리팩토링

소프트웨어 행동은 유지하면서 내부 구조를 더 쉽게 이해하고 변경할 수 있도록 개선하는 작업.

인프런 / 백기선 (AKA, WHITESHIP)

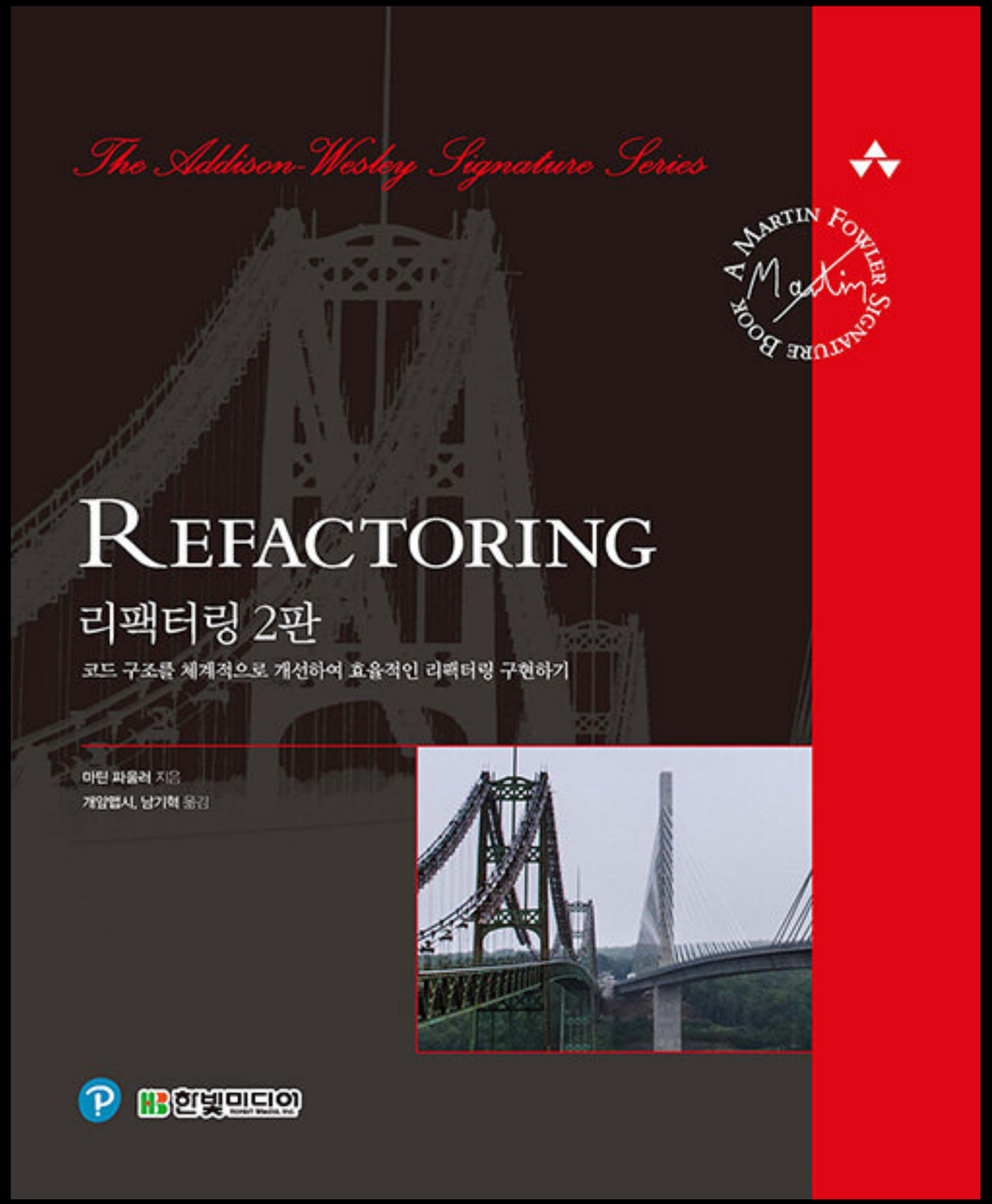
리팩토링 (Refactoring)

소프트웨어는 계속 변하기 때문에…

- 처음부터 완벽하게 시스템을 설계하는 것은 매우 어려운 일이다.
- 이미 코드를 작성한 이후에 구조를 변경하는 일이 발생한다.
- 리팩토링으로 애플리케이션 구조를 꾸준히 개선해 나가야 한다.
- 구조 변경으로 인한 버그를 줄이면서 코드를 깔끔하게 유지할 수 있는 방법이다.

리팩토링 (Refactoring)
저자, 마틴 파울러

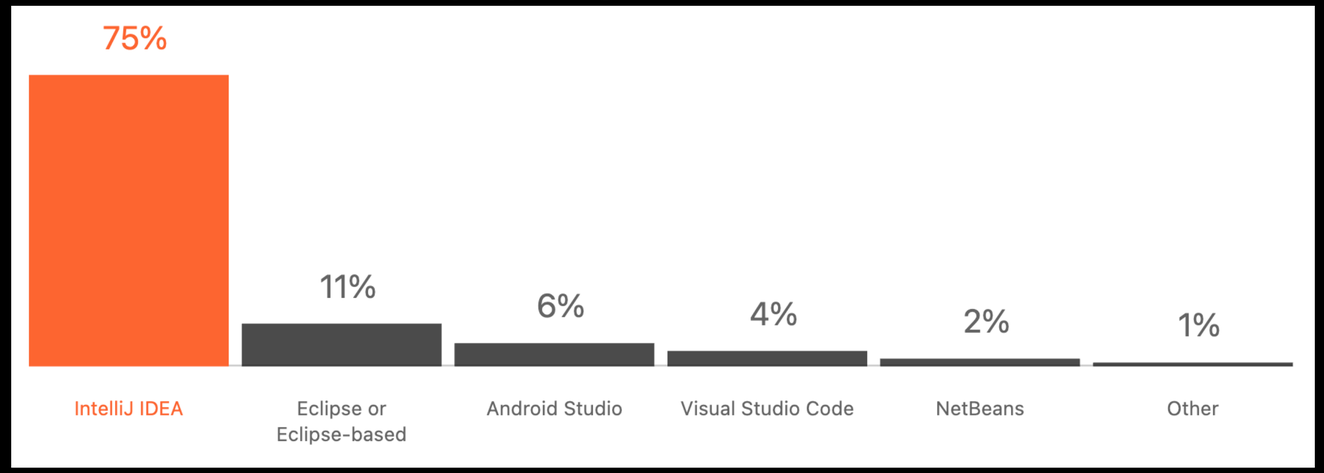
- 2018년 2판 발행, 2020년 한국어 번역본 발행
- 예제 코드로 자바스크립트를 사용하고 있다.
- 리팩토링 기술을 분류별로 살펴본다.
  - 기본 기술
  - 캡슐화 관련 기술
  - API 관련 기술
  - 상속 관련 기술
  - …



코딩으로 학습하는 리팩토링

조금 더 실용적으로 다루고 싶었습니다.

