1주차(1~5장)

1장 소개 (복잡성에 관한 모든 것)

- 소프트웨어 작성의 가장 큰 한계는 우리가 만들고 있는 시스템을 이해하는 능력
- 복잡성과 싸우기 위한 두가지 접근 방식
  - 코드를 더 간단하고 명확하게 만들어 복잡성 제거
  - 캡슐화하여 프로그래머가 모든 복잡성에 한 번에 노출되지 않고 시스템에서 작업할 수 있도록 하는 것(모듈식 설계)
- 소프트웨어 설계법
  - 폭포수 모델: 소프트웨어 개발에 적합하지 않음. 유연성 부족
  - 애자일(점진적 접근 방식): 시스템을 수시로 평가하고 변경하면 초기 문제를 빠르게 고칠 수 있다.
- 증분 개발
  - 소프트웨어 설계가 완료되지 않음
  - 설계 변경은 시스템 수명동안 계속 일어나고 개발자는 항상 설계 문제에 대해 생각해야한다.
  - 점진적 개발: 지속적인 재설계
- 이 책의 목표
  - 소프트웨어 복잡성의 특성을 설명하는 것
  - 소프트웨어 개발 프로세스중에 복잡성을 최소화하기 위해 사용할 수 있는 기술을 제시하는 것.

1.1 이책을 사용하는 방법

- 설계원칙 중 대부분이 추상적이므로 실제 코드를 보는 것이 좋다. 그래서 코드리뷰를 하며 읽어라.
- 설계 기술을 향상시키기 위해 주요 설계 문제와 관련된 문제를 야기하는 위험 신호를 인식하는 방법을 찾는다.
- 이 책의 아이디어를 적용할 때 절제와 분별력을 사용하는 것이 중요하다.

2장 복잡성의 본질

- 복잡성을 인식하는 능력은 중요한 설계 기술이다.
- 시스템이 복잡한 것을 인지하는 것이 중요.

2.1 복잡성 정의

- 복잡성: 시스템을 이해하고 수정하기 어렵게 만드는 소프트웨어 시스템의 구조와 관련된 모든 것.
  - 복잡한 시스템은 작은 개선사항을 적용하는데도 많은 작업이 필요하다.
  - 간단한 시스템에서 더 적은 노력으로 더 큰 개선을 구현할 수 있다.
- 복잡성은 개발자가 특정 목표를 달성하려고 할 때 특정 시점에서 경험하는 것.
- 다른사람이 보기에 복잡하면 복잡한 것이다.

2.2 복잡성의 증상

- 변경 증폭(Change amplification)
  - 단순해 보이는 변경에도 여러 위치에서 수정이 필요
  - 좋은 설계는 각 디자인 결정에 영향을 받는 코드의 양을 줄이는 것이므로 디자인 변경에 많은 코드 수정이 필요하지 않아야한다.
- 인지 부하(Cognitive load)
  - 개발자가 작업을 완료하기 위해 알아야하는 양이 너무 많아지면 안된다.
  - 미리 버그가 발생할 여지나 알아야 할 지식이 있다면 정리해두면 인지 부하를 줄일 수 있다.
- 알려지지 않은 미지수(unknown unknowns)
  - 작업을 완료하기 위해 어떤 코드 조각을 수정해야 하는지 또는 무엇을 수정해야 하는지 명확하지 않다.
  - 이게 제일 최악이다.
- 좋은 설계의 중요한 목표는 시스템이 명확해지는 것.

2.3 복잡성의 원인

- 복잡성은 종속성과 모호성에 의해 발생함

- 종속성이 생기는 기준: 주어진 코드 조각을 단독으로 이해하고 수정할 수 없을 때
    - 종속성의 수를 줄이고 가능한 한 단순하고 분명한 종속성을 유지해야한다.
- 모호함: 중요한 정보가 명확하지 않을 때 발생 (시간의 단위 등)
    - 종속성이 존재하는지 명확하지 않을때도 불명확하다고 판단한다.
    - 불일치도 모호함의 주요 원인
    - 불충분한 문서화로 모호함이 발생함
    - 설계 문제이기도 하다.

2.4 복잡성은 점진적이다.

- 복잡성은 단일 치명적인 오류로 인해 발생하지 않고, 많은 작은 덩어리로 축적된다.
- 복잡성의 증분 특성으로 인해 제어하기가 어렵다.
- 보이자 마자 바로 제거해야한다.

2.5 결론

- 복잡성은 종속성과 모호성의 축적에서 비롯된다.
- 결국 복잡성으로 인해 기존 코드를 수정하는 것이 어렵고 위험해진다.

