리킨다. 단계 넷: 프로덕션에서 돌리고 지켜본다. 낡은 어댑트는 잠시 두고 간다. 단계 다섯: 확신이 들면, 낡은 어댑터와 낡은 라이브러리를 지운다.

중요한 것은 단계 다섯이다. 낡은 라이브러리를 안 지우는 교체는 교체가 아니라, 복제다. 의존성 트리에 라이브러리가 둘이 공존하면, 학습 대가도 보안 대가도 두 배가 된다. 교체의 끝은 언제나 제거이며 제거는 다음 장의 주제다.

3.8 이 장의 규율

버전 번호는 힌트이고 테스트가 확인한다. 락파일은 계약이고 커밋되며 다시 해석되지 않는다. 업데이트는 배치로 오르고 메이저는 시간 상한을 두고 프로젝트로 다뤄진다. 그리고 라이브러리의 짐이 가치를 넘기면, 교체한다. 교체를 싸게 하는 일은 통합 순간에 한다.

전부의 포인트는 하나다. 변화를 당하지 않는 것. 변화의 시기도, 변화의 비용도, 우리가 정한다.

4 4장. 제거하는 기술: 의존성 위생 루틴

좋은 의존성은 잘 동작하는 것이 아니다. 빼고 싶을 때, 깨끗이 빼낼 수 있는 것이다. 제거는 빌림의 마지막 시험이다. 빼지 못한다면, 그 시스템의 주인은 우리도 그 라이브러리도 아니다. 계약서 속의 조항이다.

이 장은 세 가지를 다룬다. 특정 라이브러리를 안전하게 빼는 5단계 절차. 반쯤 지워진 상태를 다루는 법. 그리고 더 이상 필요 없는 것을 찾아내는 분기 루틴.

제거를 해 본 사람은 안다. 제거는, 설치할 때보다 우리 코드를 더 깊이 알아야 한다는 것을. 설치는 남의 코드를 들여오는 일이고, 제거는 우리 코드가 그 남의 코드를 어디까지 알고 있는지 전부 드러내는 일이다. 그래서 제거는 공포라기보다 검사에 가깝다. 잘 하면, 시스템의 약한 곳이 한 장의 목록으로 나온다.

4.1 왜 제거가 이렇게 어려운가

제거가 설치보다 어려운 데는 이유가 있다. 설치는 참조를 추가하는 일이고 제거는 그 참조를 전부 찾아내는 일이다. 참조는 네 군데에 숨는다.

임포트에는 드러난다. 도구가 찾아 준다. 설정에는 숨는다. 라이브러리에서 온 이름의 키, 환경 변수, 라이브러리가 있을 때만 의미가 있는 플래그, 라이브러리를 참고한 주석. 락파일에는 남는다. 매니페스트 항목을 지워도, 간접 의존이 라이브러리를 다시 끌어 올 수 있다. 다른 라이브러리가 그걸 의존하기 때문이다. 그리고 가장 깊은 곳, 우리 머리에는 잔상이 남는다. 라이브러리의 기대에 맞춰진 타입 이름, 함수 이름, 데이터 모양, 코드 순서까지. grep으로는 안 보인다. 코드의 흔적으로 남는다.

그래서 “패키지만 지웠다”는 제거가 끝나지 않는다. 패키지는 사라졌는데,

그 그림자는 남는다. 그리고 그 그림자가 다음 제거를 더 어렵게 만든다.

4.2 고스트 의존성

제거를 시작하고 락파일을 재생성했는데, 의존 개수가 그대로인 경우가 있다. 바로 고스트 의존성이다. 직접 의존은 지웠는데, 다른 라이브러리가 여전히 끌어오고 있어서 트리에 그대로 있는 것이다.