- 현실적으로 대부분의 개발자는 IDE를 사용하고 있습니다.
- 글이 아니라 영상으로 그 과정을 보여주고 싶다.
- 자바로 작성된 예제 코드를 제공합니다.
- 냄새 위주로 리팩토링을 살펴보자.



https://www.jetbrains.com/lp/devecosystem-2021/java/

냄새 1. 이해하기 힘든 이름

Mysterius Name

- 깔끔한 코드에서 가장 중요한 것 중 하나가 바로 “좋은 이름”이다.
- 함수, 변수, 클래스, 모듈의 이름 등 모두 어떤 역할을 하는지 어떻게 쓰이는지 직관적이어야 한다.
- 사용할 수 있는 리팩토링 기술
  - 함수 선언 변경하기 (Change Function Declaration)
  - 변수 이름 바꾸기 (Rename Variable)
  - 필드 이름 바꾸기 (Rename Field)

리팩토링 1. 함수 선언 변경하기

Change Function Declaration

함수 이름 변경하기, 메소드 이름 변경하기, 매개변수 추가하기, 매개변수 제거하기, 시그니처 변경하기

- 좋은 이름을 가진 함수는 함수가 어떻게 구현되었는지 코드를 보지 않아도 이름만 보고도 이해할 수 있다.
- 좋은 이름을 찾아내는 방법? 함수에 주석을 작성한 다음, 주석을 함수 이름으로 만들어 본다.
- 함수의 매개변수는
  - 함수 내부의 문맥을 결정한다. (예, 전화변호 포매팅 함수)
  - 의존성을 결정한다. (예, Payment 만기일 계산 함수)

리팩토링 2. 변수 이름 바꾸기
Rename Variable

- 더 많이 사용되는 변수일수록 그 이름이 더 중요하다.
  - 람다식에서 사용하는 변수 vs 함수의 매개변수
- 다이나믹 타입을 지원하는 언어에서는 타입을 이름에 넣기도 한다.
- 여러 함수에 걸쳐 쓰이는 필드 이름에는 더 많이 고민하고 이름을 짓는다.

리팩토링 3. 필드 이름 바꾸기

Rename Field

- Record 자료 구조의 필드 이름은 프로그램 전반에 걸쳐 참조될 수 있기 때문에 매우 중요하다.
  - Record 자료 구조: 특정 데이터와 관련있는 필드를 묶어놓은 자료 구조
  - 파이썬의 Dictionay, 또는 줄여서 dicts.
  - C#의 Record.
  - 자바 14 버전부터 지원. (record 키워드)
  - 자바에서는 Getter와 Setter 메소드 이름도 필드의 이름과 비슷하게 간주할 수 있다.

냄새 2. 중복 코드
Duplicated Code

- 중복 코드의 단점
  - 비슷한지, 완전히 동일한 코드인지 주의 깊게 봐야한다.
  - 코드를 변경할 때, 동일한 모든 곳의 코드를 변경해야 한다.
- 사용할 수 있는 리팩토링 기술
  - 동일한 코드를 여러 메소드에서 사용하는 경우, 함수 추출하기 (Extract Function)
  - 코드가 비슷하게 생겼지만 완전히 같지는 않은 경우, 코드 분리하기 (Slide Statements)
  - 여러 하위 클래스에 동일한 코드가 있다면, 메소드 올리기 (Pull Up Method)

리팩토링 4. 함수 추출하기

Extract Function

- “의도”와 “구현” 분리하기
- 무슨 일을 하는 코드인지 알아내려고 노력해야 하는 코드라면 해당 코드를 함수로 분리하고 함수 이름으로 “무슨 일을 하는지” 표현할 수 있다.
- 한줄 짜리 메소드도 괜찮은가?
- 거대한 함수 안에 들어있는 주석은 추출한 함수를 찾는데 있어서 좋은 단서가 될 수 있다.

리팩토링 5. 코드 정리하기

Slide Statements

- 관련있는 코드끼리 묶여있어야 코드를 더 쉽게 이해할 수 있다.
- 함수에서 사용할 변수를 상단에 미리 정의하기 보다는, 해당 변수를 사용하는 코드 바로 위에 선언하자.
- 관련있는 코드끼리 묶은 다음, 함수 추출하기 (Extract Function)를 사용해서 더 깔끔하게 분리할 수도 있다.

리팩토링 6. 메소드 올리기

Pull Up Method

- 중복 코드는 당장은 잘 동작하더라도 미래에 버그를 만들어 낼 빌미를 제공한다.
  - 예) A에서 코드를 고치고, B에는 반영하지 않은 경우
- 여러 하위 클래스에 동일한 코드가 있다면, 손쉽게 이 방법을 적용할 수 있다.
- 비슷하지만 일부 값만 다른 경우라면, “함수 매개변수화하기 (Parameterize Function)” 리팩토링을 적용한 이후에, 이 방법을 사용할 수 있다.
- 하위 클래스에 있는 코드가 상위 클래스가 아닌 하위 클래스 기능에 의존하고 있다면, “필드 올리기 (Pull Up Field)”를 적용한 이후에 이 방법을 적용할 수 있다.
- 두 메소드가 비슷한 절차를 따르고 있다면, “템플릿 메소드 패턴 (Template Method Pattern)” 적용을 고려할 수 있다.

냄새 3. 긴 함수
Long Function

- 짧은 함수 vs 긴 함수
  - 함수가 길 수록 더 이해하기 어렵다. vs 짧은 함수는 더 많은 문맥 전환을 필요로 한다.
  - “과거에는” 작은 함수를 사용하는 경우에 더 많은 서브루틴 호출로 인한 오버헤드가 있었다.
  - 작은 함수에 “좋은 이름”을 사용했다면 해당 함수의 코드를 보지 않고도 이해할 수 있다.
  - 어떤 코드에 “주석”을 남기고 싶다면, 주석 대신 함수를 만들고 함수의 이름으로 “의도”를 표현해보자.
- 사용할 수 있는 리팩토링 기술
  - 99%는, “함수 추출하기 (Extract Function)”로 해결할 수 있다.
  - 함수로 분리하면서 해당 함수로 전달해야 할 매개변수가 많아진다면 다음과 같은 리팩토링을 고려해볼 수 있다.
    - 임시 변수를 질의 함수로 바꾸기 (Replace Temp with Query)
    - 매개변수 객체 만들기 (Introduce Parameter Object)
    - 객체 통째로 넘기기 (Preserve Whole Object)
  - “조건문 분해하기 (Decompose Conditional)”를 사용해 조건문을 분리할 수 있다.
  - 같은 조건으로 여러개의 Swtich 문이 있다면, “조건문을 다형성으로 바꾸기 (Replace Conditional with Polymorphism)”을 사용할 수 있다.
  - 반복문 안에서 여러 작업을 하고 있어서 하나의 메소드로 추출하기 어렵다면, “반복문 쪼개기 (Split Loop)”를 적용할 수 있다.

리팩토링 7. 임시 변수를 질의 함수로 바꾸기

Replace Temp with Query

- 변수를 사용하면 반복해서 동일한 식을 계산하는 것을 피할 수 있고, 이름을 사용해 의미를 표현할 수도 있다.
- 긴 함수를 리팩토링할 때, 그러한 임시 변수를 함수로 추출하여 분리한다면 빼낸 함수로 전달해야 할 매개변수를 줄일 수 있다.

리팩토링 8. 매개변수 객체 만들기

Introduce Parameter Object

- 같은 매개변수들이 여러 메소드에 걸쳐 나타난다면 그 매개변수들을 묶은 자료 구조를 만들 수 있다.
- 그렇게 만든 자료구조는:
  - 해당 데이터간의 관계를 보다 명시적으로 나타낼 수 있다.
  - 함수에 전달할 매개변수 개수를 줄일 수 있다.
  - 도메인을 이해하는데 중요한 역할을 하는 클래스로 발전할 수도 있다.

리팩토링 9. 객체 통째로 넘기기

Preserve Whole Object

- 어떤 한 레코드에서 구할 수 있는 여러 값들을 함수에 전달하는 경우, 해당 매개변수를 레코드 하나로 교체할 수 있다.
- 매개변수 목록을 줄일 수 있다. (향후에 추가할지도 모를 매개변수까지도…)
- 이 기술을 적용하기 전에 의존성을 고려해야 한다.
- 어쩌면 해당 메소드의 위치가 적절하지 않을 수도 있다. (기능 편애 “Feature Envy” 냄새에 해당한다.)

리팩토링 10. 함수를 명령으로 바꾸기

Replace Function with Command

- 함수를 독립적인 객체인, Command로 만들어 사용할 수 있다.
- 커맨드 패턴을 적용하면 다음과 같은 장점을 취할 수 있다.
  - 부가적인 기능으로 undo 기능을 만들 수도 있다.
  - 더 복잡한 기능을 구현하는데 필요한 여러 메소드를 추가할 수 있다.
  - 상속이나 템플릿을 활용할 수도 있다.
  - 복잡한 메소드를 여러 메소드나 필드를 활용해 쪼갤 수도 있다.
- 대부분의 경우에 “커맨드” 보다는 “함수”를 사용하지만, 커맨드 말고 다른 방법이 없는 경우에만 사용한다.

