11장 관심사 분리: 고품질 코드의 가장 중요한 속성

관심사분리

“내 작업에서 가장 강력한 설계 원칙”

> 클래스 하나에 작업 하나, 메서드 하나에 작업 하나

→ 추상화를 개선, 결합도 효과적으로 줄임

```
def add_to_cartl(self, item):
    self.cart.add(item) // 장바구니 추가

    conn = sqlite3.connect('my_db.sqlite') // DB 저장
    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) // 장바구니 추가
    self.store.store item(item) // 저장소
    return self.calculate_cart_total();

def add_to_cart3(self, item, listener):
    self.cart.add(item) // 장바구니 추가
    listener.on_ttem_added(self, item) // 무언가 추가됨
```

두번째, 저장하는 위치는 장바구니 동작과 관련이 없다.

세번째, 장바구니 추가라는 책임만 → 모듈성 + 응집성

의존성 주입

> 의존성 주입은 코드 생성 대신 매개변수로 제공

버전1 : 코드가 특정구현에 깊숙히 결합

버전2 : 생성자 매개변수로 전달, 유연성개선

버전3 : store_item을 구현하는 모든것

코드가 시스템의 다른 컴포넌트와 협력자 역활

작동방식 몰라, 신경 쓰지 않아

본질적인 복잡성과 우발적인 복잡성을 분리하자

본질적인 복잡성 - 잔고계산, 장바구니 합산 등 복잡성 해결

우발적인 복잡성 - 데이터영속성, 화면표시

이 두가지 관심사 분리를 통해 설계를 개선

아래의 3가지 변형단계

```
public class ReallyBadCohesion
{
   p나bite boolean loadProcessAndStore() throws lOException
   {
      String[] words;
      List<String> sorted;

      try (FileReader reader =
         new FileReader("./resources/words.txt”))

         {
            char[] chars = new char[1024];
            reader.read(chars);
            words = new Strtng(chars).split(" |\0”);
         }
         sorted = Arrays.asListfwords);
         sorted.sort(null);

         try (FileWrtter writer =
            new FtleWriter("./resources/test/sorted.txt"))
         {
               for (String word : sorted)
               {
               writer.write(word);
               writer.write("\n");
               }
               return true;
            }
         }
   }
```

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

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

→ 관심사의 분리를 통해 구조를 개선

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

```
    }
}
```

DDD는 중요하다

- 경계 컨텍스트를 활용한 하양식 관심사 분리

```
class GameSheet:

    def _init_(self, rules):
        self.sheet = {}
        self.width = MAX_COLUMNS
        self.height = MAX_ROWS
        self.rules = rules // 추가된 클래스
        self._tnit_sheet()

    def add_ship(self, ship)：
    self.rules.as se rt_can_add_s h ip(ship)
    ship.orientation.place_shtp(self, ship)
    self._ship_added(ship)
```

배를 배치할 수 있는지 테스트

테스트중 게임 규칙을 빠트림

Rules라는 새로운 클래스 추출

규칙이 무엇이 될지 생각도 안하게됨, 더 잘 테스트하고 설계를 개선

테스트하기 쉬운 코드 = 관심사가 분리된 코드

테스트하기 쉬운코드

“모듈성, 응집성, 관심사 분리, 정보 은닉, 느슨한 결합” → 관심사의 분리가 된코드이다.

고품질 속성을 보여주는 시스템

육각형 아키텍처: 포트와 어뎁터

가치가 있는 수준은 코드

하나의 ‘관심사’가 다른 ‘관심사’와 상호작용하는 봉합지점

간단히 예를 살펴보자

```
void doSomething(Thing thing)
{
   String processedThing = process(thing);
   s3cItent.putObject("myBucket," "keyForMyThing," processedThing);
}

void doSomething(Thing thing)
{
   String processedThing = process(thing);
   store.storeThings("myBucket," "keyForMyThtng," processedThing);
} // 저장소의 호출을 더일관되게 만듬, 추상화 수준 높임
// 단, 새로운 클라이언트 라이브러리를 활용하기 위해 수십,수백,수천 줄 코드를 변경
```

포트: 정보가 흐르는 추상 인터페이스

어댑터: 도메인 개념을 외부 기술 컨텍스트로 번역

즉 s3용 포트와 어댑터를 만들었다면 코드의 여러곳에서 사용

포트와 어댑터는 언제 채택해야 할까

> 항상 경계 컨텍스트 사이에서 교차하는 정보를 번역

항상 서비스 사이에서 흐르는 정보를 번역하라

시스템을 보호하는 방법

1. 어댑터에서 외부 입력을 검증하거나 변환해서, 우리 시스템이 필요로 하는 형식으로 바꾼다.
2. 외부에서 온 신뢰할 수 없는 데이터를 그대로 도메인에 넘기지 않고 '감싼다(wrap)'

API가 단순한 함수 호출이 아닌 이유

API는 단순히 HTTP + JSON이 아니다.

