

# 10장 응집성: 소프트웨어의 관련요소들은 한곳에

응집성(cohesion) : 모듈 내부의 요소들이 함께 속해있는 정도이다.

복잡성을 관리하는 데 필수적인 설계의 기초 원칙이며, 모듈 내부의 요소들이 함께 속해 있는 정도로 정의된다.

## 1. 응집성의 기본 철학: 설계의 기초

좋은 소프트웨어 설계는 코드를 구조화하는 방식에 달려 있으며, 이는 \*\*"서로 관련이 없는 것은 더 멀리 떨어뜨리고, 서로 관련된 것은 더 가깝게 배치"\*\*하라는 켄트 벡(Kent Beck)의 말에 담겨 있다.

### 응집성의 중요성

- 복잡성 및 가독성 관리:** 응집성이 높을수록 코드를 읽고 이해하기 쉬워지며, 변경할 때 변경의 범위나 비용을 최소화할 수 있다.
- 유연성 저하 방지:** 응집성이 떨어지는 경우, 코드와 시스템의 유연성이 저해되고, 테스트하거나 작업하기가 훨씬 더 어려워지는 문제가 발생한다.
- 응집성 부족 감지:** 코드를 읽을 때 "이 코드가 무슨 일을 하는지 모르겠다" 고 생각한 적이 있다면, 이는 응집성이 좋지 않기 때문일 가능성이 높다.

## 2. 응집성 개선과 복잡성 분리

응집성을 개선하는 과정은 시스템의 \*\*본질적인 복잡성 (essential complexity)\*\*과 \*\*우발적인 복잡성 (accidental complexity)\*\*을 명확하게 분리하는 것과 밀접하게 관련된다.

- 본질적 복잡성:** 해결하려는 문제 자체에 내재된 복잡성이다 (예: 장바구니에 품목을 추가하고 합계를 계산하는 방법).
- 우발적 복잡성:** 컴퓨터로 유용한 작업을 수행하기 위해 부수적으로 해결해야 하는 모든 문제다 (예: 데이터 영속성, 네트워크 통신 등).

응집성이 높은 코드는 본질적인 복잡성에 집중하고 우발적인 복잡성과는 분리된 추상화 수준을 유지하려 노력해야 한다. 응집성이 부족한 코드는 종종 이 두 가지 복잡성이 완전히 혼합되어 있다.

## 2-2 예시

### ▼ 코드 10.1 순진한 응집성을 보여주는 정말 나쁜 코드

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

        try (FileReader reader =
                new FileReader("./resources/words.txt"))
        {
            char[] chars = new char[1024];
            reader.read(chars);
            words = new String(chars).split(" |\t");
        }
        sorted = Arrays.asList(words);
        sorted.sort(null);

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

하나의 메서드에 많은 기능들이 접목되어 있다.

- **파일 읽기** - words.txt 파일을 읽어옴
- **데이터 변환** - 문자열을 배열로 분리
- **데이터 정렬** - 배열을 리스트로 변환하고 정렬
- **파일 쓰기** - 정렬된 결과를 새 파일에 저장

## 개선1

```
public class BadCohesion
{
    public boolean loadProcessAndStore() throws IOException
    {
        String[] words = readWords();
        List<String> sorted = sortWords(words);
        return storeWords(sorted);
    }

    private String[] readWords() throws IOException
    {
        try (FileReader reader =
            new FileReader("./resources/words.txt"))
        {
            char[] chars = new char[1024];
            reader.read(chars);
            return new String(chars).split(" \\0");
        }
    }

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

    private boolean storeWords(List<String> sorted) throws IOException
    {
        try (FileWriter writer =
            new FileWriter("./resources/test/sorted.txt"))
        {
            for (String word : sorted)
            {
                writer.write(word + "\n");
            }
        }
    }
}
```

```

        writer.write(word);
        writer.write("\n");
    }
    return true;
}
}
}

```

- 메서드를 `readWords()`, `sortWords()`, `storeWords()`로 분리
- 각 메서드가 하나의 작업에 집중하게된다.