리팩토링 11. 조건문 분해하기

Decompose Conditional

- 여러 조건에 따라 달라지는 코드를 작성하다보면 종종 긴 함수가 만들어지는 것을 목격할 수 있다.
- “조건”과 “액션” 모두 “의도”를 표현해야한다.
- 기술적으로는 “함수 추출하기”와 동일한 리팩토링이지만 의도만 다를 뿐이다.

리팩토링 12. 반복문 쪼개기

Split Loop

- 하나의 반복문에서 여러 다른 작업을 하는 코드를 쉽게 찾아볼 수 있다.
- 해당 반복문을 수정할 때 여러 작업을 모두 고려하며 코딩을 해야한다.
- 반복문을 여러개로 쪼개면 보다 쉽게 이해하고 수정할 수 있다.
- 성능 문제를 야기할 수 있지만, “리팩토링”은 “성능 최적화”와 별개의 작업이다. 리팩토링을 마친 이후에 성능 최적화 시도할 수 있다.

리팩토링 13. 조건문을 다형성으로 바꾸기

Replace Conditional with Polymorphism

- 여러 타입에 따라 각기 다른 로직으로 처리해야 하는 경우에 다형성을 적용해서 조건문을 보다 명확하게 분리할 수 있다. (예, 책, 음악, 음식 등…) 반복되는 switch문을 각기 다른 클래스를 만들어 제거할 수 있다.
- 공통으로 사용되는 로직은 상위클래스에 두고 달라지는 부분만 하위클래스에 둠으로써, 달리지는 부분만 강조할 수 있다.
- 모든 조건문을 다형성으로 바꿔야 하는 것은 아니다.

냄새 4. 긴 매개변수 목록
Long Parameter List

- 어떤 함수에 매개변수가 많을수록 함수의 역할을 이해하기 어려워진다.
  - 과연 그 함수는 한가지 일을 하고 있는게 맞는가?
  - 불필요한 매개변수는 없는가?
  - 하나의 레코드로 뭉칠 수 있는 매개변수 목록은 없는가?
- 어떤 매개변수를 다른 매개변수를 통해 알아낼 수 있다면, “매개변수를 질의 함수로 바꾸기 (Replace Parameter with Query)”를 사용할 수 있다.
- 기존 자료구조에서 세부적인 데이터를 가져와서 여러 매개변수로 넘기는 대신, “객체 통째로 넘기기 (Preserve Whole Object)”를 사용할 수 있다.
- 일부 매개변수들이 대부분 같이 넘겨진다면, “매개변수 객체 만들기 (Introduce Parameter Object)”를 적용할 수 있다.
- 매개변수가 플래그로 사용된다면, “플래그 인수 제거하기 (Remove Falg Argument)”를 사용할 수 있다.
- 여러 함수가 일부 매개변수를 공통적으로 사용한다면 “여러 함수를 클래스로 묶기 (Combine Functions into Class)”를 통해 매개변수를 해당 클래스의 필드로 만들고 매서드에 전달해야 할 매개변수 목록을 줄일 수 있다.

리팩토링 14. 매개변수를 질의 함수로 바꾸기

Replace Parameter with Query

- 함수의 매개변수 목록은 함수의 다양성을 대변하며, 짧을수록 이해하기 좋다.
- 어떤 한 매개변수를 다른 매개변수를 통해 알아낼 수 있다면 “중복 매개변수”라 생각할 수 있다.
- 매개변수에 값을 전달하는 것은 “함수를 호출하는 쪽”의 책임이다. 가능하면 함수를 호출하는 쪽의 책임을 줄이고 함수 내부에서 책임지도록 노력한다.
- “임시 변수를 질의 함수로 바꾸기”와 “함수 선언 변경하기”를 통해 이 리팩토링을 적용한다.

리팩토링 15. 플래그 인수 제거하기

Remove Flag Argument

- 플래그는 보통 함수에 매개변수로 전달해서, 함수 내부의 로직을 분기하는데 사용한다.
- 플래그를 사용한 함수는 차이를 파악하기 어렵다.
  - bookConcert(customer, false), bookConcert(customer, true)
  - bookConcert(customer), premiumBookConcert(customer)
- 조건문 분해하기 (Decompose Condition)를 활용할 수 있다.

리팩토링 16. 여러 함수를 클래스로 묶기

Combine Functions into Class

- 비슷한 매개변수 목록을 여러 함수에서 사용하고 있다면 해당 메소드를 모아서 클래스를 만들 수 있다.
- 클래스 내부로 메소드를 옮기고, 데이터를 필드로 만들면 메소드에 전달해야 하는 매개변수 목록도 줄일 수 있다.

냄새 5. 전역 데이터
Global Data

- 전역 데이터 (예, 자바의 public static 변수)
- 전역 데이터는 아무곳에서나 변경될 수 있다는 문제가 있다.
- 어떤 코드로 인해 값이 바뀐 것인지 파악하기 어렵다.
- 클래스 변수 (필드)도 비슷한 문제를 겪을 수 있다.
- “변수 캡슐화하기 (Encapsulate Variable)”를 적용해서 접근을 제어하거나 어디서 사용하는지 파악하기 쉽게 만들 수 있다.
- 파라켈수스의 격언, “약과 독의 차이를 결정하는 것은 사용량일 뿐이다.”

리팩토링 17. 변수 캡슐화하기

Encapsulate Variable

- 메소드는 점진적으로 새로운 메소드로 변경할 수 있으나, 데이터는 한번에 모두 변경해야 한다.
- 데이터 구조를 변경하는 작업을 그보다는 조금 더 수월한 메소드 구조 변경 작업으로 대체할 수 있다.
- 데이터가 사용되는 범위가 클수록 캡슐화를 하는 것이 더 중요해진다.
  - 함수를 사용해서 값을 변경하면 보다 쉽게 검증 로직을 추가하거나 변경에 따르는 후속 작업을 추가하는 것이 편리하다.
- 불변 데이터의 경우에는 이런 리팩토링을 적용할 필요가 없다.

냄새 6. 가변 데이터
Mutable Data

- 데이터를 변경하다보면 예상치 못했던 결과나 해결하기 어려운 버그가 발생하기도 한다.
- 함수형 프로그래밍 언어는 데이터를 변경하지 않고 복사본을 전달한다. 하지만 그밖의 프로그래밍 언어는 데이터 변경을 허용하고 있다. 따라서 변경되는 데이터 사용 시 발생할 수 있는 리스크를 관리할 수 있는 방법을 적용하는 것이 좋다.
- 관련 리팩토링
  - “변수 캡슐화하기 (Encapsulate Variable)”를 적용해 데이터를 변경할 수 있는 메소드를 제한하고 관리할 수 있다.
  - **“변수 쪼개기 (Split Variable)”**을 사용해 여러 데이터를 저장하는 변수를 나눌 수 있다.
  - “코드 정리하기 (Slide Statements)”를 사용해 데이터를 변경하는 코드를 분리하고 피할 수 있다.
  - “함수 추출하기 (Extract Function)”으로 데이터를 변경하는 코드로부터 사이드 이팩트가 없는 코드를 분리할 수 있다.
  - **“질의 함수와 변경 함수 분리하기 (Separate Query from Modifier)”**를 적용해서 클라이언트가 원하는 경우에만 사이드 이팩트가 있는 함수를 호출하도록 API를 개선할 수 있다.
  - 가능하다면 **“세터 제거하기 (Remove Setting Method)”**를 적용한다.
  - 계산해서 알아낼 수 있는 값에는 **“파생 변수를 질의 함수로 바꾸기 (Replace Derived Variable with Query)”**를 적용할 수 있다.
  - 변수가 사용되는 범위를 제한하려면 “여러 함수를 클래스로 묶기 (Combine Functions into Class)”또는 **“여러 함수를 변환 함수로 묶기 (Combine Functions into Transform)”**을 적용할 수 있다.
  - **“참조를 값으로 바꾸기 (Change Reference to Value)”**를 적용해서 데이터 일부를 변경하기 보다는 데이터 전체를 교체할 수 있다.

