13장 결합도: 소프트웨어 모듈 간의 상호연관수준

결합도

소프트웨어의 모듈간의 상호 의존성 척도

결합도는 우리가 항상 완전히 제거할 수 있는 대상도 아니고, 제거해야 하는대상도아니다.

너무 느슨해도 너무 긴밀해도 문제

모듈성, 응집성, 추상화, 관심사의 분리

→ 시스템의 결합도를 낮춘다.

이전장에선,

추상화하지않으면 코드가 서로 긴밀하게 연결되는 문제!

시스템의 한부분이 변경되면 다른부분에서도 영향을끼침

긴밀한 결합 < 느슨한 결합

너무 느슨한 결합에도 비용이 들지만 일반적으로 '너무 긴밀한 결합'의 비용보다는 훨씬 더 저렴하다

수직 확장을 위해서는 결합도가 필수

사람 늘리면 더 빨라진다는 착각 (수평확장)

소프트웨어 개발은 인원을 늘린다고 빨라지지 않는다

오히려 팀이 커질수록:

- 조율 비용 증가
- 커뮤니케이션 폭증
- 릴리스 일정 꼬임

이 현상의 근본 원인 = 결합도

→ 해결방법으로는 지속적인 통합(CI) (수직확장)

코드를 바꾸면 자동으로빌드, 자동테스트

즉 결합도의 증상을 제어하는 방법

마이크로서비스: 결합도를 분리하기 위한 효과적인 방법

마이크로서비스 접근방식은

서비스를 빌드하고 다른 서비스와 독립적으로 배포

다른 서비스를 상관하지 않아도 된다.

그렇다면?

- 조직의 규묘를 확장
- 서비스를 서로 분리, 생산팀 분리
- 개발기능을 확장

하지만 비용이 발생한다.

더 잘된 추상화가 필요하며, 격리하고, 다른 서비스와의 상호작용에서 느슨하게 결합해야한다.

느슨한 결합도의 대가: 더 크고 많아진 코드

```
def add_to_cartl(self, item):
    self.cart.add(item)
    conn = sqlite3.connect(,my_db.sqlite,)
    cur = conn.cursor()
    cur.excute('INSERT INTO cart (name, price) values (item.name, item.price)')
    conn.commit()
    conn.closef)
    return self.calculate_cart_total();
```

여기서 장바구니에 넣는 부분을 따로 분리

관심사의 분리, 결합도 줄이기, 응집성 개선 적용된 아래의 코드

```
def add_to_cartl(self, item):
    self.cart.add(item)
    self.store_item(item)
    return self.calculate_cart_total();

def store_item(self, item):
    conn = sqlite3.connect(,my_db.sqlite,)
    cur = conn.cursor()
    cur.excute('INSERT INTO cart (name, price) values (item.name, item.price)')
    conn.commit()
    conn.closef)
```

실제 시스템에서는 입력하는 문자 수를 세는 것이 아니라

신중하게 생각하잘 설계하고,

코드를 통해 명확하게 의사 소통하는 방식

더 적은 코드를 작성해야한다.

결합도 모델은 한 종류만이 아니다

나이가드는 결합도를 함께 변경될 수밖에 없는 이유로 결합도를 분류했다.

유형	한 줄 정의
운영적 결합도	한쪽이 안 떠 있으면 다른 쪽이 아예 실행 불가
개발적 결합도	변경·릴리스를 반드시 같이 조율해야 함
의미론적 결합도	같은 개념(도메인 의미)을 공유해서 함께 바뀜
기능적 결합도	책임이 얽혀 있어서 함께 바뀜
부수적 결합도	딱히 이유는 없는데 같이 바뀜 (설계 실패 신호)

결합도는 **관리 대상**이지, 제거 대상이 아니다

결합도를 인식하는 건 출발점이고,

어떤 결합도를 줄이고 어떤 걸 감수할지 결정하는 게 설계다

어쨌든, 느슨한 결합이 긴밀한 결합보다는 좋다

> 아무 생각 없이 전부 분리하지 말고,
> 빠른 곳은 뭉치고, 바깥은 느슨하게

1. **추상화의 봉합 지점을 그린다**
   → 시스템에 명확한 경계선을 만든다
2. 고성능이 중요한 부분은 그 선의 한쪽에만 둔다
   → 빠르게 돌아야 하는 코드는 경계 안쪽에 몰아둔다
3. 그래서 응집성을 유지한다
   → 관련된 코드가 흩어지지 않고 한 덩어리로 모여 있다
4. 모듈/서비스를 넘나들면 추가 비용이 든다는 걸 인정한다 (느슨한결합, 모듈화)
   → 경계 넘을 때는 느려질 수 있다는 걸 미리 알고 설계한다

