제9장 모듈성: 분리와 재조합을 위한 기준

- 문제 해결을 위해 실험적인 접근 방식의 선택은 매우 중요하다 소프트웨어 개발은 의식적으로 사용자가 무엇을 좋아할지 추측하는 기술 연습으로 진행된다.

모듈성은 우리가 만드는 시스템의 복잡성을 관리하는 데 있어 매우 중요하다.
현대적인 소프트웨어 시스템은 방대하고 복잡하며 종종 정말 정교하다 이런 복잡성에 대처하려면 우리가 구축하는 시스템을 더 작고 이해하기 쉬운 조각, 즉 시스템의 다른 곳에서 일어나는 일에 대해 너무 걱정하지 않고 집중할 수 있는 조각으로 나눠야만 한다

이런 '가이드 레일'이 우리 설계 과정에서 정직함을 유지하는데 중요하다는 것이다

이런 '가이드 레일' : 20~30줄보다 긴 코드가 포함된 메서드가 있는 커밋은 거부, 매개변수가 5~6개를 넘는 메서드 서명도 거부

9.1 모듈성의 전형적인 특징

- 모듈은 프로그램에 포함될 수 있는 명령어와 데이터의 집합이라고 할 수 있다. 이는 모듈을 구성하는 비트와 바이트의 '물리적' 표현을 포착한다

코드를 작은 구획으로 나누는 것 각 구획은 다양한 컨텍스트에서 여러 번 재사용될 수 있고, 모듈 내에 위치한
코드는 유용한 작업을 수행하기 위해 시스템의 다른 부분을 요구하더라도 시스템의 다른 부분의 컨텍스트를 벗어나 독립된 단위로 쉽게 이해할 수 있을 만큼 충분히 짧다

접근을 제어하고, 다른 코드와 통신을 관리하고, 다른 모듈을 처리하는 일종의 인터페이스가 있다.

9.2 설계는 언제나 중요하다

- 많은 소프트웨어 개발자가 이런 아이디어에 관심을 기울이지 않는다. 업계에서는 소프트웨어 설계의 중요성을 과소평가해 왔었다

모듈성을 위한 설계는 프로그래밍 언어의 구문을 아는 것과는 분명히 다른 유형의 기술이다.

어느 정도 숙달되기를 원한다면 계속 노력할 필요가 있는 기술이며, 평생을 들여도 결코 완벽해질 수 없는 기술이다.

TDD 강연중에 한 수강생이 코드가 덜 복잡한 것이 왜 중요한지 물었을 때 질문한 사람이 모호하고 복잡한 코드와 명확하고 단순한 코드 사이의 영향과 가치의 차이를 보지 못해서 충격을 받았다.

근본적으로 복잡성은 소프트웨어의 소유 비용을 증가시킨다. 여기에는 직접적으로 미치는 경제적인 영향뿐만 아니라 더 주관적인 영향도 있다 복잡한 코드는 작업하기에 좋지 않기 때문에 하지만 여기서 진짜 문제는 복잡한 코드는 정의상 변경하기가 더 어렵다

우리가 작성하는 코드의 복잡성을 제한하기 위해 작업하면 실수를 저지르고 이를 수정할 가능성이 훨씬 더 높아질 수 있다

9.3 TDD의교훈: 테스트가 어렵다면 설계도 문제다

- 익스트림 프로그래밍 읽고 TDD의 초기 수용자였던 글쓴이 TDD는 글쓴이 경력을 통틀어 소프트웨어 개발 실무에서 가장 중요한 단계 중 하나다. 중요하게 여기는 이유는 우리가 일반적으로 생각하는 '테스트'와는 거의 관련이 없다는 점이 혼란을 불러일으킨다

테스트를 통해 설계를 주도하려 시도하지 않는 한 품질을 나타내는 객관적 인 척도가 없다

테스트가 작성하기 어렵다면 설계가 잘못되었음을 의미한다

TDD의 실패 성공, 리팩토링 규율을 따르면 이런 교훈은 자동으로 우리에게 전달된다. 테스트를 작성하기 어렵다면 설계의 품질이 원래보다 떨어진다

9.4 TDD로 모듈성을 장화하자

- 테스트 가능성을 달성하기 위한 설계는 어떻게 모듈성을 향상할 수 있을까

예를 들어 비행기 날개의 익면airfoil 효과를 테스트하고 싶으면 비행기를 만들어서 날려볼수 있다.