3장 작업코드로는 충분하지 않습니다(전략 대 전술적 프로그래밍)

- 전술적 사고 방식: 가능한 한 빨리 기능을 구현하는 데 중점
- 전략적 사고 방식: 시간을 투자해 깔끔한 설계를 추구
- 장기적으로 전략적 사고 방식이 더 비용이 적게 들어감

3.1 전술 프로그래밍

- 새로운 기능이나 버그 수정과 같은 작업을 수행하는 것
- 문제
    - 근시안적
    - 이와 같이 처리하다보면 복잡성이 아주 커진다.
- 전술적인 코드를 작성해서 다작할 수는 있지만 파괴의 흔적을 남긴다.

3.2 전략적 프로그래밍

- 작업 코드만으로는 충분하지 않다.
- 시스템의 장기적인 구조를 고려해야한다.
- 애초에 많이 고민을 두고 여러 대안을 두고 최선을 고르고, 만약 실수했다고 해도 전략적으로 시스템 설계를 조금씩 개선해라.

3.3 얼마나 투자할 것인가?

- 설계만 주구장창하는 것은 좋지 않다. 개발 시간의 10~20%를 추자하라
- 초기 개발시에는 20%를 초과할 수 있지만, 머지않아 그 이상을 아낄 수 있다.

3.4 스타트업과 투자

- 스타트업은 일정에 쫓겨 전술적으로 하려고 한다.
- 결국 전술적인 개발이 발목을 붙잡을 것
- 엔지니어의 품질도 굉장히 유용함. 훌륭한 엔지니어는 비용 대비 생산성이 높다. 이것은 좋은 설계에 기반.
- facebook은 전술적, google과 vmware는 전략적으로 했는데 지금 부채 비율이 다르다.

3.5 결론

- 좋은 디자인은 공짜로 오지 않는다.
- 설계 개선을 지연하면 지연은 영구가된다.

4장 모듈은 깊어야 합니다

- 모듈식 설계: 개발자가 주어진 시간에 전체 복잡성의 작은 부분만 직면하면 되도록 시스템을 설계하는 법

4.1 모듈형 디자인

- 모듈식 설계에서 소프트웨어 시스템은 상대적으로 독립적인 모듈 모음으로 분해됨
  - 하지만 결과적으로 모듈 서로간에 메소드나 함수를 호출해야하기 때문에 알아야한다.
  - 이러면 종속성이 생김

    - 모듈식 설계의 목표는 모듈간의 종속성을 최소화하는 것.
- 종속성을 관리하기 위해 각 모듈을 인터페이스와 구현의 두 부분으로 생각한다.
    - 인터페이스는 다른 모듈에서 작업하는 개발자가 알아야 하는 것만 알음
    - 실제 구현은 인터페이스를 이용해 수행하는 코드에서 이루어 짊
- 모듈은 인터페이스와 구현이 있는 코드 단위이다.
- 최고의 모듈은 인터페이스가 구현보다 훨씬 간단한 모듈이다.
    - 두가지 장점을 가지고 있음
        - 간단한 인터페이스는 모듈이 나머지 시스템에 부과하는 복잡성을 최소화 한다.
        - 인터페이스를 변경하지 않는 방식으로 모듈을 수정하면 다른 모듈은 수정의 영향을 받지 않는다.

4.2 인터페이스에는 무엇이 있나요?

- 공식 정보
    - 코드에 명시적으로 지정됨.
    - 일부는 프로그래밍 언어로 정확성 확인 가능
    - ex) 매개변수 이름, 타입, 반환값 타입, 예외 등
- 비공식 정보
    - 프로그래밍 언어에서 이해하거나 시행할 수 있는 방식으로 지정되지 않음.
    - 함수가 인수중 하나에 의해 명명된 파일을 삭제한다는 사실과 같은 고수준 동작이 포함됨
    - 주석을 통해서만 설명할 수 있음
- 비공식적인 정보가 공식적인 정보보다 더 크고 복잡하다.

4.3 추상화

- 추상화: 중요하지 않은 세부 종보를 생략한 엔티티의 단순화된 보기
    - 복잡한 것에 대해 생각하고 조작하는 것을 더 쉽게 만들어주기 때문에 유용함
- 추상화에서 생략된 중요하지 않은 세부사항이 많을수록 좋다. 세부사항은 중요하지 않은 경우에만 생략할 수 있다.
- 추상화가 두가지 방법으로 잘못될 수 있음

    - 실제로 중요하지 않은 세부 정보를 포함할 수 있음
    - 추상화가 실제로 중요한 세부 정보를 생략하는 경우이다.