리팩토링 18. 변수 쪼개기
Split Variable

- 어떤 변수가 여러번 재할당 되어도 적절한 경우
  - 반복문에서 순회하는데 사용하는 변수 또는 인덱스
  - 값을 축적시키는데 사용하는 변수
- 그밖에 경우에 재할당 되는 변수가 있다면 해당 변수는 여러 용도로 사용되는 것이며 변수를 분리해야 더 이해하기 좋은 코드를 만들 수 있다.
  - 변수 하나 당 하나의 책임(Responsibility)을 지도록 만든다.
  - 상수를 활용하자. (자바스크립트의 const, 자바의 final)

리팩토링 19. 질의 함수와 변경 함수 분리하기

Separate Query from Modifier

- “눈에 띌만한” 사이드 이팩트 없이 값을 조회할 수 있는 메소드는 테스트 하기도 쉽고, 메소드를 이동하기도 편하다.
- 명령-조회 분리 (command-query separation) 규칙:
  - 어떤 값을 리턴하는 함수는 사이드 이팩트가 없어야 한다.
- “눈에 띌만한 (observable) 사이드 이팩트”
  - 가령, 캐시는 중요한 객체 상태 변화는 아니다. 따라서 어떤 메소드 호출로 인해, 캐시 데이터를 변경하더라도 분리할 필요는 없다.

리팩토링 20. 세터 제거하기

Remove Setting Method

- 세터를 제공한다는 것은 해당 필드가 변경될 수 있다는 것을 뜻한다.
- 객체 생성시 처음 설정된 값이 변경될 필요가 없다면 해당 값을 설정할 수 있는 생성자를 만들고 세터를 제거해서 변경될 수 있는 가능성을 제거해야 한다.

리팩토링 21. 파생 변수를 질의 함수로 바꾸기

Replace Derived Variable with Query

- 변경할 수 있는 데이터를 최대한 줄이도록 노력해야 한다.
- 계산해서 알아낼 수 있는 변수는 제거할 수 있다.
  - 계산 자체가 데이터의 의미를 잘 표현하는 경우도 있다.
  - 해당 변수가 어디선가 잘못된 값으로 수정될 수 있는 가능성을 제거할 수 있다.
- 계산에 필요한 데이터가 변하지 않는 값이라면, 계산의 결과에 해당하는 데이터 역시 불변 데이터기 때문에 해당 변수는 그대로 유지할 수 있다.

리팩토링 22. 여러 함수를 변환 함수로 묶기

Combine Functions into Transform

- 관련있는 여러 파생 변수를 만들어내는 함수가 여러곳에서 만들어지고 사용된다면 그러한 파생 변수를 “변환 함수 (transform function)”를 통해 한 곳으로 모아둘 수 있다.
- 소스 데이터가 변경될 수 있는 경우에는“여러 함수를 클래스로 묶기 (Combine Functions into Class)”를 사용하는 것이 적절하다.
- 소스 데이터가 변경되지 않는 경우에는 두 가지 방법을 모두 사용할 수 있지만, 변환 함수를 사용해서 불변 데이터의 필드로 생성해 두고 재사용할 수도 있다.

리팩토링 23. 참조를 값으로 바꾸기

Change Reference to Value

- 레퍼런스 (Reference) 객체 vs 값 (Value) 객체
  - https://martinfowler.com/bliki/ValueObject.html
  - “Objects that are equal due to the value of their properties, in this case their x and y coordinates, are called value objects.”
  - 값 객체는 객체가 가진 필드의 값으로 동일성을 확인한다.
  - 값 객체는 변하지 않는다.
  - 어떤 객체의 변경 내역을 다른 곳으로 전파시키고 싶다면 레퍼런스, 아니라면 값 객체를 사용한다.

냄새 7. 뒤엉킨 변경
Divergent Change

- 소프트웨어는 변경에 유연하게(soft) 대처할 수 있어야 한다.
- 어떤 한 모듈이 (함수 또는 클래스가) 여러가지 이유로 다양하게 변경되어야 하는 상황.
  - 예) 새로운 결제 방식을 도입하거나, DB를 변경할 때 동일한 클래스에 여러 메소드를 수정해야 하는 경우.
- 서로 다른 문제는 서로 다른 모듈에서 해결해야 한다.
  - 모듈의 책임이 분리되어 있을수록 해당 문맥을 더 잘 이해할 수 있으며 다른 문제는 신경쓰지 않아도 된다.
- 관련 리팩토링 기술
  - **“단계 쪼개기 (Split Phase)”**를 사용해 서로 다른 문맥의 코드를 분리할 수 있다.
  - **“함수 옮기기 (Move Function)”**를 사용해 적절한 모듈로 함수를 옮길 수 있다.
  - 여러가지 일이 하나의 함수에 모여 있다면 “함수 추출하기 (Extract Function)”를 사용할 수 있다.
  - 모듈이 클래스 단위라면 **“클래스 추출하기 (Extract Class)”**를 사용해 별도의 클래스로 분리할 수 있다.

리팩토링 24. 단계 쪼개기

Split Phase

- 서로 다른 일을 하는 코드를 각기 다른 모듈로 분리한다.
  - 그래야 어떤 것을 변경해야 할 때, 그것과 관련있는 것만 신경쓸 수 있다.
- 여러 일을 하는 함수의 처리 과정을 각기 다른 단계로 구분할 수 있다.
  - 예) 전처리 -> 주요 작업 -> 후처리
  - 예) 컴파일러: 텍스트 읽어오기 -> 실행 가능한 형태로 변경
- 서로 다른 데이터를 사용한다면 단계를 나누는데 있어 중요한 단서가 될 수 있다.
- 중간 데이터(intermediate Data)를 만들어 단계를 구분하고 매개변수를 줄이는데 활용할 수 있다.

리팩토링 25. 함수 옮기기
Move Function

- 모듈화가 잘 된 소프트웨어는 최소한의 지식만으로 프로그램을 변경할 수 있다.
- 관련있는 함수나 필드가 모여있어야 더 쉽게 찾고 이해할 수 있다.
- 하지만 관련있는 함수나 필드가 항상 고정적인 것은 아니기 때문에 때에 따라 옮겨야 할 필요가 있다.
- 함수를 옮겨야 하는 경우
  - 해당 함수가 다른 문맥 (클래스)에 있는 데이터 (필드)를 더 많이 참조하는 경우.
  - 해당 함수를 다른 클라이언트 (클래스)에서도 필요로 하는 경우.
- 함수를 옮겨갈 새로운 문맥 (클래스)이 필요한 경우에는 “여러 함수를 클래스로 묶기 (Combine Functions info Class)” 또는 “클래스 추출하기 (Extract Class)”를 사용한다.
- 함수를 옮길 적당한 위치를 찾기가 어렵다면, 그대로 두어도 괜찮다. 언제든 나중에 옮길 수 있다.

리팩토링 26. 클래스 추출하기

Extract Class

- 클래스가 다루는 책임(Responsibility)이 많아질수록 클래스가 점차 커진다.
- 클래스를 쪼개는 기준
  - 데이터나 메소드 중 일부가 매우 밀접한 관련이 있는 경우
  - 일부 데이터가 대부분 같이 바뀌는 경우
  - 데이터 또는 메소드 중 일부를 삭제한다면 어떻게 될 것인가?
- 하위 클래스를 만들어 책임을 분산 시킬 수도 있다.