고스트는 세 갈래로 갈린다. 첫 번째 갈래: 끌어오는 쪽의 라이브러리가 업데이트되면, 자연히 사라진다. 이때 할 일은 하나다. 부모 라이브러리를 최신으로 올려 본다. 두 번째 갈래: 부모가 오래됐고 업데이트가 어렵다. 그러면 부모의 교체를 검토해야 한다. 우리가 직접 쓰는 것은 아니지만 트리의 무게는 우리가 치르는 대가다. 세 번째 갈래: 고스트가 보안 고지의 대상이다. 직접 쓰지 않는다고, 공격 표면에서 빠지는 것은 아니다. 그 경우, 끌어오는 쪽의 교체를 가장 우선으로 올려야 한다.

고스트를 찾아내는 방법은 역방향 추적이다. 의존성 트리 명령으로, "누가 이걸 의존하는가"를 거꾸로 읽는다. 우리 코드에서 삭제한 것이 트리에 남아 있다면, 그 잔류는 반드시 누군가의 의존 목록 안에 있다. 그 누군가를 찾는 것이, 제거의 진짜 끝이다.

4.3 안전한 제거, 5단계

제거는 5단계로 한다. 단계를 건너뛰면, 반쯤 지워진 상태가 된다.

단계 하나: 참조 매핑. 지우기 전에, 참조를 전부 찾는다. 라이브러리 이름을 코드 전체에서 검색한다. 설정 파일, 배포 스크립트, CI 설정, 문서까지.

```
# 참조 매핑: 라이브러리 이름으로 전부 찾아낸다
grep -rn "date-utils" src/ config/ scripts/ docs/
```

그리고 이 결과를, 어딘가에 붙여 둔다. 다음 단계에서 확인할 목록으로

쓴다. 매핑 없이 지우면, 지운 줄로 착각한다. 락파일도 확인한다. 다른 것의 간접 의존이기도 한지. 그렇다면, 제거 대상은 라이브러리가 아니라 "우리의 그 사용"이다.

단계 둘: 코드 제거. 호출과 임포트를 지운다. 어떤 기능이 그 라이브러리에 기대고 있었다면, 먼저 우리 코드나 다른 라이브러리로 대체한 뒤에 지운다. 제거와 대체는 별개의 일이다. 동시에 하면 안 된다.

단계 셋: 매니페스트와 락파일. 매니페스트 항목을 지우고 락파일을 재생성한다. 트리의 개수가 줄었는지 확인한다. 개수가 안 줄었다면, 다른 라이브러리가 끌어오는 것이므로, 단계 하나로 돌아간다.

단계 넷: 클린 빌드와 테스트. 기존 기계의 증분 빌드가 아니다. 신선한 클론, 깨끗한 환경에서의 빌드다. 증분 빌드는 오래된 산출물을 숨긴다. 클린 빌드야말로, 제거가 끝났는지 확인할 유일한 방법이다.

단계 다섯: 측정. 빌드 시간, 번들 크기, 의존 개수. 이전과 이후를 비교해서, 어딘가에 기록한다. 이 숫자는 제거가 값했다는 증거이며, 다음 리뷰에서 다시 볼 자료다.

제거 비용의 신호를 미리 아는 것도 방법이다. 시작 전에 아래 세 가지를 보면, 이번 제거가 가벼운지 무거운지 대략 안다.

신호	읽는 법
참조 수	건드릴 곳이 몇 개인가
타입 깊이	새어 나온 곳이 몇 겹인가
간접 의존	지워도 다시 오는가

세 개가 전부 작으면, 이번 제거는 반나절 안에 끝난다. 하나라도 크면, 제거를 프로젝트로 보고 브랜치를 따로 만든다.

단계 넷에서 실패하면, 디버깅하면서 임시 패치하지 않는다. 그 커밋을

되돌리고 단계 하나로 돌아간다. 반쯤 지워진 제거는, 지우지 않은 것보다 나쁘다. 보이는 깨짐은 숨은 깨짐보다 싸다.

4.4 반쯤 지워진 상태, 다루는 법

실제로 가장 흔한 사고는, 반쯤 지워진 제거다. 증상은 대개 이런 모양이다. 로컬에서는 빌드가 되는데, CI에서는 깨진다. 로컬의 증분 빌드는, 지운 라이브러리의 오래된 산출물을 아직 쓰고 있다. CI는 신선한 환경이라, 그 산출물이 없다. 그래서 로컬이 더 거짓말을 한다.