느슨한 결합과 관심사 분리의 상관관계

관심사 분리 (Separation of Concerns)

- “역할을 잘 나눴는가?”
- 주문 처리 / 주문 저장처럼
  → *무엇을 하는지* 기준으로 나눔

느슨한 결합 (Loose Coupling)

- “서로 얼마나 덜 의존하는가?”
- 한쪽이 바뀌어도

  → 다른 쪽이 덜 깨지는 구조

👉 비슷해 보이지만 완전히 같은 개념은 아님

그래서 이런 경우가 생긴다

① 관심사 분리는 잘했는데, 결합은 빡센 경우 (꽤 흔함)

예시: 주문 처리 서비스 ↔ 주문 저장 서비스

- 역할은 잘 나눔 → 👍 관심사 분리 O
- 하지만:
  - 주고받는 데이터 구조가 복잡함
  - ‘주문’ 개념이 바뀌면 둘 다 같이 수정해야 함

결과

→ 관심사 분리 O / 느슨한 결합 X (긴밀한 결합)

---

② 느슨하게 연결했는데, 관심사 분리는 망한 경우 (더 위험함)

예시: 계좌 A ↔ 계좌 B 송금

- 서비스 2개
- 메시지로 비동기 통신 (겉보기엔 느슨함)

하지만 실제로는:

- A: 돈 빼기
- B: 돈 넣기
- ‘송금’이라는 하나의 거래가 둘로 쪼개짐

문제

- 중간에 장애 나면?
  - 돈은 빠졌는데 안 들어감 😱
- 하나의 핵심 책임(송금)이 분리되지 않음

결과

→ 느슨한 결합 O / 관심사 분리 X

→ *아주 나쁜 설계*

한 줄로 정리하면

> 느슨한 결합 = 좋은 설계 ❌
>
> 관심사 분리 = 무조건 안전 ❌

좋은 설계란?

- **책임은 하나로 응집**
- **그 책임 덩어리들끼리는 느슨하게 연결**

DRY는 너무 단순하다

DRY = Don’t Repeat Yourself

- 같은 로직, 같은 규칙을 여러 군데에 두지 말자
- 바뀌면 **한 곳만 고치면 되게** 하자

DRY를 넓게 적용하면 생기는 일

- 여러 서비스가 **같은 코드 / 라이브러리**에 의존
- 한쪽이 바뀌면 다른 쪽도 **같이 바꿔야 함**

  이건 **개발적인 결합도**

> “내가 업데이트했는데
>
> 너도 같이 업데이트해야만 배포 가능함”

팀의 **자율성 박살,** 배포 지연, 원치 않는 작업 강제

원칙

- **DRY는 배포 파이프라인 내부까지만**
- **파이프라인 간 DRY는 피하라**

마이크로서비스에서 하면 안 되는 것

- 서비스 간 코드 공유
- 공통 라이브러리 강제 사용

마이크로서비스 간 DRY ❌

마이크로서비스 내부 DRY ⭕

느슨한 결합을 위한 비동기식 구현 방법

같은 프로세스 안에서:

- 함수 호출 → 거의 실패 안 함

하지만

서비스 A → 네트워크 → 서비스 B

이 순간부터:

- 네트워크 문제
- 타임아웃
- 패킷 손실
- 서버 다운
- 응답은 갔는데 결과는 안 옴

실패할 수 있는 지점이 **폭발적으로 늘어남**

그래서 필요한 것:

- 비동기 통신
- 재시도
- 타임아웃
- 서킷 브레이커
- 실패를 전제로 한 설계

네트워크/원격 통신이 갖는 우발적인 복잡성 (끊김, 지연, 실패 등)과

내 서비스가 본질적으로 해야 할 비즈니스 로직 사이에 결합도를 낮추는 장치를 잘 두면 시스템이 정상일 때도, 장애가 있을 때도 같은 코드가 동작하도록 만들 수 있다