냄새 8. 산탄총 수술
Shotgun Surgery

- 어떤 한 변경 사항이 생겼을 때 여러 모듈을 (여러 함수 또는 여러 클래스를) 수정해야 하는 상황.
  - “뒤엉킨 변경” 냄새와 유사하지만 반대의 상황이다.
  - 예) 새로운 결제 방식을 도입하려면 여러 클래스의 코드를 수정해야 한다.
- 변경 사항이 여러곳에 흩어진다면 찾아서 고치기도 어렵고 중요한 변경 사항을 놓칠 수 있는 가능성도 생긴다.
- 관련 리팩토링 기술
  - “함수 옮기기 (Move Function)” 또는 **“필드 옮기기 (Move Field)”**를 사용해서 필요한 변경 내역을 하나의 클래스로 모을 수 있다,
  - 비슷한 데이터를 사용하는 여러 함수가 있다면 “여러 함수를 클래스로 묶기 (Combine Functions into Class)”를 사용할 수 있다.
  - “단계 쪼개기 (Split Phase)”를 사용해 공통으로 사용되는 함수의 결과물들을 하나로 묶을 수 있다.
  - **“함수 인라인 (Inline Function)”**과 **“클래스 인라인 (Inline Class)”**로 흩어진 로직을 한 곳으로 모을 수도 있다.

리팩토링 27. 필드 옮기기
Move Field

- 좋은 데이터 구조를 가지고 있다면, 해당 데이터에 기반한 어떤 행위를 코드로 (메소드나 함수) 옮기는 것도 간편하고 단순해진다.
- 처음에는 타당해 보였던 설계적인 의사 결정도 프로그램이 다루고 있는 도메인과 데이터 구조에 대해 더 많이 익혀나가면서, 틀린 의사 결정으로 바뀌는 경우도 있다.
- 필드를 옮기는 단서:
  - 어떤 데이터를 항상 어떤 레코드와 함께 전달하는 경우.
  - 어떤 레코드를 변경할 때 다른 레코드에 있는 필드를 변경해야 하는 경우.
  - 여러 레코드에 동일한 필드를 수정해야 하는 경우
  - (여기서 언급한 '레코드'는 클래스 또는 객체로 대체할 수도 있음)

리팩토링 28. 함수 인라인

Inline Function

- “함수 추출하기 (Extract Function)”의 반대에 해당하는 리팩토링
  - 함수로 추출하여 함수 이름으로 의도를 표현하는 방법.
- 간혹, 함수 본문이 함수 이름 만큼 또는 그보다 더 잘 의도를 표현하는 경우도 있다.
- 함수 리팩토링이 잘못된 경우에 여러 함수를 인라인하여 커다란 함수를 만든 다음에 다시 함수 추출하기를 시도할 수 있다.
- 단순히 메소드 호출을 감싸는 우회형 (indirection) 메소드라면 인라인으로 없앨 수 있다.
- 상속 구조에서 오버라이딩 하고 있는 메소드는 인라인 할 수 없다. (해당 메소드는 일종의 규약이니까)

리팩토링 29. 클래스 인라인

Inline Class

- “클래스 추출하기 (Extract Class)”의 반대에 해당하는 리팩토링
- 리팩토링을 하는 중에 클래스의 책임을 옮기다보면 클래스의 존재 이유가 빈약해지는 경우가 발생할 수 있다.
- 두개의 클래스를 여러 클래스로 나누는 리팩토링을 하는 경우에, 우선 “클래스 인라인”을 적용해서 두 클래스의 코드를 한 곳으로 모으고 그런 다음에 “클래스 추출하기”를 적용해서 새롭게 분리하는 리팩토링을 적용할 수 있다.

냄새 9. 기능 편애

Feature Envy

- 어떤 모듈에 있는 함수가 다른 모듈에 있는 데이터나 함수를 더 많이 참조하는 경우에 발생한다.
  - 예) 다른 객체의 getter를 여러개 사용하는 메소드
- 관련 리팩토링 기술
  - “함수 옮기기 (Move Function)”를 사용해서 함수를 적절한 위치로 옮긴다.
  - 함수 일부분만 다른 곳의 데이터와 함수를 많이 참조한다면 “함수 추출하기 (Extract Function)”로 함수를 나눈 다음에 함수를 옮길 수 있다.
- 만약에 여러 모듈을 참조하고 있다면? 그 중에서 가장 많은 데이터를 참조하는 곳으로 옮기거나, 함수를 여러개로 쪼개서 각 모듈로 분산 시킬 수도 있다.
- 데이터와 해당 데이터를 참조하는 행동을 같은 곳에 두도록 하자.
- 예외적으로, 데이터와 행동을 분리한 디자인 패턴 (전략 패턴 또는 방문자 패턴)을 적용할 수도 있다.

냄새 10. 데이터 뭉치

Data Clumps

- 항상 뭉쳐 다이는 데이터는 한 곳으로 모아두는 것이 좋다.
  - 여러 클래스에 존재하는 비슷한 필드 목록
  - 여러 함수에 전달하는 매개변수 목록
- 관련 리팩토링 기술
  - “클래스 추출하기 (Extract Class)”를 사용해 여러 필드를 하나의 객체나 클래스로 모을 수 있다.
  - “매개변수 객체 만들기 (Introduce Parameter Object)” 또는 “객체 통째로 넘기기 (Preserve Whole Object)”를 사용해 메소드 매개변수를 개선할 수 있다.

냄새 11. 기본형 집착
Primitive Obsession

- 애플리케이션이 다루고 있는 도메인에 필요한 기본 타입을 만들지 않고 프로그래밍 언어가 제공하는 기본 타입을 사용하는 경우가 많다.
  - 예) 전화번호, 좌표, 돈, 범위, 수량 등
- 기본형으로는 단위 (인치 vs 미터) 또는 표기법을 표현하기 어렵다.
- 관련 리팩토링 기술
  - **“기본형을 객체로 바꾸기 (Replace Primitive with Object)”**
  - **“타입 코드를 서브클래스로 바꾸기 (Replace Type Code with Subclasses)”**
  - **“조건부 로직을 다형성으로 바꾸기 (Replace Conditional with Polymorphism)”**
  - “클래스 추출하기 (Extract Class)”
  - “매개변수 객체 만들기 (Introduce Parameter Object)”

리팩토링 30. 기본형을 객체로 바꾸기

Replace Primitive with Object

- 개발 초기에는 기본형 (숫자 또는 문자열)으로 표현한 데이터가 나중에는 해당 데이터와 관련있는 다양한 기능을 필요로 하는 경우가 발생한다.
  - 예) 문자열로 표현하던 전화번호의 지역 코드가 필요하거나 다양한 포맷을 지원하는 경우.
  - 예) 숫자로 표현하던 온도의 단위 (화씨, 섭씨)를 변환하는 경우.
- 기본형을 사용한 데이터를 감싸 줄 클래스를 만들면, 필요한 기능을 추가할 수 있다.

리팩토링 31. 타입 코드를 서브클래스로 바꾸기

Replace Type Code with Subclasses

- 비슷하지만 다른 것들을 표현해야 하는 경우, 문자열(String), 열거형 (enum), 숫자 (int) 등으로 표현하기도 한다.
  - 예) 주문 타입, “일반 주문”, “빠른 주문”
  - 예) 직원 타입, “엔지니어”, “매니저”, “세일즈”
- 타입을 서브클래스로 바꾸는 계기
  - 조건문을 다형성으로 표현할 수 있을 때, 서브클래스를 만들고 “조건부 로직을 다형성으로 바꾸기”를 적용한다.
  - 특정 타입에만 유효한 필드가 있을 때, 서브클래스를 만들고 “필드 내리기”를 적용한다.

리팩토링 32. 조건부 로직을 다형성으로 바꾸기

Replace Conditional with Polymorphism

- 복잡한 조건식을 상속과 다형성을 사용해 코드를 보다 명확하게 분리할 수 있다.
- swich 문을 사용해서 타입에 따라 각기 다른 로직을 사용하는 코드.
- 기본 동작과 (타입에 따른) 특수한 기능이 섞여있는 경우에 상속 구조를 만들어서 기본 동작을 상위클래스에 두고 특수한 기능을 하위클래스로 옮겨서 각 타입에 따른 “차이점”을 강조할 수 있다.
- 모든 조건문을 다형성으로 옮겨야 하는가? 단순한 조건문은 그대로 두어도 좋다. 오직 복잡한 조건문을 다형성을 활용해 좀 더 나은 코드로 만들 수 있는 경우에만 적용한다. (과용을 조심하자.)

