12장 정보 은닉과 추상화: 우리의 적인가 친구인가

정보 은닉과 추상화

세부사항을 제거해 더 중요한 세부사항에 주의를 집중하는 과정

정보 은닉과 추상화는 한몸이다

그뒤에 뭐가있는지 신경 쓰지 않도록 하는것, 사용법만 알면된다.

'큰 진흙탕'이 된 코드의 원인을 찾아서

코드 기반은 너무 복잡하고 얽혀 있기에 변경을 두려워하는 경우가 많다.

조직적이고 문화적인 문제

코드 작성을 위해 나를 고용했다면, 내가 할 수 있는 한 최선을 다하는것

소프트웨어 개발의 전문적인 접근 방식의 기초

리팩토링, 테스트, 좋은 설계를위한 시간, 버그 발견시 수정, 협업, 의사 소통, 학습에 소용되는 비용

→ 이걸 기반으로 고품질의 결과물을 만들고, 설계아이디어를 제공하는것

기술적인 문제와 설계의 문제

> 설계를 동결하는 순간 구식이 된다.

우리가 아이디어, 팀, 코드 또는 기술을 더 많이 배우면서 변화할 수 있는 능력을 유지해야한다는 철학을 받아들이면

1. 실수하고 바로 잡기
2. 직면한 문제에 이해를 깊게하고 새로운 이해를 설계에 반영
3. 제품과 기술을 점진적으로 발전

→ 이것이 좋은 소프트웨어 공학을 위한 목표이다.

몇년후에 방문해도 이해할 수 있는 공간

→ 추상화와 정보 은닉

추상화 수준 높이기 (추가, 요약)

- **추상화 수준을 높이는 아이디어는 유망하지만**, 현실에서 복잡한 시스템을 구현하려면 여전히 **세부적, 텍스트 기반의 프로그래밍이 필요**.
- 시각적 언어와 다이어그램은 보조적 도구로는 유용하지만, 완전히 대체하기에는 한계가 있음.
- 텍스트 기반 언어는 **아이디어를 간결하고 유연하게 인코딩**할 수 있다는 점에서 강점이 있음.

과도하게 공들인 공학의 우려

기술적 유혹과 실용주의

기술 전문가로서 우리는 '반짝이는 기술 아이디어'를 쫓는 경향이 있음.

- 문제: 새로운 기술이나 멋진 접근법을 무조건 따라가면 실제 문제 해결보다 기술 자체에 몰입하게 됨.
- 해결책: 항상 문제를 해결하는 맥락에서 기술을 평가하고, 시험이나 프로토타입 정도로 탐색.

  → 멋진 기술을 전체 개발에 적용해 위험을 키우지 말 것.

단순함과 YAGNI 원칙

단순함을 추구하면 오히려 좋은 결과와 이력서·경력 가치도 높아짐.

미래 대비 설계("나중에 필요할 것")는 오히려 미성숙한 설계의 신호.

- 켄트 벡의 YAGNI(You Ain't Gonna Need It!) 원칙:

  → 지금 필요한 코드만 작성하라.

소프트웨어는 매우 유연하지만 취약함. 과도하게 미래를 대비하려다 보면 불필요한 복잡성을 만들고 코드 변경이 어려워짐.

코드 변경의 두려움과 해결책

많은 조직에서 '영웅적인 프로그래머'에게 의존해 어려운 변경을 처리하게 됨.

진정한 해결책은 영웅 모델이 아님.

- 대신 필요한 접근
  1. 추상

     → 시스템의 복잡성을 은닉하여 한 부분을 안전하게 변경 가능.
  2. 테스트

     → 코드 변경 시 다른 부분에 영향을 주지 않도록 확인.

추상화를 높이려면 테스트 코드부터 작성하라

테스트 케이스를 멋지게 작성할 수 있도록 아이디어를 쉽게 표현할 수 있는 코드 인터페이스로 정의한다.

즉, 코드도 사용하기 쉽다.

이모든 설계행위는 코드의 구현 세부 사항에 도달하기 전에 이뤄진다.

→ 추상화에 기반한 이런 접근방식, 설계에 가벼운 접근 방식

좋은 추상화가 핵심이다.