▼ 위의예시

문제는 보통 이렇게 섞여 있음 ❌

```
publicvoidplaceOrder(Order order) {
try {
        paymentService.pay(order);// 네트워크 호출
        inventoryService.decrease(order);// 또 네트워크
        saveOrder(order);
    }catch (TimeoutException e) {
// ??? 다시 시도? 실패? 롤백?
    }
}
```

📌 결과

- 비즈니스 코드에
- 네트워크 예외 처리 범벅
- 정상/비정상 코드가 섞임
- 테스트도 어려움

저자가 말하는 “결합도를 줄이는 아이디어”

👉 **네트워크를 직접 다루지 않게 만든다**

핵심 전략

- 비동기
- 메시지
- 이벤트
- 큐
- 아웃박스 패턴

“정상일 때도, 장애일 때도 같은 코드” 예제

✅ 좋은 구조 예시 (개념)

```
publicvoidplaceOrder(Order order) {
OrderPlacedevent=newOrderPlaced(order);
    eventStore.save(event);// 로컬 DB, 확실함
}
```

📌 여기서 끝

→ **네트워크 없음**

이후는 비동기

```
OrderPlaced 이벤트
 ├─ 결제 서비스가 처리
 ├─ 재고 서비스가 처리
 └─ 알림 서비스가 처리
```

- 결제 서비스가 잠깐 죽어도
- 주문 서비스 코드는 변하지 않음
- 나중에 다시 처리 가능

느슨한 결합을 위한 설계

테스트가능한 코드 = 느슨하게 결합된 시스템을 제공

강건한 결합도로 영원히 고통받는 대규모 조직

사람들이 착각하는 것

- 프로그래밍이 어렵다
- 언어, 문법, 기술이 문제다

저자가 말하는 진실

- **몇 줄 코드 쓰는 건 누구나 배운다**
- 진짜 어려움은:
  - 시스템이 커질 때
  - 팀이 많아질 때
  - 변경이 잦아질 때

조정된 접근 (Coordinated)

특징

- 하나의 코드베이스
- 하나의 빌드
- 하나의 테스트
- 하나의 배포

장점

- 일관성 최고
- 문제를 바로 발견

대가

- 빠른 CI/CD에 엄청난 투자 필요
- 하루에도 여러 번 피드백 받아야 함

👉 **결합이 높을수록 → 피드백을 더 자주 받아야 함**

분산 접근 (Distributed = 마이크로서비스)

특징

- 팀 독립
- 서비스 독립
- 배포 독립
- 조정 최소화

장점

- 조직 확장성 최고
- 팀 수를 계속 늘릴 수 있음

대가

- 설계 난이도 급상승
- 강제 규칙 없음
- 느슨한 제어를 받아들여야 함

👉 **조정 비용을 포기하고, 설계 비용을 지불**

어느쪽이든 비용이 발생

정리

전략 1: 결합은 높게, 대신 피드백을 극도로 빠르게

방법

- 지속적 통합 (CI)
- 지속적 배포 (CD)
- 큰 단일 시스템

의미

- 깨지면 바로 알게 함
- 하루에도 여러 번 확인

대가

- 엄청난 자동화 투자
- 인프라/테스트 비용 큼

👉 “결합을 감당할 체력”이 필요

전략 2: 결합 자체를 줄인다 (분리)

방법

- 서비스 분리
- 독립 배포
- 비동기 통신
- 변경 격리

의미

- 남에게 영향 안 주고 변경
- 팀 자율성 확보

대가

- 설계 난이도 증가
- 운영 복잡성 증가

👉 “설계로 비용을 치른다”

전략 3: 인터페이스를 얼린다

방법

- 합의된 계약(API, 스키마)

- 절대 깨지지 않음
- 버전 관리

의미

- 변경하지 않기로 약속

대가

- 미래 유연성 감소
- 초기 설계 실패 시 치명적

👉 “변경 비용을 미래로 넘김”

즉

결합도는 제거 대상이 아니라

‘어디에 둘지 선택해야 하는 대상’이다.