냄새 12. 반복되는 switch 문

Repeated Swtiches

- 예전에는 switch 문이 한번만 등장해도 코드 냄새로 생각하고 다형성 적용을 권장했다.
- 하지만 최근에는 다형성이 꽤 널리 사용되고 있으며, 여러 프로그래밍 언어에서 보다 세련된 형태의 switch 문을 지원하고 있다.
- 따라서 오늘날은“반복해서 등장하는 동일한 switch 문”을 냄새로 여기고 있다.
- 반복해서 동일한 switch 문이 존재할 경우, 새로운 조건을 추가하거나 기존의 조건을 변경할 때모든 switch 문을 찾아서 코드를 고쳐야 할지도 모른다.

냄새 13. 반복문

Loops

- 프로그래밍 언어 초기부터 있었던 반복문은 처음엔 별다른 대안이 없어서 간과했지만 최근 Java와 같은 언어에서 함수형 프로그래밍을 지원하면서 반복문에 비해 더 나은 대안책이 생겼다.
- “반복문을 파이프라인으로 바꾸는 (Replace Loop with Pipeline)” 리팩토링을 적용하면 필터나 맵핑과 같은 파이프라인 기능을 사용해 보다 빠르게 어떤 작업을 하는지 파악할 수 있다.

리팩토링 33. 반복문을 파이프라인으로 바꾸기

Replace Loop with Pipeline

- 콜렉션 파이프라인 (자바의 Stream, C#의 LINQ - Language Integrated Query)
- 고전적인 반복문을 파이프라인 오퍼레이션을 사용해 표현하면 코드를 더 명확하게 만들 수 있다.
  - 필터 (filter): 전달받은 조건의 true에 해당하는 데이터만 다음 오퍼레이션으로 전달.
  - 맵 (map): 전달받은 함수를 사용해 입력값을 원하는 출력값으로 변환하여 다음 으퍼레이션으로 전달.
- https://martinfowler.com/articles/refactoring-pipelines.html

냄새 14. 성의없는 요소

Lazy Element

- 여러 프로그래밍적인 요소(변수, 메소드, 클래스 등)를 만드는 이유
  - 나중에 발생할 변화를 대비해서…
  - 해당 함수 또는 클래스를 재사용하려고…
  - 의미있는 이름을 지어주려고…
- 가끔 그렇게 예상하고 만들어 놓은 요소들이 기대에 부응하지 못하는 경우가 있는데 그런 경우에 해당 요소들을 제거해야 한다.
- 관련 리팩토링 기술
  - “함수 인라인 (Inline Function)”
  - “클래스 인라인 (Inline Class)”
  - 불필요한 상속 구조는 **“계층 합치기 (Collapse Hierarchy)”**를 사용할 수 있다.

리팩토링 34. 계층 합치기

Collapse Hierarchy

- 상속 구조를 리팩토링하는 중에 기능을 올리고 내리다 보면 하위클래스와 상위클래스 코드에 차이가 없는 경우가 발생할 수 있다. 그런 경우에 그 둘을 합칠 수 있다.
- 하위클래스와 상위클래스 중에 어떤 것을 없애야 하는가? (둘 중에 보다 이름이 적절한 쪽을 선택하지만, 애매하다면 어느쪽을 선택해도 문제없다.)

냄새 15. 추측성 일반화

Speculative Generality

- 나중에 이러 저러한 기능이 생길 것으로 예상하여, 여러 경우에 필요로 할만한 기능을 만들어 놨지만 “그런 일은 없었고…”결국에 쓰이지 않는 코드가 발생한 경우.
- XP의 YAGNI (You aren’t gonna need it) 원칙을 따르자.
- 관련 리팩토링
  - 추상 클래스를 만들었지만 크게 유요하지 않다면 “계층 합치기 (Collapse Hierarchy)”
  - 불필요한 위임은 “함수 인라인 (Inline Function)” 또는 “클래스 인라인 (Inline Class)”
  - 사용하지 않는 매개변수를 가진 함수는 “함수 선언 변경하기 (Change Function Declaration)”
  - 오로지 테스트 코드에서만 사용하고 있는 코드는 “**죽은 코드 제거하기 (Remove Dead Code)**”

리팩토링 35. 죽은 코드 제거하기

Remove Dead Code

- 사용하지 않는 코드가 애플리케이션 성능이나 기능에 영향을 끼치지는 않는다.
- 하지만, 해당 소프트웨어가 어떻게 동작하는지 이해하려는 사람들에게는 꽤 고통을 줄 수 있다.
- 실제로 나중에 필요해질 코드라 하더라도 지금 쓰이지 않는 코드라면 (주석으로 감싸는게 아니라) 삭제해야 한다.
    - 나중에 정말로 다시 필요해진다면 git과 같은 버전 관리 시스템을 사용해 복원할 수 있다.

냄새 16. 임시 필드

Temporary Field

- 클래스에 있는 어떤 필드가 특정한 경우에만 값을 갖는 경우.
- 어떤 객체의 필드가 “특정한 경우에만” 값을 가진다는 것을 이해하는 것은 일반적으로 예상하지 못하기 때문에 이해하기 어렵다.
- 관련 리팩토링
  - “클래스 추출하기 (Extract Class)”를 사용해 해당 변수들을 옮길 수 있다.
  - “함수 옮기기 (Move Function)”을 사용해서 해당 변수를 사용하는 함수를 특정 클래스로 옮길 수 있다.
  - **“특이 케이스 추가하기 (Introduce Special Case)”**를 적용해 “특정한 경우”에 해당하는 클래스를 만들어 해당 조건을 제거할 수 있다.

리팩토링 36. 특이 케이스 추가하기

Introduce Special Case

- 어떤 필드의 특정한 값에 따라 동일하게 동작하는 코드가 반복적으로 나타난다면, 해당 필드를 감싸는 “특별한 케이스”를 만들어 해당 조건을 표현할 수 있다.
- 이러한 매커니즘을 “특이 케이스 패턴”이라고 부르고 “Null Object 패턴”을 이러한 패턴의 특수한 형태라고 볼 수 있다.

냄새 17. 메시지 체인
Message Chains

- 레퍼런스를 따라 계속해서 메소드 호출이 이어지는 코드.
  - 예) this.member.getCredit().getLevel().getDescription()
- 해당 코드의 클라이언트가 코드 체인을 모두 이해해야 한다.
- 체인 중 일부가 변경된다면 클라이언트의 코드도 변경해야 한다.
- 관련 리팩토링
  - “**위임 숨기기 (Hide Delegate)**”를 사용해 메시지 체인을 캡슐화를 할 수 있다.
  - “함수 추출하기 (Extract Function)”로 메시지 체인 일부를 함수로 추출한 뒤, “함수 옮기기 (Move Function)”으로 해당 함수를 적절한 이동할 수 있다.

리팩토링 37. 위임 숨기기

Hide Delegate

- 캡슐화 (Encapsulation)란 어떤 모듈이 시스템의 다른 모듈을 최소한으로 알아야 한다는 것이다. 그래야 어떤 모듈을 변경할 때, 최소한의 모듈만 그 변경에 영향을 받을 것이고, 그래야 무언가를 변경하기 쉽다.
- 처음 객체 지향에서 캡슐화를 배울 때 필드를 메소드로 숨기는 것이라 배우지만, 메소드 호출도 숨길 수 있다.
  - person.department().manager(); -> person.getManager()
  - 이전의 코드는 Department를 통해 Manager에 접근할 수 있다는 정보를 알아야 하지만, getManager()를 통해 위임을 숨긴다면 클라이언트는 person의 getManager()만 알아도 된다. 나중에 getManager() 내부 구현이 바뀌더라도 getManager()를 사용한 코드는 그대로 유지할 수 있다.

