11장. 관심사 분리: 고품질 코드의 가장 중요한 속성

관심사 분리

- 컴퓨터 프로그램을 별개 구획으로 분리해 각 구획이 별도의 관심사를 다루도록 하는 설계 원칙
- 모듈성과 응집성에 대한 구체적인 표현
- 결합도를 줄이고 코드와 시스템의 응집성과 모듈성을 개선하는 데 도움을 주는 채택 가능한 기술

▼ 코드 10.5 세 가지 응집성 예시

```
def add_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()

    return self.calculate_cart_total();

def add_to_cart2(self, item):
    self.cart.add(item)
    self.store.store_item(item)
    return self.calculate_cart_total();

def add_to_cart3(self, item, listener):
    self.cart.add(item)
    listener.on_item_added(self, item)
```

저자는 add_to_cart3 를 선호함

- 저장이라는 개념을 핵심 비즈니스에서 제거한 측면이 맘에 든다고 함

add_to_cart3 의 주된 반대 내용

- 명확성이 떨어진다.
- 명확성이 코드의 미덕이라는 점에선 저자도 동일하지만 이는 컨텍스트의 문제일 뿐이라 함

관심사 분리를 기본 원칙으로 상당히 중요하게 생각하는 이유

- 집중력을 좁게 유지하도록 상기시켜 주기 때문

의존성 주입

- 관심사 분리를 위한 매우 유용한 도구는 의존성 주입(DI) 이다.
- 의존성 주입은 도구나 프레임워크의 기능이 아닌 대다수 언어, 특히 객체 지향 또는 함수형 언어에서 기본적으로 수행할 수 있는 작업이며 강력한 설계 접근 방식 이다.

▼ 코드 11.2 우발적인 복잡성과 본질적인 복잡성을 분리하기

```
public interface Accidental
{
    String[] readWords() throws IOException
    boolean storeWords(List<String> sorted) throws IOException
}

public class Essential
{
    public boolean loadProcessAndStore(Accidental accidental) throws IOException
    {
        List<String> sorted = sortWords(accidental.readWords());
        return accidental.storeWords(sorted);
    }

    private List<String> sortWords(String[] words)
    {
        List<String> sorted = Arrays.asList(words);
        sorted.sort(null);
        return sorted;
    }
}
```

▼ 코드 11.3 추상화를 통해 우발적인 복잡성을 제거하기

```
public interface WordSource
{
    String[] words();
}

public interface WordsListener
{
    void onWordsChanged(List<String> sorted);
}

public class WordSorter
{
    public void sortWords(WordSource words, WordsListener listener)
    {
```



```
        listener.onWordsChanged(sort(words.words()));
    }

    private List<String> sort(String[] words)
    {
        List<String> sorted = Arrays.asList(words);
        sorted.sort(null);
        return sorted;
    }
}
```

경계 컨텍스트를 활용한 하향식 관심사 분리

- 우리는 문제 영역의 관점에서 설계를 유도하는 방법을 모색할 수도 있다.

- 상대방의 함대를 침몰시키는 어린이용 게임인 <배틀쉽> 예제

▼ 코드 11.4 개념이 누락됨

```
class GameSheet:

    def __init__(self):
        self.sheet = {}
        self.width = MAX_COLUMNS
        self.height = MAX_ROWS
        self.ships = {}
        self._init_sheet()

    def add_ship(self, ship):
        self._assert_can_add_ship(ship)
        ship.orientation.place_ship(self, ship)
        self._ship_added(ship)
```

- GameSheet는 배의 위치 '그리고' 게임 규칙을 담당하고 있다.
- 클래스나 메서드 설명에 포함된 '그리고' 라는 문구는 경고 신호다.
- 위 내용의 경우 '규칙' 이라는 개념을 놓치고 있다는 사실이 명백히 보임

▼ 코드 11.5 코드에 귀를 기울이기

```
class GameSheet:

    def __init__(self, rules):
        self.sheet = {}
        self.width = MAX_COLUMNS
        self.height = MAX_ROWS
        self.rules = rules
        self._init_sheet()

    def add_ship(self, ship):
        self.rules.assert_can_add_ship(ship)
        ship.orientation.place_ship(self, ship)
        self._ship_added(ship)
```

- GameSheet 는 Ships 컬렉션을 유지할 필요가 없어졌고, 규칙 준수 여부를 검증하는 데 초점을 맞춘 코드 진화의 시작에 불과했던 9~10 줄짜리 유효성 검사 로직이 제거되었다.

테스트하기 쉬운 코드 = 관심사가 분리된 코드

- 테스트하기 쉬운 코드를 만들기 위해서는 관심사를 분리해야만 하며 그렇지 않으면 테스트는 집중력을 잃을 것이다.

육각형 아키텍처: 포트와 어댑터

▼ 코드 11.6 S3에 문자열을 저장하기

```
void doSomething(Thing thing) {
    String processedThing = process(thing);
    s3client.putObject("myBucket," "keyForMyThing," processedThing);
}
```

- 첫 번째 줄은 함수나 메서드의 세계에서 의미 있는 일을 하는데 초점을 맞추고 있다.
  - (무언가를) 처리하는 부분이 비즈니스라는 컨텍스트에서 의미가 있을 수 있지만 이것이 이코드의 초점, 본질적인 부분이라는 점을 제외하고 나면 실제로는 중요하지 않다.
- 두번째 줄은 외계인이다.
  - 우리 로직 핵심에 우발적인 복잡성을 던져 넣은 침입자.