4.4 딥 모듈

- 최고의 모듈: 강력한 기능을 제공하면서 인터페이스가 단순한 모듈 (깊이가 깊다)
- 모듈이 제공하는 이점은 기능. 모듈의 비용은 인터페이스이다.
- 심층 모듈은 사용하기 쉬우면서도 상당한 구현 복잡성을 숨기기 때문에 강력한 추상화를 제공한다.

4.5 얕은 모듈 - Red flag

> ❓ p37 얕은모듈
> 함수형 프로그래밍 방식이 대부분 이렇게 진행되지 않나요? 그러면 대부분이 얕은 모듈이 될 수밖에 없는 것 같은데, 정확하게 Red flag까지로 인식하는 것이 맞는지 여러분의 의견이 궁금합니다.

- 제공하는 기능에 비해 인터페이스가 상대적으로 복잡한 모듈
- 메서드가 문서화되어 있다면 메서드 코드보다 문서가 더 길어지는 경우.
- 호출자가 메서드를 통해 데이터를 조작하는데 걸리는 것 보다 메서드를 호출하는데 더 많은 키 입력이 필요한 경우.
- Red flag: 복잡성과의 싸움에서 그다지 도움이 되지 않음

4.6 클래스염(classitis)

- 각각의 새 클래스에서 기능의 양을 최소화하도록 권장됨
- 단순한 클래스를 여러개 만들 수 있겠지만 결과적으로 복잡성을 증대시킨다.

4.7 예제: java 및 unix I/O

- 오늘날 classitis의 제일 눈에 띄는 예는 Java클래스 라이브러리이다.
- 필요한 기능을 한 모듈로 묶고 인터페이스를 만들어 사용자가 여러 메커니즘을 인식할 필요가 없도록 하는 것이 좋다.

4.8 결론

- 구현에서 모듈의 인터페이스를 분리해서 시스템의 나머지 부분에서 구현의 복잡성을 숨길 수 있다.
- 모듈 사용자는 인터페이스에서 제공하는 추상화만 이해하면 된다.
- 간단한 인터페이스를 가지면서 중요한 기능만을 제공하도록 깊이있게 만드는 것이 좋다.

5장 정보 은닉(및 유출)

- 딥 모듈을 만드는 방법에 대해 배운다.

5.1 정보 은닉(Information hiding)

> ❓ p41 정보 은닉
> 설명으로 미루어 보자면 사용하는 곳에서는 최대한 정보를 필요없도록 하게 만들고, 구현하는 곳에서 모두 정보를 은닉해서 복잡도를 줄이자는 이야기 같습니다.
> 여러분은 동의하시나요?
> 예) Pagination API를 구현할때 클라이언트에게 page만 받고 limit, offset 등은 해당 API 내에서 처리

- 정보 은닉: 딥 모듈을 달성하기 위한 중요한 기술
- 기본 아이디어: 각 모듈이 설계 결정을 나타내는 몇 가지 지식을 캡슐화해야 한다.
- 숨겨진 정보는 일부 메커니즘을 구현하는 방법(데이터 구조 및 알고리즘)에 대한 세부정보로 구성됨
- 정보 은닉는 두가지 방법으로 복잡성을 줄인다.
  - 모듈에 대한 인터페이스를 단순화 한다.
  - 시스템의 진화를 더 쉽게 만든다.
- class의 private으로 정보은닉하는 것과는 다른 개념이다. 개별 요소는 클래스 외부에서 항목에 직접 액세스하는 것을 불가능하게 하므로 정보 숨기기에 도움이 될 수 있다. 하지만 getter, setter와 같은 공용 메서드로 노출 가능하다.
- 정보 은닉의 가장 좋은 형태는 정보가 모듈 내에 완전히 숨겨져있어 모듈 사용자에게 관련이 없고 보이지 않는 것이다.

5.2 정보 유출(Information leakage) - Red flag

- 설계 결정이 여러 모듈에 반영될 때 정보유출이 발생
  - 이렇게 하면 종속성이 생기고 복잡도가 올라감
- 정보의 일부가 모듈의 인터페이스에 반영되면 유출된 것
- 정보가 인터페이스에 나타나지 않더라도 정보가 유출될 수 있다.
  - 두 클래스가 특정 파일 형식에 대해서 알고있으면, 파일 형식이 바뀔경우 두 클래스도 변경되어야 한다.
  - 굉장히 치명적!
- 클래스 간 정보유출이 발생한 경우 “단일 클래스만 영향이 미치도록 재설계하려면 어떻게 해야할지” 고민해야한다.
  - 영향 받는 클래스가 작고 밀접한 정보를 유출한 경우 단일 클래스로 병합
  - 모든 클래스에서 정보를 가져오고 해당 정보만 캡슐화 하는 새 클래스 생성
    - 세부 정보에서 벗어나 추상화하는 간단한 인터페이스를 찾을 수 있는 경우에만 효과적