냄새 18. 중재자

Middle Man

- 캡슐화를 통해 내부의 구체적인 정보를 최대한 감출 수 있다.
- 그러나, 어떤 클래스의 메소드가 대부분 다른 클래스로 메소드 호출을 위임하고 있다면 중재자를 제거하고 클라이언트가 해당 클래스를 직접 사용하도록 코드를 개선할 수 있다.
- 관련 리팩토링
  - **“중재자 제거하기 (Remove Middle Man)”** 리팩토링을 사용해 클라이언트가 필요한 클래스를 직접 사용하도록 개선할 수 있다.
  - “함수 인라인 (Inlince Function)”을 사용해서 메소드 호출한 쪽으로 코드를 보내서 중재자를 없앨 수도 있다.
  - **“슈퍼클래스를 위임으로 바꾸기 (Replace Superclass with Delegate)”**
  - **“서브클래스를 위임으로 바꾸기 (Replace Subclass with Delegate)**

리팩토링 38. 중재자 제거하기

Remove Middle Man

- “위임 숨기기”의 반대에 해당하는 리팩토링.
- 필요한 캡슐화의 정도는 시간에 따라 그리고 상황에 따라 바뀔 수 있다.
- 캡슐화의 정도를 “중재자 제거하기”와 “위임 숨기기” 리팩토링을 통해 조절할 수 있으다.
- 위임하고 있는 객체를 클라이언트가 사용할 수 있도록 getter를 제공하고, 클라이언트는 메시지 체인을 사용하도록 코드를 고친 뒤에 캡슐화에 사용했던 메소드를 제거한다.
- Law of Demeter를 지나치게 따르기 보다는 상황에 맞게 활용하도록 하자.
  - 디미터의 법칙, “가장 가까운 객체만 사용한다.”

리팩토링 39. 슈퍼클래스를 위임으로 바꾸기

Replace Superclass with Delegate

- 객체지향에서 “상속”은 기존의 기능을 재사용하는 쉬우면서 강력한 방법이지만 때로는 적절하지 않은 경우도 있다.
- 서브클래스는 슈퍼클래스의 모든 기능을 지원해야 한다.
  - Stack이라는 자료구조를 만들 때 List를 상속 받는것이 좋을까?
- 서브클래스는 슈퍼클래스 자리를 대체하더라도 잘 동작해야 한다.
  - 리스코프 치환 원칙
- 서브클래스는 슈퍼클래스의 변경에 취약하다.
- 그렇다면 상속을 사용하지 않는 것이 좋은가?
  - 상속은 적절한 경우에 사용한다면 매우 쉽고 효율적인 방법이다.
  - 따라서, 우선 상속을 적용한 이후에, 적절치 않다고 판단이 된다면 그때에 이 리팩토링을 적용하자.

리팩토링 40. 서브클래스를 위임으로 바꾸기

Replace Subclass with Delegate

- 어떤 객체의 행동이 카테고리에 따라 바뀐다면, 보통 상속을 사용해서 일반적인 로직은 슈퍼클래스에 두고 특이한 케이스에 해당하는 로직을 서브클래스를 사용해 표현한다.
- 하지만, 대부분의 프로그래밍 언어에서 상속은 오직 한번만 사용할 수 있다.
  - 만약에 어떤 객체를 두가지 이상의 카테고리로 구분해야 한다면?
  - **위임을 사용하면 얼마든지 여러가지 이유로 여러 다른 객체로 위임을 할 수 있다.**
- 슈퍼클래스가 바뀌면 모든 서브클래스에 영향을 줄 수 있다. 따라서 슈퍼클래스를 변경할 때 서브클래스까지 신경써야 한다.
  - 만약에 서브클래스가 전혀 다른 모듈에 있다면?
  - **위임을 사용한다면 중간에 인터페이스를 만들어 의존성을 줄일 수 있다.**
- “상속 대신 위임을 선호하라.”는 결코 “상속은 나쁘다.”라는 말이 아니다.
  - 처음엔 상속을 적용하고 언제든지 이런 리팩토링을 사용해 위임으로 전환할 수 있다.

냄새 19. 내부자 거래

Insider Trading

- 어떤 모듈이 다른 모듈의 내부 정보를 지나치게 많이 알고 있는 코드 냄새. 그로인해 지나치게 강한 결합도(coupling)가 생길 수 있다.
- 적절한 모듈로 “함수 옮기기 (Move Function)”와 “필드 옮기기 (Move Field)”를 사용해서 결합도를 낮출 수 있다.
- 여러 모듈이 자주 사용하는 공통적인 기능은 새로운 모듈을 만들어 잘 관리하거나, “위임 숨기기 (Hide Delegate)”를 사용해 특정 모듈의 중재자처럼 사용할 수도 있다.
- 상속으로 인한 결합도를 줄일 때는 “슈퍼클래스 또는 서브클래스를 위임으로 교체하기”를 사용할 수 있다.

냄새 20. 거대한 클래스

Large Class

- 어떤 클래스가 너무 많은 일을 하다보면 필드도 많아지고 중복 코드도 보이기 시작한다.
- 클라이언트가 해당 클래스가 제공하는 기능 중에 일부만 사용한다면 각각의 세부 기능을 별도의 클래스로 분리할 수 있다.
  - “클래스 추출하기 (Extract Class)”를 사용해 관련있는 필드를 한 곳으로 모을 수 있다.
  - 상속 구조를 만들 수 있다면 **“슈퍼클래스 추출하기 (Extract Superclass)”**또는 “타입 코드를 서브클래스로 교체하기”를 적용할 수 있다.
- 클래스 내부에 산재하는 중복 코드는 메소드를 추출하여 제거할 수 있다.

리팩토링 41. 슈퍼클래스 추출하기

Extract Superclass

- 두개의 클래스에서 비슷한 것들이 보인다면 상속을 적용하고, 슈퍼클래스로 “필드 올리기 (Pull Up Field)”와 “메소드 올리기 (Pull Up Method)”를 사용한다.
- 대안으로는 “클래스 추출하기 (Extract Class)”를 적용해 위임을 사용할 수 있다.
- 우선은 간단히 상속을 적용한 이후, 나중에 필요하다면 “슈퍼클래스를 위임으로 교체하기”를 적용한다.

냄새 21. 서로 다른 인터페이스의 대안 클래스들

Alternative Classes with Different Interfaces

- 비슷한 일을 여러 곳에서 서로 다른 규약을 사용해 지원하고 있는 코드 냄새.
- 대안 클래스로 사용하려면 동일한 인터페이스를 구현하고 있어야 한다.
- “함수 선언 변경하기 (Change Function Declaration)”와 “함수 옮기기 (Move Function)을 사용해서 서로 동일한 인터페이스를 구현하게끔 코드를 수정할 수 있다.
- 두 클래스에서 일부 코드가 중복되는 경우에는“슈퍼클래스 추출하기 (Extract Superclass)”를 사용해 중복된 코드를 슈퍼클래스로 옮기고 두 클래스를 새로운 슈퍼클래스의 서브클래스로 만들 수 있다.

냄새 22. 데이터 클래스
Data Class

- 데이터 클래스: public 필드 또는 필드에 대한 게터와 세터만 있는 클래스.
  - 코드가 적절한 위치에 있지 않기 때문에 이러한 냄새가 생길 수 있다.
  - 예외적으로“단계 쪼개기”에서 중간 데이터를 표현하는데 사용할 레코드는 불변 객체로 데이터를 전달하는 용도로 사용할 수 있다.
- public 필드를 가지고 있다면 **“레코드 캡슐화하기 (Encapsulate Record)”**를 사용해 게터나 세터를 통해서 접근하도록 고칠 수 있다.
- 변경되지 않아야 할 필드에는“세터 제거하기 (Remove Setting Method)”를 적용할 수 있다.
- 게터와 세터가 사용되는 메소드를 찾아보고 “함수 옮기기 (Move Function)”을 사용해서 데이터 클래스로 옮길 수 있다.
- 메소드 전체가 아니라 일부 코드만 옮겨야 한다면 “함수 추출하기 (Extract Function)”을 선행한 뒤에 옮길 수 있다.

