2주차(6~9장)

6장 범용 모듈은 더 깊다

- 새모듈을 설계할 때 고민
  - 범용 또는 특수목적
- 범용 솔루션
  - 나중에 시간을 절약하기 위해 조금 더 많은 시간을 미리 투자하는 것
  - 실제로 필요하지 않은 기능이 포함될 수도 있음
- 특수 목적 접근 방식
  - 나중에 리팩토링해서 범용으로 만들수도 있음
  - 증분 접근 방식과 유사

6.1 클래스를 다소 범용적으로 만들기

- 최적은 다소 범용적으로 만들기
- 다소: 현재 요구사항은 반영하지만 해당 인터페이스는 여러 용도를 지원할 수 있을 만큼 충분이 일반적이어야함
- 범용 접근 방식은 특수목적 접근방식보다 더 간단하고 더 깊은 인터페이스를 제공한다.

6.2 예: 편집기용 텍스트 저장

- GUI에서 보이는대로 기능을 하는 클래스가 아니라, 범용으로 사용할 수 있는 것을 만들어야 좋다
- 지우기 기능
  - 특수 목적 접근 방식: 왼쪽 문자 선택 + 삭제하기
- 사용자 인터페이스와 텍스트 클래스 간 정보유출을 일으켰다.

6.3 더 범용적인 API

- 텍스트 기능의 관점에서만 정의되어야 한다.
- 텍스트 수정

    - 삽입(포지션 위치, 새로운 텍스트)
    - 삭제(시작위치, 종료위치)
- 위와같은 기능을 사용하면 다양한 삭제 키 구현 가능
- 대화식 편집기 이외에도 다른 용도로 사용 가능

6.4 보편성은 더 나은 정보은닉으로 이어짐

- 범용 접근 방식은 텍스트와 사용자 인터페이스 클래스를 보다 명확하게 구분해 정보 숨기기를 향상시킨다.

6.5 자신에게 물어볼 질문

- 인터페이스의 범용과 특수 목적 사이에서 균형을 찾는 질문들

1. 현재 요구사항을 모두 충족하는 가장 간단한 인터페이스는 무엇입니까?
    - 메서드의 수를 줄인다면 범용적으로 가고있는 좋은 신호
2. 이 메소드는 얼마나 많은 상황에서 사용됩니까?
    - 여러 특수목적 메서드를 단일 범용 메서드로 대체할 수 있는지 확인
3. 이 API는 현재 요구사항에 사용하기 쉬운가요?

7장 다른 계층, 다른 추상화

- 파일시스템에서
    - 최상위 계층: 파일 추상화 구현체. 가변 길이 바이트 배열로 구성
    - 최하위 계층: 보조 저장장치와 메모리간에 블록을 이동하는 장치 드라이버
- 네트워크 전송 프로토콜에서
    - 최상위 계층: 바이트 스트림
- 시스템에 유사한 추상화가 있는 인접 계층이 포함된 경우 이는 클래스 분해에 문제가 있음을 나타냄

7.1 통과 메소드 (Pass-through methods)

- 인접 레이어에 유사한 추상화가 있는 경우 문제는 통과 메소드의 형태로 나타남. 동일한 다른 메서드를 호출하는 것 외에는 거의 수행하지 않는 메서드

> 💡 위험 신호: Pass-through method
>
> 통과 메서드는 통과 메서드와 같은 API를 사용해 다른 메서드에 인수를 전달하는 것 외에는 아무것도 수행하지 않는 방법이다. 클래스간에 명확한 책임 분담이 없음을 나타낸다.

- 통과 메서드는 클래스를 더 얕게 만든다.
  - 복잡성을 추가하지만 시스템의 기능을 증가시키지는 않는다.
- 해결책: 각 클래스가 고유하고 일관된 책임 집합을 갖도록 클래스를 리팩토링하는 것
  - 바로 하위 계층을 참조하도록 만들던가
  - 계층을 찢던가
  - 동일한 계층으로 병합하던가.

7.2 인터페이스 중복은 언제 허용되나

- 중요한 것은 각각의 새로운 메서드가 중요한 기능을 제공해야한다.
- 허용되는 예: Dispatcher
  - 여러 다른 메서드 중 어떤 메서드가 각 작업을 수행할 지 선택하는 유용한 기능 제공

7.3 데코레이터

- 데코레이터 디자인 패턴(Wrapper)은 API 복제를 장려하는 패턴
- 기존 객체를 가져와 기능을 확장시킨다.
- 데코레이터는 클래스의 특수목적의 확장을 조금 더 일반적인 코어에서 분리하는 것이다.
- 얕은 경향이 있다.
- 자가 점검
  1. 데코레이터 클래스를 만들지 않고 기본 클래스에 직접 새 기능을 추가할 수 있습니까?
  2. 새 기능이 특정 사례에 특화된 경우 별도의 클래스르 만드는 대신에 사용 사례와 병합하는 것이 합리적입니까?

3. 새 데코레이터를 만드는 대신 기존 데코레이터와 새 기능을 병합할 수 있습니까?
4. 새로운 기능이 기존 기능을 감쌀 필요가 있는지 자문: 기본 클래스와 독립 클래스로 구현할 수 있을까?

7.4 인터페이스와 구현

- 클래스의 인터페이스와 일반적으로 구현이 달라야한다.
- 내부적으로 사용되는 표현은 인터페이스에 나타나는 추상화와 달라야한다.