추상화의 가치

우리는 소비자로서 추상화의 힘을 이미 경험함

- 운영체제, API, 클라우드 서비스(AWS S3 등) 등은 하드웨어나 복잡한 시스템을 감춘 추상화 덕분에 단순하게 사용 가능.

추상화를 통해 개발자는 복잡성을 숨기고 안전하게 시스템을 활용할 수 있음.

소프트웨어 생산자와 추상화

많은 개발자가 자신의 코드에서 추상화에 충분히 주의를 기울이지 않음.

초기 운영체제는 하드웨어 추상화가 제한적이었지만, 현대 시스템은 전체 스택에서 추상화를 제공.

좋은 추상화는 단순함과 안전성을 동시에 제공:

- 예: PC 비디오 카드 변경 시 앱이 계속 작동
- 예: AWS S3 API – 복잡한 분산 저장 시스템을 단순한 API로 추상화

데이터 구조와 추상화

HTML, XML, JSON 등은 의미론적 구조를 명시적으로 제공하며 데이터 통신과 이해를 단순화.

1. 평문 텍스트도 하나의 추상화
    - 단순한 문자 스트림 외에는 정보 구조에 크게 신경 쓰지 않고 데이터 다루기 가능.
    - 심지어 평문 텍스트도 사람이 설계하고 합의한 규칙 기반 추상화임.
2. 파일 시스템과 유닉스의 파이프(piping) 역시 강력한 추상화로서, 모듈 간 로직 연결을 단순화.

즉 추상화는 소프트웨어 개발의 핵심 도구이자 인간의 컴퓨터 이해 능력의 기반

핵심은 좋은 추상화를 만드는것

구멍 난 추상화

구분	예시	누수 포인트
하드웨어 제약	고속 서버 + 가비지 컬렉션	메모리 접근 속도가 예상과 다름
기술 세부사항 노출	주문 모듈 + HTML 오류 코드	비즈니스 추상화가 기술적 세부사항에 침범
데이터 전송	JSON 통신	데이터 크기/형식 때문에 내부 제약 드러남

추상화의 누수란, 추상화가 숨기려고 한 세부 사항이 실제 사용 환경에서 드러나는 현상.완전히 피할 수 없지만, 누수를 예상하고 설계 하면 관리 가능.

세계 지도와 지하철 노선도의 비유로 배우는 추상화 기법

지도(map) 예시로 보는 추상화

지도 종류	목적	특징	누수/한계
메르카토르 차트 (항해용)	배/비행기로 두 지점 사이 항해	방위 일정하게 유지, 경로 계획 가능	구면 표면의 최단거리와 다름 → 거리 정보가 일부 왜곡됨
런던 지하철 노선도 (해리 벡)	지하철 이동 경로 확인	역 간 연결 관계 명확, 실제 거리 무시	지리적 정확성 없음 → 걸어서 이동할 때 부적합

같은 '지도'라는 추상화지만 목적에 따라 선택이 달라야 한다.

중요한 것은 문제를 해결하는 데 유용한 모델을 만드는 것이지, 현실을 완벽히 반영하는 것이 아님.

소프트웨어 추상화에 적용

코드에서의 추상화도 마찬가지:

- 구현 방식이나 내부 구조를 완벽히 숨길 필요는 없음.
- 중요한 것은 문제를 해결하고 변경 가능성을 유지하며 테스트할 수 있는 유용한 모델을 만드는 것.

TDD(Test-Driven Development) 예시:

- 테스트를 먼저 작성하면, 문제가 무엇인지 추상화한 명세가 만들어진다.
- 테스트가 변하면 추상화도 변한다 → 추상화를 개선할 수 있는 피드백 제공.

효율한 소프트웨어 개발이 되는 지침 몇가지

이벤트 스토밍으로 추상화를 달성하자

문제 도메인의 관심사를 자연스럽게 분리하는 기법

이벤트 스토밍

추상화된 우발적인 복잡성

본질적 복잡성과 우발정 복잡성을 구분하고, 추상화를 통해 복잡성을 관리해야한다.

복잡성의 두 종류

- 본질적 복잡성(Inherent Complexity)

  문제 도메인 자체가 갖는 복잡성.

  예: 장바구니에 항목을 추가하고 합계를 계산하는 비즈니스 로직
- 우발적 복잡성(Accidental Complexity)

  구현 환경, 기술 스택 등에서 발생하는 복잡성.