이와 같이 다소 순진한 접근 방식을 취하면 무언가를 배우기에 앞서 모든 작업을 먼저 수행해야만 한다.

모든 변수를어떻게 관리할 수 있을까? 이 문제를 해결하기 위해 이런 전체 시스템에 폭포수 접근 방식을 취하면 된다

9.5 REST API로 모듈성을 강화하자

- 서비스라는 개념은 프로그래밍 언어가 직접 제공하지 않지만, 소프트웨어 설계에서 널리 사용된다

실용적으로 서비스는 다른 코드에 기능을 제공하면서 내부 구현을 숨기는 코드이며, 이는 정보 은닉의 중요한 방식이다.
시스템 설계에서는 봉합 지점**(seam)** 을 설정해, 그 지점의 반대편에서 일어나는 세부 사항을 알 필요 없도록 만드는 것이 핵심이다. 서비스는 시스템 내에서 세부 사항을 감추는 모듈로 볼 수 있으며, 서비스 경계는 **알려진 것(외부에 노출된 것)** 과 **숨겨진 것(내부 구현)** 의 차이를 만든다
대규모 코드에서 흔히 나타나는 문제는 이러한 **경계 구분이 무너지는 것**이며, 동일한 메서드 호출과 동일한 데이터 구조가 경계를 넘나들며 사용되는 경우가 많다. 이런 경우 입력 검증, 출력 변환 등이 이뤄지지 않아 코드가 복잡하게 얽히고 수정이 어려워진다.

REST API의 도입은 이런 문제에 진전된 접근을 제공하는데, 서비스 가장자리에 **변환 지점(입·출력 변환 단계)** 을 두는 방식이 유용하다. 하지만 여전히 일부 시스템에서는 HTML 등을 그대로 전달하거나 내부에서 직접 사용하여 경계를 모호하게 만드는 문제가 있다. 봉합 지점/경계는 **정보 변환 및 검증 지점**이 되어야 하며, 서비스 진입점은 **오용을 방지하는 방어벽** 역할을 해야 한다.

이는 서비스 통신 방식(메서드 호출, HTML, XML, 메시징 등)에 관계없이 적용되며, 일종의 **포트와 어댑터 모델**과 유사하다.핵심은 **모듈성 유지**이며, 모듈의 내부 동작이 외부에 노출되면 모듈성이 사라지고, 모듈 간 통신은 모듈 내부 통신보다 강화된 보호가 필요하다.

9.6 배포 파이프라인으로 모듈성을 강화하자

- 글쓴이는 「Continuous Delivery」에서 **항상 릴리스 가능한 상태로 소프트웨어를 유지하는 작업 방식**을 제안했다. 이를 위해 **배포가 쉽고 간단해야** 하며, 소프트웨어가 **테스트 가능하고 배포 가능하도록 작업하는 방식**이 작업 품질에 큰 영향을 준다고 강조한다

배포 파이프라인의 개념

배포 파이프라인은 **커밋에서 시작해 릴리스 가능한 결과물을 만들어내는 메커니즘**이다.

단순 빌드 또는 테스트가 아니라 **커밋에서 서비스까지 이어지는 자동화된 경로**를 의미한다.

파이프라인에서 모든 것이 “양호”하면, 추가적인 승인이나 테스트가 없어도 **즉시 릴리스 가능해야 한다**.

릴리스 가능성과 모듈성

릴리스 가능한 결과물은 **독립적으로 배포 가능해야** 한다.

파이프라인 결과가 다른 파이프라인의 결과에 의존한다면 **평가 범위가 불명확해지고 주기 시간이 길어져** 문제가 발생한다.

소프트웨어 개발에서 가장 확장 가능한 접근 방식은 **작업 단위를 분배하고 팀 간 의존성을 줄이는 것**이다.

독립성의 필요성과 마이크로서비스

각 팀이 다른 팀에 의존하지 않고 **독립적으로 빌드·테스트·배포**할 수 있어야 한다.

이를 가능하게 하는 기술적 접근 방식이 **모듈성을 높여 각 모듈이 독립적으로 동작하도록 만드는 것**, 즉 **마이크로서비스**다.

마이크로서비스는 **독립적으로 배포 가능**해야 하며, 다른 서비스와 함께 테스트해야 하는 구조라면 마이크로서비스가 아니다.