5.3 시간 분해 (Temporal decomposition) - Red flag

- 시간 분해는 정보 유출의 일반적인 원인임
- 시간분해에서 시스템의 구조는 작업이 발생하는 시간 순서에 해당함
- 시간 분해의 함정에 빠지기 쉬움
  - 기본적으로 코드는 순서에 맞춰서 짜기 때문
- 모듈을 설계할 때 작업이 발생하는 순서가 아니라 각 작업을 수행하는 데 필요한 지식에 중점을 두자.

5.4 예: HTTP 서버

- HTTP 프로토콜은 요청 및 응답의 형식을 지정하며 둘 다 텍스트로 표시된다.
- 학생들은 웹 서버가 들어오는 HTTP 요청을 수신하고 응답을 보내는 것을 쉽게 만들기 위해 하나 이상의 클래스를 구현하도록 요청 받았다.

5.5 예: 너무 많은 클래스

- 흔한 실수: 코드를 많은 수의 얕은 클래스로 나누어 클래스간의 정보 유출로 이어짐
- HTTP 요청을 수신하기 위해 두가지 다른 클래스를 사용한 경우

    - 첫 번째 클래스: 네트워크 연결의 요청을 문자열로 파싱
    - 두 번째 클래스: 문자열을 구문분석
    - 시간분해의 예로 정보유출이 발생함
        - 특정 순서로 두 클래스의 두 메서드를 사용해야하기 때문에 복잡성을 야기했다.
- 위와 같은 경우 단일 클래스로 변합하는 것이 더 좋다.
    - 지식을 격리하고 호출자에게 더 간단한 인터페이스를 줄 수 있기 때문에
- 클래스를 더 크게 만들어 정보 은닉을 개선해야하는 이유
    - 특정 기능에 관련된 모든 코드를 결합하여 결과 클래스에 모든 항목이 포함되도록 하기 위해
    - 인터페이스의 수준을 높이기 위해
- 하지만 너무 크게 만들면 안된다.

5.6 예: HTTP 매개변수 처리

- POST메소드의 요청에서 정보들은 헤더, query params, body 여러곳에 들어가있다.
- 대부분은 매개변수 관련 두 가지 좋은 선택이 있다.
    - 서버는 매개변수가 요청의 헤더인지 body인지 신경안쓰므로 둘을 병합한다.
    - HTTP 구문 분석기는 매개변수 값을 웹 서버로 반환하기 전에 디코딩 하므로 그림 5.1의 주석 매개변수 값은 올바른 문자열로 반환될 수 있다.
- 인터페이스를 너무 얕게 사용해서 단일 메서드를 사용하게 된다면 정보를 숨길 수 있는 기회를 잃게된다.
    - 받은 값을 그대로 내보내게 된다면(내부 노출) 해당 표현이 변경될 때 모든 호출자를 수정해야한다.
    - 이를 더 잘게 쪼개서 string과 int로 나누어 파라미터를 세분화하면 호출자는 그만큼 예외처리할 범위가 줄어들어서 좋다.

5.7 예: HTTP 응답의 기본값

- HTTP응답은 프로토콜 버전을 지정해야하는데, 호출자가 호출할 경우 오히려 모를 수 있으니 응답에 따른 프로토콜 기본값을 자동으로 채워주도록 하는 것이 좋다.
- 기본값은 일반적인 사례를 가능한 한 단순하게 만들기 위해 인터페이스를 설계해야한다는 것을 보여줌

위험신호: 과다노출
일반적으로 사용되는 기능에 대한 API가 거의 사용되지 않는 다른 기능에 대해 배우도록 강요하는 것은 인지부하가 증가한다.

5.8 클래스 내 정보 숨기기

- API와 관련된 정보 은닉 뿐만 아니라 클래스 내부와 같은 다른 수준에서도 적용 가능하다.

5.9 너무 지나친 행동

- 정보 은닉은 숨겨진 정보가 모듈 외부에서 필요하지 않은 경우만 의미가 있다.
    - 정보가 모듈 외부에서 필요한 경우 숨기면 안된다.

5.10 결론

- 모듈이 많은 정보를 숨기면 모듈이 제공하는 기능의 양이 증가하지만 인터페이스가 축소되는 경향이 있다. → 깊은 모듈을 만드는법
- 시스템을 모듈로 분해할 때 런타임에 작업이 발생하는 순서에 영향을 받지 않도록 해야 한다.