두 번째 증상은, 테스트는 다 초록인데, 특정 경로에서만 깨지는 경우다. 그 경로가 라이브러리를 쓰고 있었고, 참조 매핑에서 놓쳤다. 주석으로만 남아 있던 참조, 환경 변수로만 남아 있던 설정, 배포 스크립트 한 줄.

회복 절차는 단순하다. 먼저, 제거 커밋을 되돌린다. 동작하던 상태로 돌아가야, 다음 제거의 기준점이 생긴다. 다음으로, 깨진 것에서 힌트를 뽑는다. 에러 메시지에 나온 파일 이름과 함수 이름이, 놓친 참조의 장소다. 그 힌트로 참조 매핑을 다시 돌린다. 그리고 다시 제거한다. 이번에는, 힌트를 매핑 결과에 합쳐서.

반쯤 지워진 상태를 두려워하지 않아도 된다. 두려워해야 할 것은, 되돌리지 않는 것이다. 되돌림이 쉬운 상태, 즉 커밋 단위가 작은 상태라면, 실패의 대가는 몇 분이다. 그래서 제거도, 다른 코드 작업처럼 작은 커밋으로 나눈다. 코드 제거, 매니페스트 제거, 락파일 재생성, 각자 커밋. 되돌리고 싶은 단위를, 커밋 단위로 만든다.

4.5 분기 30분 리뷰

제거는, 라이브러리가 죽었을 때의 대응만이 아니다. 대부분의 불필요한 의존은 조용히 죽는다. 알리지도 않고 필요없어진 것만 남는다. 서비스는

진화했고 라이브러리의 일은 사라졌다. 라이브러리만 트리에 있다. 분기 리뷰가 그걸 찾는다.

루틴은 단순하고 1장의 재고 시트로 한다.

시간은 30분을 정해 둔다. 주에 한 번이 아니어도 된다. 분기다. 분기라면, 1년에 네 번. 30분은 지켜질 수 있는 크기다. 지켜지지 않는 루틴은 없는 루틴과 같다.

분기를 건너뛴 일이 있다면? 벌을 줄 필요는 없다. 다음 리뷰에서 "건너뛴 분기" 항목을 하나 더 두면 된다. 건너뜀이 한 번이면 사고가 아니다. 두 번이면, 희망이 아니라 루틴이 된다. 지킬 때의 기록이 쌓이는 것이 핵심이다.

단계 하나: 직접 의존성을 차례로 훑는다. 단계 둘: 각자에게 질문한다. "너는 들여올 때 맡은 일을 아직 하고 있나?" 일이 바뀌었거나 사라졌다면, 표에 마크를 한다. 단계 셋: 바깥 신호를 확인한다. 마지막 리뷰 이후에 메이저가 올라왔는가. 릴리스가 멈춘 것은 아닌가, 18개월 넘게 안 왔다면 [추정]. 취약성 고지가 있었는가. 유지자가 바뀐 것은 아닌가. 단계 넷: 분류한다.

판정	기준
유지	쓰고 있고, 짐이 적다
관찰	쓰는데, 신호가 하나 있다
교체	짐이 크고, 후보가 있다
퇴장	필요 없다, 지금 뺀다

분류의 정확도보다, 분류를 했느냐가 중요하다. "유지"를 "관찰"로 틀려 봐도, 다음 분기에 한 번 더 보는 일이다. 위험한 것은 분류 안 된 것이다. 아무도 안 보는 의존성은, 아무도 모르게 죽어 가는 의존성이다.

"퇴장" 판정은, 5단계 절차를 같은 주에 돌리는 것이다. 표 위에 한 달

남아 있는 판정은, 돌아오지 않는 판정이다. 제거는 기한이 있는 작업이지, 메모가 아니다.