두 가지 전략

전체 시스템을 함께 빌드·테스트·배포하여 의존성을 제거

그러나 빠른 피드백을 제공하려면 큰 공학적 투자가 필요.

각 모듈을 독립적으로 빌드·테스트·배포

범위가 작아 빠르고 고품질의 결과를 얻기 쉬움.

모듈성의 설계 요구

독립성을 유지하려면 모듈 간 상호작용과 프로토콜이 안정적이어야 한다.

API 버전 관리 같은 기술도 필요하다.

모듈성은 비용이 들며 설계 노력이 요구된다.

중간 지점의 부재

많은 이들이 두 극단 사이의 이상적인 중간을 원하지만 **현실적 중간 지점은 존재하지 않는다**.

중간 지점은 오히려 단일 구조보다 더 느리고 복잡해질 수 있다.

마이크로서비스는 소프트웨어를 확장하는 가장 좋은 방법이지만 **복잡성과 비용을 감수해야 한다**.

9.7 모듈성의 규모는 크고 작음이 없다

- 모듈성은 모든 규모에서 중요하며, 배포 가능성이 시스템 수준의 모듈을 이해하는 데는 유용하지만 **고품질 코드를 만드는 데는 충분하지 않다**.현대에는 서비스 중심 사고가 강하지만, 모듈성은 지난 수십 년 동안 시스템 설계의 핵심 도구로 활용되어 왔다.시스템 수준에서만 모듈성을 적용하면 **여전히 작업하기 어려운 부실한 시스템이 될 수 있다**.

코드 수준에서도 모듈성 필요

모듈성은 **가독성 높은 코드**를 위해서도 적용되어야 한다.

클래스, 메서드, 함수는 **단순하고 읽기 쉬우며**, 필요할 경우 **더 작은 하위 모듈로 분해**돼야 한다.

TDD는 이러한 세분화된 모듈 구조를 촉진하며, 테스트를 위해 의존성 주입 같은 기법도 사용된다.

의존성 주입은 **테스트 가능하고 모듈화된 설계**를 유도하는 효과적인 도구로 설명된다.

모듈화된 설계에 대한 비판과 반론

일부는 의존성 주입이나 모듈식 설계가 코드 표면적을 넓혀 **제어 흐름 파악을 어렵게 한다고 비판**한다.

그러나 이는 핵심을 놓친 것으로, 테스트를 위해 필요한 표면적은 원래 코드가 가져야 할 실제 표면적이라는 점을 강조한다.

표면적을 숨긴 채 테스트가 부족한 설계가 오히려 **이해하기 더 어렵다**고 지적한다.

테스트의 역할

테스트는 코드·설계·문제의 본질을 드러내며, 다른 방법으로 얻기 힘든 중요한 정보를 제공한다.

결과적으로 **테스트는 더 나은 모듈식 시스템과 코드를 만드는 핵심 도구**이다.

9.8 고성과 개발 조직의 특징: 모듈형

- 고성과 개발 조직에서는 **모듈성이 중요**하다. 대규모 컴퓨터 시스템에서 반복적으로 제기되는 질문은 사실상 “더 많은 인력을 투입해 더 빨리 소프트웨어를 만들 수 있는가?”라는 것인데, 인력 추가에는 근본적 한계가 있다. 프레드 브룩스의 비유처럼 많은 인력이 있다고 해서 작업이 더 빨라지는 것은 아니다.

병렬화의 핵심은 **결합도를 낮추는 것**이다. 구성 요소들이 서로 독립적이라면 원하는 만큼 병렬로 작업할 수 있지만, 결합도가 높으면 **통합 비용** 때문에 병렬화가 제한된다. 가장 좋은 병렬화 방식은 **통합이 필요 없는 방식으로 나누는 것**이며, 마이크로서비스는 이러한 접근 방식의 대표적인 사례다.

4,000개 이상의 프로젝트 분석 결과, **작은 팀(5명 이하)** 은 **큰 팀(20명 이상)** 보다 인당 생산성이 약 4배 높았고, 9개월 프로젝트에서는 큰 팀보다 단지 1주일만 더 걸렸다. 따라서 소규모 팀이 효율적으로 일하기 위해서는 팀 간 결합을 최소화해야 한다.

결론적으로, **조직을 확장하려면 최소한의 조정만 필요한 모듈형 팀과 시스템을 구축해야 하며**, 모듈성을 유지하는 노력은 확장 가능하고 고성과 조직의 핵심 특징이다.

정리: