9장 모듈성: 분리와 재조합을 위한 기준

모듈성

시스템의 컴포넌트를 분리하고 재조합할 수 있는 특성

유연성 + 다양한 사용이라는 이점

저자가 선택한방식 : 설계에서 좋은 모들성을 달성하기위한 기술

“지속적인 통합과 배포 파이프라인의 커밋단계에서”
20~30줄보다 긴 코드가 포함된 메서드 커밋 거부 , 메개변수가 5~6개를 넘는 메서드 서명 거부

설계는 언제나 중요하다

모듈성이 좋고 관심사가 잘 분리된 코드

1. 해결하려는 문제에 더 많이 알게됨
2. 작업하기 더 쉬움
3. 테스트하기 더 쉽고
4. 수정하기 더 쉬워짐

복잡한 코드

1. 변경하기가 어렵다.
2. 코드를 잘 이해하지 못했다.
3. 실수가 숨어있을 만한 곳도 훨씬 더 많다.

모듈성을 위한 설계는 프로그래밍 언어의 구문을 아는 것과는 다른 유형의 기술

적절히 추상화된 시스템을 어떻게 만들수 있을까?

TDD의 교훈: 테스트가 어렵다면 설계도 문제다

코드를 작성하기 쉬운 테스트는 테스트 대상에서 우리가 고품질 코드의 전형적인 특징으로 중요하게 여기는 속성을 필연적으로 드러낸다. → “실패”, “성공”, “리팩토링”

휼륭한 설계 만듬 = 테스트 주도 설계 접근 방식

설계자의 기술과 경험에 의존

TDD로 모듈성을 강화하자

전체 시스템을 테스트하려 하면 너무 복잡해지고, 무엇이 문제인지 알 수 없다.

하지만 시스템을 “모듈”로 쪼개면, 측정이 정밀해지고 문제 원인을 명확히 알 수 있다

1. 통제가능하다.

비행기 전체를 테스트하면 아래 변수를 통제할 수 없음:

- 바람 세기
- 온도
- 압력
- 기타 구성 요소(엔진, 조종 장치)
- 복잡한 상호작용

그러면 정확한 원인(원자 단위)을 확인할 수 없다.

그래서 날개만 떼서 풍동에서 테스트하면:

- 변수를 통제할 수 있고
- 반복 가능한 결과를 얻을 수 있고
- 작은 단위에서 정밀하게 문제를 확인 가능

즉, 모듈은 전체보다 테스트 가능성이 훨씬 높다.

2. 단, 전체 시스템에서는 부적합하다. (변수통제 어려움)

시스템 A → B → C의 구조를 예로 듭니다.

```
A → B → C
```

B를 테스트해야 하는데 A와 C가 방해합니다:

- A가 이상한 메시지를 보내면?
- C가 응답을 멈추면
- A 또는 C 버전이 바뀌면?
- 장애 위치가 어디인지 알 수 없음

즉, 전체 시스템을 테스트하면 원인이 모호한 실패만 발생.

그래서 End-to-End(E2E) 테스트는 “모든 게 잘 돌아가는지 확인하는 용도지

병목이나 버그를 찾는 용도로는 매우 부적합”하다는 걸 강조

3. 위의 테스트의 문제점에 벗어나려면?

테스트 대상 시스템(SUT)을 정확하게 평가하려면 측정 지점(point of measurement)이 필요하다.

- 즉, 시스템 내부에 테스트 데이터를 주입하고
- 내부 동작을 관찰하고
- 결과를 정확하게 수집할 수 있는 지점을 말함.

측정값은 신중하게 받아들여야 한다. 결정론적이어야 한다.