예시로 하나 보자. 분기 리뷰에서 로깅 라이브러리를 만난다. 들여온 지 두 해, 지금은 플랫폼이 구조화 로그를 받아 준다. 그 라이브러리의 남은 일은, 로컬 개발 때 콘솔에 찍어 주는 것. 질문을 건다. "들여올 때 맡은 일을 아직 하고 있나." 답: 반쯤 한다. 바깥 신호도 본다. 릴리스는 온다, 단, 마이너만. 취약성 고지는 없다. 분류: 관찰. 다음 분기에, 콘솔 로깅을 우리 코드의 열 줄로 대체하면, 퇴장이다. 제거 비용은 재고 시트에 이미 적혀 있다. 반나절.

같은 시트에서 다른 라이브러리를 만난다. 결제 대행. 릴리스가 20개월 전으로 멈췄고, 유지자의 계정에 최근 활동이 없다. 질문은 필요 없다. 분류: 교체. 후보가 아직 없어도, 교체의 시작은 "후보 탐색"이다. 재고 시트의 교체 칸에 후보를 쓰는 순간, 관리 중인 의존성이 된다.

4.6 1페이지 체크리스트

이 책 전체를 한 페이지로 압축한 것이다. 신규 프로젝트 시작할 때, 그리고 분기마다 돌린다.

고르기 전에: - 범위를 정확히 적었는가, 대체하는 것이 무엇인지 - 유지보수 신호 여섯 가지를 확인했는가 - 라이선스를 확인했는가, "없음"이 아닌가 - 30분 실험을 했는가, 제거까지 해 봤는가 - 잠금의 높이를 알 수 있는가

통합한 뒤: - 락파일을 커밋했는가 - 인터페이스가 우리 것이는가, 라이브러리 타입이 우리 인터페이스를 안 넘나 - 업데이트 배치의 시간대를 정했는가

분기마다: - 재고 시트를 갱신했는가, 제거 비용과 마지막 갱신 - 각 의존성을 분류했는가, 유지, 관찰, 교체, 퇴장 - "퇴장"들을 전부 실행했는가 - 제거 결과를 측정하고 기록했는가

체크리스트는 완벽하지 않다. 다만, 라이브러리가 조용히 죽지 못하게 할 만큼은 충분하다.

돌리는 시기도 정해 둔다. 신규 프로젝트라면, 첫 의존성을 들이기 전에 "고르기 전에" 항목을 전부 닫고 시작한다. 기존 프로젝트라면, 분기 리뷰 첫날에 "분기마다" 항목을, 그리고 리뷰가 끝날 때쯤 "통합한 뒤" 항목을 확인한다. 체크하는 것이 핵심이다. 체크를 안 한 항목은, 없는 항목과 같다.

4.7 빌려 쓰는 코드도 우리 코드다

제목으로 돌아온다. 빌려 쓰는 코드도 우리 코드다. 하지만 그것은, 라이브러리의 소스가 우리 것이라는 뜻이 아니다. 쓰는 결과물이 우리 것이라는 뜻이다. 업데이트의 비용, 학습의 비용, 보안의 비용, 제거의 비용. 전부 우리 계좌에 찍힌다.

그러니 이 책의 규율을 한 줄로 압축하면, 이렇다. 설치는 계약이다. 계약 순간에 조항을 기록한다. 대가, 제거 비용, 잠금. 나머지는 유지다. 체인지로그를 읽고 테스트를 돌리고 표를 갱신하고 때 되면 빼는 것.

내일 하나를 하자. 재고 시트에서 제거 비용이 가장 큰 라이브러리를 골라, 단계 하나, 참조 매핑만 해 본다. 하나면 된다. 가장 무거운 의존성 하나를 참조까지 매핑해 보면, 시스템이 어디에 약한지 알게 된다. 그리고 다음 설치에서, 가장 조심해야 할 것이 무엇인지도.

빌림은 공짜가 아니다. 다만, 대가를 기록한 빌림은, 관리할 수 있는 빌림이다.