## 그러나 개선되었지만 여전히 개선이 필요하다.

- 관심사 많고, 모듈화 안되어있고, 하드코딩된 문자열 사용, 개별기능 테스트 불가.
- 

```

// 파일 읽기만 담당
public class WordFileReader {
    public String[] readWords(String filePath) throws IOException {
        try (FileReader reader = new FileReader(filePath)) {
            char[] chars = new char[1024];
            reader.read(chars);
            return new String(chars).split(" \\0");
        }
    }
}

// 정렬만 담당
public class WordSorter {
    public List<String> sort(String[] words) {
        List<String> sorted = Arrays.asList(words);
        sorted.sort(null);
        return sorted;
    }
}

// 파일 쓰기만 담당

```

```

public class WordFileWriter {
    public void writeWords(List<String> words, String filePath) throws IOException {
        try (FileWriter writer = new FileWriter(filePath)) {
            for (String word : words) {
                writer.write(word);
                writer.write("\n");
            }
        }
    }
}

// 전체 프로세스 조정
public class WordProcessor {
    private WordFileReader reader;
    private WordSorter sorter;
    private WordFileWriter writer;

    public WordProcessor() {
        this.reader = new WordFileReader();
        this.sorter = new WordSorter();
        this.writer = new WordFileWriter();
    }

    public void processWordFile(String inputPath, String outputPath) throws IOException {
        String[] words = reader.readWords(inputPath);
        List<String> sorted = sorter.sort(words);
        writer.(sorted, outputPath);
    }
}

```

- 각 클래스가 **단일 책임**만 수행
- 파일 경로를 **파라미터**로 받아 재사용 가능
- 각 클래스를 독립적으로 **테스트** 가능
- 필요시 각 컴포넌트를 쉽게 **교체** 가능 (예: 다른 정렬 알고리즘)

### 3. TDD와 응집성 강화의 관계

테스트 주도 개발(TDD)은 설계를 추진함으로써 코드의 응집성을 높이기 위해 설계에 **압력**을 가할 수 있다.

- **설계 품질 피드백:** 테스트 가능한 설계를 달성하려는 노력은 응집성을 위한 **최적의 지점**에 도달하도록 장려한다.
- **테스트의 역할:** 테스트 작성이 어렵다면, 이는 설계가 잘못되었음을 의미하며 응집성에 도 문제가 있을 가능성이 높다. TDD의 규율은 응집성을 높이는 기술과 경험을 증폭시킨다.

### 4. 응집성과 모듈성, 조직 구조

응집성은 모듈성(Modularity)과 밀접하게 관련되어 있다. 응집성은 모듈 내부 요소들의 관련 정도를 측정하는 반면, 모듈성은 시스템을 분리하는 기준을 제공한다.

#### 조직적 응집성

응집성의 개념은 코드를 넘어 조직 구조에도 적용되어, **조직의 효율성과 확장성**에 영향을 미친다.

- **고성과 팀의 특징:** '데브옵스 현황' 보고서에 따르면, 높은 성과를 예측하는 주요 요인 중 하나는 팀이 팀 외부의 누구에게도 허락을 구할 필요 없이 스스로 결정을 내릴 수 있는 능력이다.
- **정보와 기술의 결합:** 이는 팀의 정보와 기술에 응집성(cohesion)이 높다는 것을 의미하며, 팀이 결정을 내리고 진전을 이루기 위해 필요한 모든 것을 갖추고 있음을 뜻한다.

#### 균형의 필요성

- 응집성은 **모듈성의 반대 개념**으로, 이 둘 사이의 균형을 잡는 것이 중요하다. 이 균형을 잡는 데 가장 효과적인 도구 중 하나는 **관심사 분리**다.
- 응집성은 단순히 코드를 한곳에 모으는 것이 아니라, 함께 변화해야 하는 개념들을 논리적으로 묶어 코드를 읽기 쉽고, 유연하며, 변경하기 안전하게 만드는 핵심적인 설계 품질 속성이다.