데이터 구조가 곧 API

외부 데이터 형식이 바뀜,

내부 로직이 깨짐,

수정 비용이 커짐

> 포트와 어댑터가 필요하다

- API가 바뀌면

  → 어댑터 파일 딱 1개만 수정
- 도메인 로직 수백·수천 줄은 손대지 않음
- 테스트는 FakeFileStorage로 가능
- 로컬/테스트/S3/GCP 전환이 즉시 가능
- API 설계 변화에 굉장히 강한 구조

▼ 코드예시

✅ 1. 포트(인터페이스)를 먼저 만든다 — 도메인이 의존할 유일한 것

외부 API가 어떻게 생겼든, 도메인이 바라보는 형태를 **내가 정의**한다.

이게 바뀌지 않으면 도메인 코드는 절대 수정되지 않음.

```
// 포트: 도메인에서 바라보는 추상화
public interface FileStorage {
    void upload(String bucket, String key, byte[] data);
}
```

✔️ 이 인터페이스는 *내가 원하는 형태*로 정의한다.

✔️ 외부 API가 지금 어떻게 생겼는지는 신경 쓰지 않는다.

✅ 2. 어댑터를 만든다 — 외부 API를 포트 형태로 변환하는 곳

예를 들어 AWS S3 SDK v1을 처음 사용한다고 하자.

```
public class S3FileStorageAdapter implements FileStorage {

    private final AmazonS3 s3;

    public S3FileStorageAdapter(AmazonS3 s3) {
        this.s3 = s3;
    }

    @Override
    public void upload(String bucket, String key, byte[] data) {
        ObjectMetadata metadata = new ObjectMetadata();
        metadata.setContentLength(data.length);

        s3.putObject(bucket, key, new ByteArrayInputStream(data), metadata);
```

```
    }
}
```

✔️ 이 클래스 하나만 외부 S3 SDK에 의존한다.

✔️ 나중에 SDK가 바뀔 경우 여기만 수정하면 된다.

✅ 3. 도메인 로직에서는 절대 S3를 모르도록 한다

도메인 서비스:

```
public class ReportService {

    private final FileStorage storage;

    public ReportService(FileStorage storage) {
        this.storage = storage;
    }

    public void publishReport(Report report) {
        byte[] data = report.export();
        storage.upload("reports", report.id(), data);
    }
}
```

➡️ 여기서는 S3, GCP, 로컬, MinIO 같은 것이 무엇인지 전혀 모른다.

➡️ FileStorage 인터페이스만 알고 있다.

따라서 외부 API가 바뀌어도 이 코드는 그대로 유지된다.

✅ 4. API가 변경되었다! → 어댑터만 수정

예를 들어 AWS S3 SDK가 v2로 바뀌었고, 사용 방식이 완전히 달라졌다고 가정하자:

```
public class S3FileStorageAdapterV2 implements FileStorage {

    private final S3Client client;

    public S3FileStorageAdapterV2(S3Client client) {
        this.client = client;
    }

    @Override
    public void upload(String bucket, String key, byte[] data) {
        client.putObject(
            PutObjectRequest.builder()
```

```
                .bucket(bucket)
                .key(key)
                .build(),
            RequestBody.fromBytes(data)
        );
    }
}
```

중요한 점:

- **FileStorage 인터페이스는 그대로다**
- **ReportService 같은 도메인 코드도 하나도 안 고친다**
- **변경된 외부 API는 어댑터에서만 반영한다**

✅ 5. DI로 포트–어댑터를 연결

예:

```
FileStorage storage = new S3FileStorageAdapterV2(new S3Client());
ReportService reportService = new ReportService(storage);
```

Spring이라면:

```
@Configuration
public class AppConfig {

    @Bean
    FileStorage fileStorage(S3Client client) {
        return new S3FileStorageAdapterV2(client);
    }

    @Bean
    ReportService reportService(FileStorage storage) {
        return new ReportService(storage);
    }
}
```

TDD를 이용한 상향식 관심사 분리

모든 개발을 테스트라는 관점에서 추친

테스트 중심으로 개발을 구조화 한다면?

결정의 비용과 이점에 눈을 뜨게됨

정리

관심사 분리는 고품질 코드의 속성이다.

모듈, 클래스, 함수가 하나의 작업을 한다면 더나은 소프트웨어 설계방향으로 이끌어주는 도구가 될것이다.