리팩토링 42. 레코드 캡슐화하기

Encapsulate Record

- 변하는 데이터를 다룰 때는 레코드 보다는 객체를 선호한다.
  - 여기서“레코드”란, public 필드로 구성된 데이터 클래스를 말함.
  - 데이터를 메소드 뒤로 감추면 객체의 클라이언트는 어떤 데이터가 저장되어 있는지 신경쓸 필요가 없다.
  - 필드 이름을 변경할 때 점진적으로 변경할 수 있다.
  - 하지만 자바의 Record는 불변 객체라서 이런 리팩토링이 필요없다.
- public 필드를 사용하는 코드를 private 필드와 게터, 세터를 사용하도록 변경한다.

냄새 23. 상속 포기

Refused Bequest

- 서브클래스가 슈퍼클래스에서 제공하는 메소드나 데이터를 잘 활용하지 않는다는 것은 해당 상속 구조에 문제가 있다는 뜻이다.
  - 기존의 서브클래스 또는 새로운 서브클래스를 만들고 슈퍼클래스에서 “메소드와 필드를 내려주면 (Push Down Method / Field)” 슈퍼클래스에 공동으로 사용하는 기능만 남길 수 있다.
- 서브클래스가 슈퍼클래스의 기능을 재사용하고 싶지만 인터페이스를 따르고 싶지 않은 경우에는 “슈퍼클래스 또는 서브클래스를 위임으로 교체하기” 리팩토링을 적용할 수 있다.

냄새 24. 주석

Comments

- 주석을 남겨야 할 것 같다면 먼저 코드를 리팩토링하라. 불필요한 주석을 줄일 수 있다.
  - 모든 주석이 나쁘다는 것도 아니고, 주석을 쓰지 말자는 것도 아니다.
  - 주석은 좋은 냄새에 해당하기도 한다.
- 관련 리팩토링
  - “함수 추출하기”를 사용해 설명이 필요한 부분을 별도의 메소드로 빼낸다.
  - “함수 선언부 변경하기”를 사용해 함수 이름을 재정의할 수 있다.
  - 시스템적으로 어떤 필요한 규칙이 있다면, “어서션 추가하기 (Introduce Assertion)”을 적용할 수 있다.

리팩토링 43. 어서션 추가하기

Introduce Assertion

- 종종 코드로 표현하지 않았지만 기본적으로 가정하고 있는 조건들이 있다. 그런 조건을 알고리듬을 파악하거나 주석을 읽으면서 확인할 수 있다.
- 그러한 조건을 Assertion을 사용해서 보다 명시적으로 나타낼 수 있다.
- Assertion은 if나 swtich 문과 달리 “항상” true이길 기대하는 조건을 표현할 때 사용한다.
  - 프로그램이 Assertion에서 실패한다면 프로그래머의 실수로 생각할 수 있다.
  - Assertion이 없어도 프로그램이 동작해야 한다. (자바에서는 컴파일 옵션으로 assert 문을 사용하지 않도록 설정할 수도 있다.)
- 특정 부분에선 특정한 상태를 가정하고 있다는 것을 명시적으로 나타냄로써, 의사소통적인 가치를 지니고 있다.

카탈로그 1. 기본 기술

가장 자주 사용하는 리팩토링 기술

- 함수 추출하기 (Extract Function)
- 함수 인라인하기 (Inline Function)
- **변수 추출하기 (Extract Variable)**
- **변수 인라인하기 (Inline Variable)**
- 함수 선언 변경하기 (Change Function Declaration)
- 변수 캡슐화하기 (Encapsulate Vraiable)
- **변수 이름 바꾸기 (Rename Variable)**
- 매개변수 객체 만들기 (Introduce Parameter Object)
- 여러 함수를 클래스로 묶기 (Combine Functions into Class)
- 여러 함수를 변환 함수로 묶기 (Combine Functions into Trasnform)
- 단계 쪼개기 (Split Phase)

카탈로그 2. 캡슐화

모듈에서 외부 시스템으로 공개하지 않아도 되는 데이터를 숨기는 기술

- 레코드 캡슐화하기 (Encapsulate Record)
- **컬렉션 캡슐화하기 (Encapsulate Collection)**
- 기본형을 객체로 바꾸기 (Replace Prinitive with Object)
- 임시 변수를 질의 함수로 바꾸기 (Replace Temp with Query)
- 클래스 추출하기 (Extract Class)
- 클래스 인라인하기 (Inline Class)
- 위임 숨기기 (Hide Delegate)
- 중재자 제거하기 (Remove Middle Man)
- **알고리듬 교체하기 (Substitute Algorithm)**

카탈로그 3. 기능 옮기기

함수나 필드 또는 문장을 적절한 위치로 옮기는 기술

- 함수 옮기기 (Move Function)
- 필드 옮기기 (Move Field)
- **문장을 함수로 옮기기 (Move Statements into Function)**
- **문장을 호출한 곳으로 옮기기 (Move Statements to Callers)**
- **인라인 코드를 함수 호출로 바꾸기 (Replace Inline Code with Function Call)**
- 문장 슬라이드하기 (Slide Statements)
- 반복문 쪼개기 (Split Loop)
- 반복문을 파이프라인으로 바꾸기 (Replace Loop with Pipeline)
- 죽은 코드 제거하기 (Remove Dead Code)

카탈로그 4. 데이터 조직화

데이터 구조를 다루는 기술

- 변수 쪼개기 (Split Variable)
- 필드 이름 바꾸기 (Rename Field)
- 파생 변수를 질의 함수로 바꾸기 (Replace Derived Variable with Query)
- **참조를 값으로 바꾸기 (Change References to Value)**
- **값을 참조로 바꾸기 (Change Value to Reference)**

카탈로그 5. 조건부 로직 간소화

복잡한 조건문을 다루는 기술

- 조건문 분해하기 (Decompose Conditional)
- **조건식 통합하기 (Consolidate Conditional Expression)**
- **중첩 조건문을 보호 구문으로 바꾸기 (Replace Nested Conditional with Guard Clauses)**
- 조건부 로직을 다형성으로 바꾸기 (Replace Conditional with Polymorphism)
- 특이 케이스 추가하기 (Introduce Special Case)
- 어서션 추가하기 (Introduce Assertion)

카탈로그 6. API 리팩토링

쉽고 이해하고 사용할 수 있는 API를 만드는 기술

- 질의 함수와 변경 함수 분리하기 (Separate Query from Modifier)
- **함수 매개변수화하기 (Parameterize Function)**
- 플래그 인수 제거하기 (Remove Flag Argument)
- 객체 통째로 넘기기 (Preserve Whole Object)
- 매개변수를 질의 함수로 바꾸기 (Replace Parameter with Query)
- **질의 함수를 매개변수로 바꾸기 (Replace Query with Parameter)**
- 세터 제거하기 (Remove Setting Method)
- **생성자를 팩토리 함수로 바꾸기 (Replace Constructor with Factory Function)**
- 함수를 명령으로 바꾸기 (Replace Function with Command)
- **명령을 함수로 바꾸기 (Replace Command with Function)**

카탈로그 7. 상속 다루기

상속을 제대로 사용하는 기술

- 메소드 올리기 (Pull Up Method)
- **필드 올리기 (Pull Up Field)**
- **생성자 본문 올리기 (Pull Up Constructor Body)**
- 메서드 내리기 (Push Down Method)
- 필드 내리기 (Push Down Field)
- 타입 코드를 서브클래스로 바꾸기 (Replace Type Code with Subclasses)
- **서브클래스 제거하기 (Remove Subclass)**
- 슈퍼클래스 추출하기 (Extract Superclass)
- 계층 합치기 (Collapse Hierarchy)
- 서브클래스를 위임으로 바꾸기 (Replace Subclass with Delegate)
- 슈퍼클래스를 위임으로 바꾸기 (Replace Superclass with Delegate)

리팩토링 참고 자료

- 서적
  - 리팩터링 2판 (리팩토링 개정판)
  - 패턴을 활용한 리팩터링
  - GoF의 디자인 패턴
- 웹 사이트
  - https://refactoring.com/catalog/
  - https://wiki.c2.com/?WikiPagesAboutRefactoring

감사합니다.

백기선 올림