7.5 통과 변수

- 계층간 API 복제의 또 다른 형태는 긴 메서드 체인을 통해 전달되는 변수(Pass-through Variable)
- 제거하는 방법
  - 최상위 메서드와 최하위 메서드간 공유된 객체가 이미 있는지 확인하는 방법
  - 전역변수에 저장하는 방법. 추천하지 않음
  - context 객체를 설정하는 것. 추천함. 하지만 이상적이지는 않음

8장 복잡성을 낮추기

- 개발자보다 사용자가 더 많기 때문에 모듈 내부적으로 복잡성을 처리하는 것이 더 좋음

8.1 예: 편집기 텍스트 클래스

- 줄을 나누거나 병합하는 것이 아닌 임의 범위를 조정하는 식으로 구현하면 더 간단해진다.
- 이 접근 방식은 텍스트 클래스 내에서 분할 및 병합의 복잡성을 캡슐화해 복잡성을 줄인다.

8.2 예: 구성 매개변수

- 요청 재시도 횟수, 캐시크기 등
- 구성 매개변수는 성능적으로 유용하지만 사용자입장에서는 그 값을 결정하는 것이 어려울 수 있다.
- 구성 매개변수 굳이 안써도 되는 거면 쓰지 말자

- 자동으로 해결할 수 있는 문제라면 해결하자

8.3 너무 지나침

- 복잡성을 낮출때는 신중하게 하자.
- 모든 기능을 하나의 클래스로 내리는건 말도안된다.
- 복잡도를 낮추는 때
  1. 풀다운되는 복잡성이 클래스의 기존 기능과 밀접하게 관련되어있는 경우
  2. 복잡성을 낮추면 애플리케이션의 다른 곳에서 많은 단순화가 이루어지는 경우
  3. 복잡성을 낮추는 경우 클래스의 인터페이스가 단순화 되는 경우

9장 더나은 결합 또는 더 나은 분리

- 두 가지 기능이 주어졌을 때 동일한 위치에서 함께 구현하는가 또는 구현을 분리해야 하는가
  - 시스템의 복잡성을 줄이고 모듈성을 개선하는 방향으로
  - 세분화를 하면 개별 구성요소가 단순해지지만 추가 복잡성을 생성한다.
  - 일부 복잡성은 구성요소의 수에서 비롯된다
  - 세분화하면 구성요소를 관리하기 위한 추가 코드가 생성된다.
  - 세분화는 분리를 생성한다.
  - 중복 발생 가능
- 코드를 함께 사용하는 것은 밀접하게 관련된 경우 유용하다.
  - 관련 없으면 분리하는것이 더 나을 수 있다.
  - 관련 있는지 체크리스트
    - 정보를 공유하는지
    - 함께 사용되는지
    - 두 코드 조각을 포함하는 간단한 상위 수준 범주가 있는 경우 개념적으로 겹치는 경우
    - 다른 코드를 보지 않고서는 코드중 하나를 이해하기 어려운 경우

9.1 정보가 공유되는 경우 결합하기

- 구문을 읽고 분석해야하는 경우 두개가 정보를 공유하니 한 클래스로 결합하는 것이 좋다.

9.2 인터페이스를 단순화 하려면 함께 결합하기

- 두 개 이상의 모듈이 하나의 모듈로 결합되면 원래 인터페이스보다 더 간단하거나 사용하기 쉬운 새 모듈에 대한 인터페이스를 정의 가능하다.

9.3 중복을 없애기 위해 결합하기

- 반복되는 코드가 있다면 코드를 리팩토링해서 반복을 제거해라

9.4 범용코드와 특수코드 분리하기

- 여러 다른 목적으로 사용할 수 있는 메커니즘이 포함되어있는 경우 해당 모듈은 하나의 범용 메커니즘만 제공해야한다.
- 하위 계층은 범용이고 상위계층은 특수목적인 경우가 많다.

위험신호: 반복

반복이 있는경우 제대로 추상화되지 않았다는 증거

9.5 예: 삽입 커서 및 선택

위험 신호: 특별-일반 혼합물 (Special-General Mixture)

범용 메커니즘에 해당 메커니즘의 특정 용도에 특화된 코드도 포함되어 있을 때 발생
더 복잡하게 만들고 정보누출을 만들음

9.6 예: 로깅을 위한 별도의 클래스

- 로깅 메서드를 제거하고 오류가 감지된 위치에 로깅문을 배치하는 것이 더 좋다.

9.7 예: 편집기 실행 취소 메커니즘

- 실행 취소 기능을 세가지 범주로 나누고 각각 다른 위치에서 구현됨

    - 작업을 관리 및 그룹화하고 실행 취소/ 다시실행 작업을 호출하기 위한 범용 메커니즘
    - 특정 작업의 세부사항
    - 작업을 그룹화하기 위한 정책

9.8 분리 및 결합 방법

- 기존 메서드를 분할 및 결합하는 기준이 있을 수 있다.
    - 라인 수 기반도 있지만 옳지는 않다.
    - 긴 방법이 나쁘진 않다.
- 메서드를 설계할때는 항상 깨끗하고 단순한 추상화를 해야한다.
- 각 메서드는 한가지 일을 완벽하게 수행해야 한다.
- 추상화를 생성하는 방법 두가지
    - 하위작업을 부모자식 별도의 메서드로 분해하는 것
    - 원래 메서드의 호출자에게 각각 표시되는 두개의 개별 메서드로 분할하는 것 (같은 계층)

위험신호: 결합된 메서드

각 메서드를 독립적으로 이해할 수 있어야 한다. 다른 메서드의 구현도 이해하지 않고 한 메서드의 구현을 이해할 수 없다면 위험신호이다. 또는 두 개의 코드 조각이 물리적으로 분리되어있지만 각각은 다른 쪽을 봐야만 이해할 수 있는 경우도 위험신호이다.