4주차(15~마지막)

15장 주석 먼저 작성

> 설계 과정의 일부로 주석 사용

15.1 늦은 주석은 나쁜 주석이다.

- 주석을 미루는 것은 별로 좋지 않다.
- 코드 완성이 되고 나서 주석을 작성하기 때문에 주석에 힘을 쏟지 않는다.

15.2 주석 먼저 작성

- 방법
  1. 새 클래스의 인터페이스 주석 작성
  2. 그다음 중요한 공용 메서드에 대한 인터페이스 주석 및 선언 작성 (메서드 본문 비움)
  3. 기본 구조가 옳다고 생각할 때까지 1~2 반복
  4. 클래스에서 중요한 인스턴스 변수에 대한 선언과 주석 작성
  5. 메서드 본문을 채우고 구현 주석을 추가
  6. 리팩토링을 하며 새로운 인터페이스 주석을 달고 메소드를 추가하고 구현하고 구현 주석을 달음
- 장점
  - 더 나은 주석을 생성한다.
  - 설계를 개선한다.
  - 주석 작성이 재밌어진다.

15.3 주석은 설계 도구

- 주석은 추상화를 캡쳐할 수 있는 도구이다
- 주석은 복잡성을 미리 알 수 있게 해준다.
  - 메서드나 변수에 긴 주석이 필요한 경우 좋은 추상화가 없다는 것

- 인터페이스 주석을 통해 알아차릴 수 있음

> 💡 위험신호: 설명하기 어려움
>
> 메서드나 변수를 설명하는 주석은 단순하면서도 완전해야 한다. 그렇지 않다면 설계에 문제가 있다.

15.4 초기 주석은 재미있다.

- 설계단계 때 쉽게 작성할 수 있으니 재미있어진다.

15.5 초기 주석은 비용이 많이 드는가?

- 물론 추가 시간이 들겠지만 전체 개발 비용의 10% 채 되지 않을 것
- 주석을 작성함으로 나중에 더 좋아질 수 있음

16장 기존 코드 수정

16.1 전략적 유지

- 전술적 프로그래밍: 추가 복잡성을 초래하더라도 빠르게 기능을 구현
- 전략적 프로그래밍: 훌륭한 시스템 설계를 생성하는 것을 중점적으로 한다.
- 최소 기능 변경만 하지 말고 원하는 변경사항을 고려해 현재 시스템 설계가 최상인지 확인해보기

16.2 주석 유지하기: 주석을 코드 근처에 두기

- 기존 코드를 수정할 때 기존 주석도 수정해야한다.
- 코드 가까이에 주석을 배치하여 개발자가 코드르 변경할 때 주석을 볼 수 있도록 해야한다.
- 구현 주석을 작성할 때 전체 메서드에 대한 모든 주석을 메서드 맨 위에 두지 마라.

16.3 주석은 커밋 로그가 아니라 코드에 속한다.

- 커밋 로그에 자세히 쓰지말고 주석에 자세히 써라

16.4 주석 유지: 중복 방지

- 주석을 최신상태로 유지하려면 중복을 피해라
- 여러 코드에 중복으로 들어가야할 내용이 있다면 다른 곳에 써두고 링크를 걸어라
- 메서드 호출 앞에 주석을 또 쓰지마라

16.5 주석 유지: diff 확인

- 버전관리 시스템에 올리기 전에 몇분만 커밋에 대한 모든 변경사항을 검사해라.
- todo나 fixme 같은 것들을 남기는 것을 감지할 수 있다.

16.6 상위 수준 주석은 유지관리가 더 쉽다.

- 주석은 코드보다 상위수준이고 추상적인 경우 유지관리가 더 쉽다.

17장 일관성

- 일관성: 시스템의 복잡성을 줄이고 동작을 보다 명확하게 만드는 강력한 도구
- 일관성을 실수를 줄여준다.

17.1 일관성의 예

- 이름, 코드 스타일, 여러 구현이 있는 인터페이스, 디자인 패턴, 불변성

17.2 일관성 보장

- 문서 스타일 가이드, 자동화된 검사기, 코드리뷰, 기존 코드 참조하기, 조직에서 너무 변경하려고 하지 말기

17.3 너무 지나침

- 유사하지 않은 일들은 다른 방식으로 진행되어야 한다.
- 어울리지 않는 변수명/디자인 패턴 등을 활용하면 복잡성과 혼란을 야기한다.

18장 코드는 명확해야한다.