▼ 코드 11.7 포트를 통해 S3에 문자열을 저장하기

```
void doSomething(Thing thing) {
    String processedThing = process(thing);
    store.storeThings("myBucket," "keyForMyThing," processedThing);
}
```

- 위 코드와 다른 점은 '저장소로 호출'의 내용이 이 함수의 다른 아이디어와 더 일관되게 만들면서 추상화 수준을 높였다.
- 새로운 추상화는 '포트' 또는 정보가 흐르는 벡터로 생각할 수 있다.
- 이 포트의 구체적인 구현은 '어댑터'로 위 예제에서는 'AWS S3 저장소' 라는 컨텍스트가 어댑터가 된다.
- 위 변경 이후 우리의 코드는 S3에 대해 아무것도 알지 못하며, 심지어 S3가 사용되고 있다는 사실조차 알지 못한다.
- 코드는 더 일관성 있는 추상화를 유지하게 된다.

→ "포트와 어댑터 패턴", 핵사고날 아키텍처

포트와 어댑터는 언제 채택하면 좋을까

- 항상 경계 컨텍스트 사이에서 교차하는 정보를 번역하자.
  - 포트와 어댑터를 사용해 서비스 사이에서 통신하라는 말

API 가 단순한 함수 호출이 아닌 이유

- API 는 해당 API 를 외부에 공개하는 서비스 또는 라이브러리의 소비자에게 노출되는 모든 정보를 말한다.
- 개발자와 나누는 비공식적 대화에서 'API' 라는 용어가 'HTTP 를 통한 텍스트' 의 동의어로 사용되는 경우가 일반적이다.
- 엄밀히 말해, 어떤 종류의 프로그래밍을 지원하기 위한 다양한 코드 비트 사이의 통신 수단은 모두 API 다.

정리

- 관심사 분리는 확실히 고품질 코드의 속성이다.
- 관심사 분리는 더 나은 소프트웨어를 설계하는 방향으로 우리를 이끄는 환상적인 도구다.

관심사 분리의 목표와 역할:

- 관심사 분리는 코드와 시스템의 명확성과 집중력에 중점을 둡니다,.

- 이 원칙은 시스템의 **모듈성, 응집성, 추상화**를 개선하고, 결과적으로 **결합도**를 효과적으로 최소화하는 핵심 기술입니다,.
- 관심사 분리를 통해 작성된 코드는 깔끔하고, 구성 가능하며, 유연하고, 효율적이며, 확장 가능하고, 변화에 개방적인 상태를 유지할 수 있습니다,.

복잡성 분리:

관심사 분리를 위한 효과적인 접근 방식은 시스템의 복잡성을 두 가지 유형으로 명확하게 나누는 것입니다:

1. **본질적인 복잡성 (Essential Complexity):** 해결하려는 문제 자체에 내재된 복잡성입니다 (예: 장바구니에 품목을 추가하고 합계를 계산하는 방법). 이는 시스템이 제공하는 진정한 가치와 관련됩니다.

2. **우발적인 복잡성 (Accidental Complexity):** 컴퓨터에서 유용한 작업을 수행하기 위해 부수적으로 해결해야 하는 모든 문제(예: 데이터 영속성, 화면 표시, 보안 등)를 말합니다.

설계자는 본질적인 복잡성을 다루는 코드와 우발적인 복잡성을 다루는 코드 사이에 명확한 경계를 설정하여, 핵심 비즈니스 영역을 우발적인 복잡성으로부터 분리해야 합니다,. 이렇게 하면 코드의 가독성, 테스트 가능성, 유연성, 유용성이 개선됩니다,.

관심사 분리를 위한 도구:

- **의존성 주입 (Dependency Injection):** 코드가 의존성을 자체적으로 생성하는 대신 매개변수로 제공받도록 하여 유연성을 개선하고 결합도를 최소화합니다,,,.
- **테스트 가능성 및 TDD:** 코드를 테스트하기 쉽게 만들려고 노력하면 관심사를 분리해야만 하는 압력을 받게 됩니다,,. 테스트를 통해 개발을 추진하면 (TDD), 설계 결정의 비용과 이점에 일찍 눈을 뜨게 되며, 설계가 원하는 추상화를 따르도록 압력을 가합니다.
- **포트와 어댑터 (Ports & Adapters):** 모듈(서비스) 간의 경계(봉합 지점)에 번역 계층을 두어, 한쪽의 구현 세부 사항이 다른 쪽으로 누수되는 것을 방지합니다,,. 예를 들어, 외부 서비스(AWS S3 등)와의 상호 작용을 추상화하여, 해당 기술이 변경되어도 우리 코드의 다른 부분은 영향을 받지 않도록 합니다,.

관심사 분리를 기본 원칙으로 적용하면, 개발자는 집중력을 좁게 유지할 수 있으며, 이로 인해 시스템이 더 모듈화되고 응집성이 높아지며 결합도가 낮아지게 됩니다,.