→ 테스트가 믿을 만하려면

- 같은 버전
- 같은 테스트
- 같은 입력

  이면 얼마나 많이 반복해도 무조건 같은 결과가 나와야 한다

ex) 금융 거래소 구축

> 금융 시스템을 만들었을 때 테스트를 잘하기 위해 입력을 기록하고
>
> 그 입력을 재생(replay)하면 시스템의 상태가 실제 운영 환경과 완전히 동일하게 복원 될 정도로
>
> 100% 결정론적이었다
>
> 외부 시스템을 모두 mock으로 대체해 '테스트 가능한 경계'를 만들면, 거대한 시스템도 입력/출력만 보고 블랙박스 테스트할 수 있다.
>
> - 실제 은행 API → mock 은행
> - 실제 정산 시스템 → mock 정산 시스템
> - 실제 메시지 큐 → mock 큐
>
> → 테스트가 잘 되도록 만들다 보니 시스템이 엄청 잘 설계되어 버렸다.

즉 모듈화를 통해 측정하려는 대상에대한 통제력을 높이고, 테스트를 더 정밀하고 재현 가능하게 만든다.

REST API로 모듈성을 강화하자

서비스란 내부 세부사항을 숨기고, 외부에는 안정된 인터페이스만 제공하는 “경계가 명확한 모듈”이다.

이 경계가 지켜지지 않으면 시스템 전체가 스파게티가 된다.

REST API는 이 경계를 무조건 만들도록 강제하는 계기가 되었다.

서비스나 API의 가장자 리에 변환 지점을 두는 아이디어는 오히려 강력하게 권장

들어오는 데이터와 나가는 데이터를 서비스 내부가 이해하기 쉬운 형태로 변환

- 서비스/모듈 경계
  - 각 서비스가 외부와 내부를 나누는 경계
  - 내부 구현을 외부에 노출하지 않고, 외부는 정의된 인터페이스만 사용
  - 역할: 모듈성 보호, 복잡성 격리
- 변환 지점 (Transformation Layer)
  - 외부 입력 → 내부 데이터 모델
  - 내부 데이터 모델 → 외부 메시지/응답
  - 목적:
    - 내부 구조를 외부에 노출하지 않음
    - 데이터 정리, 검증, 단순화
    - 다른 시스템 변화에 대한 방어벽 역할
  - 예: 금융 거래소에서 외부 계좌 등록 메시지를 내부 처리 모델로 변환
- 어댑터 (Adapter)
  - 외부 시스템과 내부 서비스 사이의 중간 계층
  - 책임:
    - 외부 메시지를 내부 모듈이 이해할 수 있는 구조로 변환
    - 내부 데이터를 외부 API 포맷으로 변환
    - 외부 변화가 내부 모듈에 직접 영향 가지 않도록 격리
  - 예: REST API, 메시지 큐, SOAP, 외부 서비스 연동
- 인터페이스 (Interface / Contract)
  - 내부 모듈과 어댑터 간의 계약
  - 역할:
    - 입력과 출력의 형식과 규약을 정의
    - 내부 구현 변경 시 외부 영향 최소화
    - 테스트 가능성 확보
  - FE, BE, 다른 시스템 간 공통 모델을 정의할 때 활용

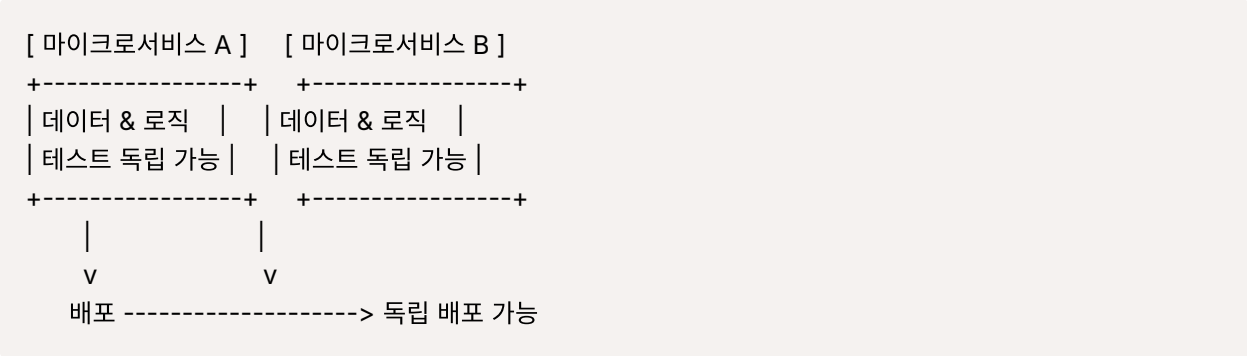
즉 서비스 경계에서 변환 지점을 두고, 어댑터와 인터페이스를 활용하면 내부 모듈을 외부 변화로부터 격리하고, 모듈성·안정성·테스트 가능성을 모두 확보할 수 있다.