  예: RAM과 디스크의 차이, 데이터베이스 연결 실패, 네트워크 장애

> 좋은 설계는 이 두 복잡성을 최대한 분리하고, 우발적 복잡성이 본질적 복잡성을 침범하지 않도록 하는 것

추상화와 관심사 분리

글에서 제시된 **add_to_cart** 예제를 통해 이해할 수 있음:

1. add_to_cart1

```
defadd_to_cart1(self, item):
self.cart.add(item)
    conn = sqlite3.connect('my_db.sqlite')
    cur = conn.cursor()
    cur.execute('INSERT INTO cart (name, price) values (item.name, item.price)')
    conn.commit()
    conn.close()
returnself.calculate_cart_total()
```

- **문제점**
  - 비즈니스 로직(장바구니 합계 계산)과 저장소 처리(DB 접근)가 뒤섞임
  - 우발적 복잡성이 본질적 복잡성을 침범
  - 응집성 낮음, 테스트 어려움

2. add_to_cart2

```
defadd_to_cart2(self, item):
self.cart.add(item) // 본질적 복잡성
self.store.store_item(item)  // 우발적 복잡성
return self.calculate_cart_total()
```

- **개선점**
  - 저장소(Store)라는 추상화 도입
  - 우발적 복잡성을 분리하여 코드 응집성 향상

3. add_to_cart3

```
defadd_to_cart3(self, item, listener):
self.cart.add(item) // 본진적 복잡성
    listener.on_item_added(self, item) // 우발적 복잡성 위임
```

- **개선점**
  - 추상화 유지 + 이벤트 리스너를 통한 확장성 제공
  - 저장소 구현 세부 사항 노출 최소화
  - 테스트 용이성 증가
- **의미**: 본질적 복잡성을 유지하면서, 우발적 복잡성에 대한 영향을 최소화

추상화 누수(Leakage) 관리

- 추상화에도 누수 가능성이 있음

  예: 저장 시도 실패, 네트워크 문제

- 코드 12.2: 추상화 누수 최소화 방법

```
defadd_to_cart2(self, item):
if self.store.store_item(item):
self.cart.add(item)
return self.calculate_cart_total()
```

- 저장 성공 여부를 **불린값**으로 제한해 도메인 추상화를 오염시키지 않음
- 실패 처리는 내부에서 관리

즉 추상화는 우발적 복잡성을 감추고 본질적 복잡성에 집중하도록 돕는다.

타사 시스템과 타사 코드를 격리하자

버전	타사 코드 결합	추상화 수준	장점
1	강하게 결합 (sqlite3)	낮음	없음, 테스트 어려움
2	추상화를 통해 간접 접근	중간	저장소 구현 자유, 테스트 용이
3	이벤트 리스너로 위임	높음	확장성, 관심사 분리 최상, 테스트 용이

- **타사 라이브러리에 직접 결합**하면 코드 유연성이 떨어지고, 우발적 복잡성이 본질적 복잡성을 침범
- 자체 추상화를 통해 타사 코드와 본인의 코드 사이를 단열판처럼 분리

결과

코드가 더 유연해지고, 다양한 저장소 구현이나 이벤트 처리로 확장 가능하며, 테스트가 쉬워짐

추상화와 구상화 사이의 트레이드오프

```
public ArrayList<String> doSomething1(HashMap<String,String> map); // 너무 구체적
public List<String> doSomething2(Map<String,String> map);       // 적절한 추상화
public Object doSomething3(Object map);                          // 지나치게 일반적
```

내가 거의 확실하게 관심이 있는 사안은 ArrayLtst가 아닌 List의 특성

구분	doSomething1	doSomething2
매개변수	HashMap (구체적)	Map (인터페이스)

구분	doSomething1	doSomething2
반환 타입	ArrayList (구체적)	List (인터페이스)
유연성	낮음	높음
테스트 용이성	낮음	높음
정보 은닉/추상화	부족	충분

추상화를 위한 최적의 지점을 파악하는 능력은 테스트 가능성을 고려한 설계를 통해 향상된다.

정리

추상화는 소프트웨어의 핵심

코드의 성격이 무엇이든 정보를 은닉하는 봉합지점을 만드면 코드가 더 나아진다.