- 모호함은 복잡성의 주요 원인

- 모호함은 시스템에 대한 중요한 정보가 새로운 개발자에게 명확하게 전달되지 않을 때 발생
    - 코드를 명확하게 작성해서 모호함을 해결

18.1 코드를 더 명확하게 만드는 것들

- 좋은 이름을 선택하는 것
- 일관성을 지키는 것
- 빈칸(탭)이나 줄바꿈 등을 잘 써서 한번에 인식되도록 만들기
- 주석을 사용해서 누락된 정보를 제공하기

18.2 코드를 덜 명확하게 만드는 것들

- 이벤트 기반 프로그래밍: 제어흐름을 따르기가 어렵다.
    - 이를 보완하려면 각 핸들러 함수에 대한 인터페이스 주석을 사용해서 언제 호출되는지 표기하기

> 💡 위험 신호: 명확하지 않은 코드
>
> 빠른 읽기로 코드의 의미와 동작을 이해할 수 없다면 위험 신호이다.

- 제네릭 컨테이너: 제네릭은 그룹화 된 요소에 의미를 모호하게 하는 일반 이름이 있기 때문에 명확하지 않은 코드를 생성한다.
    - 사용하지 않는 것이 좋다. 차라리 새로 만들자
- 선언 및 할당을 위한 다양한 유형
    - 할당과 선언을 일치시키는 것이 좋음
- 독자의 기대를 위반하는 코드

19장 소프트웨어 동향

19.1 객체지향 프로그래밍과 상속

- 상속의 소프트웨어 복잡성 관점에서의 의미 두 가지

- 인터페이스 상속
  - 동일한 인터페이스를 여러번 사용해서 복잡성에 대한 영향력 제공
  - 인터페이스의 구현이 다양할 수록 인터페이스가 더 깊어진다.
- 구현 상속
  - 시스템이 발전함에 따른 수정해야하는 코드의 양을 줄임
  - 하위클래스간에 종속성을 만들기에 주의해서 사용해야한다.

19.2 애자일 개발

- 민첩한 개발의 가장 중요한 요소 중 하나는 개발이 점진적이고 반복적이어야 한다.
- 전술적 프로그래밍으로 이루어질 수 있다.
- 증분식 개발은 일반적으로 좋지만 개발 증분은 기능이 아닌 추상화여야 한다.

19.3 단위 테스트

- 테스트가 개발과 밀접하게 통합되어야 한다.
- 단위 테스트와 시스템 테스트로 나뉘어짐

19.4 테스트 주도 개발

- 테스트 주도 개발은 코드를 작성하기 전 단위 테스트를 작성하는 소프트웨어 개발 접근 방식
- 꼭 정답은 아님
  - 기능을 구현하는데 힘을 쏟게 만들기 때문
  - 너무 점진적이다.
- 개발 단위는 기능이 아닌 추상화여야 한다.
- 테스트를 먼저 작성하는 것이 합리적인 때는 버그를 수정할 때이다.

19.5 디자인 패턴

- 잘 알려진 디자인 패턴을 사용하면 된다.
- 하지만 과잉적용하면 안된다.

19.6 게터와 세터

- 반드시 필요하지는 않는다.
- 애초에 인스턴스 변수를 노출하는 것이 좋지 않다.
- getter/setter는 얕은 메서드이고 인터페이스를 복잡하게 만들기 때문에 피하는 것이 좋다.

20장 성능을 위한 설계

20.1 성능에 대해 생각하는 방법

- 정상적인 개발 과정에서 성능에 대해 얼마나 걱정해야 하는가?
- 최대 속도를 위해 모든 명령문을 최적화하려고 하면 개발 속도가 느려지고 불필요한 복잡성이 많이 발생한다.
- 대부분의 최적화는 실제로 도움이 되지 않는다.
- 가장 좋은 접근 방식은 성능에 대한 기본 지식을 사용하여 “자연스럽게 효율적”이면서도 깨끗하고 단순한 설계 대안을 선택하는 이러한 극단 사이에 있는 것이다.
- 예
    - 네트워크 통신
    - 입출력 저장소의 속도
    - 동적 메모리 할당/해제
    - 캐시 미스

20.2 수정 전 측정

- 느린 것 같다는 직관에 따라 변경을 시작하면 실제로 느리지도 않은 일에 더 복잡하게 만들 수 있음
- 변경 전에 시스템의 기존 동작을 측정해라
    - 측정을 통해 성능 조정이 가장 큰 영향을 미칠 위치를 식별한다.
    - 성능을 다시 측정할 수 있도록 기준선을 제공한다.

20.3 요주의 경로를 중심으로 설계

- 성능을 향상시키는 가장 좋은 방법은 캐시 도입이나 다른 알고리즘 접근법을 사용하는 것 같은 근본적인 변경을 사용하는 것

- 근본적인 해결책이 없는 경우 설계를 변경해야한다.
    - 작업을 수행하는데 가장 간단한 코드를 다시 짜보자