배포 파이프라인으로 모듈성을 강화하자

배포 파이프라인은 단순히 자동화된 절차가 아니라 소프트웨어 설계 원칙과 직결된다.

항상 릴리스 가능하게 설계하려면, 소프트웨어를 독립적으로 배포 가능한 단위로 모듈화하고, 그 단위를 검증하고 배포하는 파이프라인을 구성해야 한다.

ex) 마이크로서비스



만약 A와 B를 항상 함께 테스트해야 한다면 → 독립성이 깨진 상태

모듈성의 규모는 크고 작음이 없다

클래스, 메서드, 함수 → 읽기쉬워야함, 독립적인 작은 하위모듈로 구성

의존성주입→ 코드에 압력을 가해 여러작은 조각으로 구성된 시스템을 만들어야함

즉 복잡성을 잘 관리할 수 있는 설계에 집중하면, 그 자체로 코드가 높은 품질인지 판단할 수 있는 기준이 된다

복잡성을 잘 관리하는 설계란 ?

모듈화, 정보 은닉, 낮은 결합도와 높은 응집도, 명확한 인터페이스, 테스트 가능성을 갖춘 구조로, 변경과 확장에 유연하게 대응할 수 있는 설계

고성과 개발 조직의 특징: 모듈형

좋은 소프트웨어 설계와 조직 구조는 모두 모듈화와 독립성을 기반으로 해야 한다

모듈성 = 독립성 = 확장성

> 큰 조직에서 단순히 인력을 늘린다고 소프트웨어 생산성이 비례해서 올라가지 않음.
>
> - 브룩스의 법칙: 여자 9명이 한 달 만에 아기를 낳을 수 없다.
> - 인력을 늘려도 통합 비용과 의사소통 비용 때문에 생산성 증가가 제한됨.

해결책: 소프트웨어를 독립적인 모듈로 설계하여 병렬로 개발 가능하게 만들기.

정리

모듈성은 복잡성을 방어하는데 필요한 도구 모음 중 첫번째이다.

다른곳에 미치는 영향에 대해 걱정하지 않고 코드와 시스템을 한곳에서 변경하는 능력

추상화 + 관심사 분리 + 결합도 + 응집성 아이디어와 밀접하게 연결되어있다.

만들어본 모듈성 & 설계 단순 체크리스트

1. 모듈성

- 각 모듈/컴포넌트는 하나의 책임만 수행하는가?
- 내부 구현이 외부에 노출되지 않는가?
- 모듈 간 의존성이 최소화되어 있는가?
- 코드/함수/메서드가 너무 길거나 복잡하지 않은가?

2. 테스트 가능성

- 모듈 단위로 독립적 테스트 가능한가?
- 동일 입력에 대해 결정론적 결과를 제공하는가?
- 테스트를 위해 불필요하게 전체 시스템을 동작시킬 필요가 없는가?

3. 경계 & 인터페이스

- 서비스/모듈 경계가 명확하게 정의되어 있는가?
- 외부 입력/출력은 내부 모델로 변환 지점을 거치는가?
- 외부 시스템과 내부 모듈 간 어댑터가 있는가?
- 입력/출력의 형식이 계약(Interface)으로 정의되어 있는가?

4. 배포 & 독립성

- 모듈/서비스가 독립적으로 배포 가능한가?
- 배포 파이프라인에서 자동화된 빌드, 테스트, 배포가 가능한가?
- 팀이 병렬로 개발할 수 있도록 모듈 독립성이 보장되는가?

5. 복잡성 관리

- 구조 변경 시 영향 범위가 최소화되는가?
- 관심사 분리가 되어 있어 UI, 비즈니스 로직, 데이터 처리가 구분되는가?
- 테스트 기반 설계를 통해 설계 품질을 검증할 수 있는가?