Refactoring

학습 목표

- 리팩토링 개념과 코드 스멜(Code Smell)의 개념을 이해한다
- 코드 스멜을 찾고 리팩토링 기법들을 연습한다
- 연습한 리팩토링 기법들을, 코드의 품질 개선 및 코드 리뷰에 적용할 수 있다

목차

리팩토링 개요

리팩토링 정의

"소프트웨어의 겉보기 동작은 그대로 유지한 채,
코드를 이해하고 수정하기 쉽도록
내부 구조를 변경하는 기법 혹은
리팩토링 기법을 적용해서 소프트웨어를 재구성하는 것"

- Martin Fowler

- 단순한 코드정리 ≠ 리팩토링 (정리방법의 체계)
- S/W 재구성(Restructuring)의 특수한 한 형태
- 가독성과 유지보수성은 좋게 하면서 동작을 보존하는 작은 단계들
- 작은 단계들이 순차적으로 연결되어 큰 변화를 만들어 내는 것

리팩토링 모드 : 켄트 벡의 Two Hats

- 코드 변경의 **두 가지** 주요 목적
  - 기능 구현과 리팩토링

- 변경 목적에 따른 작업 방식 차이
  - 기능 구현 : 기능의 요구 사항에 집중
  - 리팩토링 : 코드의 가독성, 유지 보수성에 집중(기능추가-X)

- 목적에 부합하는 SW 개발 방식 선택

  리팩토링 모자와 **기능 구현 모자**를 바꿔 쓰듯이 필요에 따라,
  자신이 쓰고 있는 모자가 무엇인지를 알고
  그에 따른 **작업 방식의 차이**를 분명하게 인지해야 한다.

리팩토링의 목적

- S/W 설계의 개선
- S/W의 가독성의 개선
  - 내부 품질의 개선
- 버그를 쉽게 찾게 도와줌
- **프로그래밍 속도를 높임**
  - 개발 기간의 단축

리팩토링을 해야 할 때 (1/2)

- **언제나, 모든 코드**에 대해 리팩토링을 해야 할까 ?

- The Rule of Three : 3 Strike 면 리팩토링을 하자!

  1. **처음에는 리팩토링 없이,** 그냥 한다
  2. 비슷한 일을 두 번째로 하게 되면, 또 일단 그냥 한다
  3. 비슷한 일을 **세 번째** 하게 된다면, 그때 리팩토링한다

* (Don Roberts가 Martin Fowler에게 제시한 가이드)

리팩토링을 해야 할 때 (2/2)

- 준비과정에서의 리팩토링 (Preparatory Refactoring)
  - 코드 베이스에 기능을 새로 추가하기 직전에 수행
- 이해를 위한 리팩토링 (Comprehension Refactoring)
  - 코드 변경 이전에 코드를 이해하기 쉽도록 코드 정리
- 쓰레기 줍기 리팩토링 (Litter-Pickup Refactoring)
  - 비효율적으로 기능을 수행하는 코드
- 계획된 리팩토링 (Planned Refactoring)
- 오래 걸리는 리팩토링 (Long-Term Refactoring)
  - 전체 라이브러리 교체와 같은 대규모 리팩토링
  - 리팩토링 중에도 SW 가 정상 동작하도록 유념
- 코드 리뷰 (Code Review)
  - 리팩토링을 통해, 한 차원 높은 아이디어나, 리뷰 결과를 더 구체적으로 도출(pair programming)

리팩토링을 하지 말아야 할 때

- 수정 및 이해할 필요가 **없는** 코드
  - 더 이상 refactor 할 필요가 없는 수명이 다한 경우
  - refactor하는 게 어떤 이점도 없는 경우
  - 리팩토링은 수정되어야 하는 코드, 읽어야 하는 코드에 집중

- 리팩토링을 **포기**하는 코드
  - refactor하는 것보다 새로 작성하는 게 더 쉬운 경우
  - ex.- Low Code Quality

리팩토링 고려 사항 (1/2)

- 새 기능의 개발 속도 저하
  - 리팩토링 수행 여부의 판단 기준은 **경제적 효과**
  - 리팩토링에 대한 **긍정적**인 인식의 제고, **과도한** 리팩토링의 지양
  - 리팩토링 보다, 기능 개발이 **우선**

- Code의 Ownership
  - Ownership이 없는 경우, 리팩토링한 코드를 코드 베이스에 반영하기 어려움
  - Ownership을 큰 단위로 정하여(ex. 개인->부서), 반영을 쉽게 해 줌 (혹은 오픈소스 개발모델에서의 방식을 활용)

- Branch
  - 독립 브랜치로 작업하는 시간이 길어질수록, Master 브랜치로의 Merge 작업은 상당히 복잡
  - Continuous Integration

리팩토링 고려 사항 (2/2)

- Testing
  - 리팩토링 후에도 외부 동작이 변경되지 않았음을 확인 (Protection of Business Code)
  - 수시로, 빠르게 수행 (Self Test Code*)

- Legacy Code
  - 대부분은 Test Code가 없다는 점에서, 리팩토링이 어려움
  - 추가할 틈새를 찾아 Test Code를 보강, 이때 테스트가 쉽도록 리팩토링을 수행
  - 한번에 legacy code 전체를 refactor 하기보단, 조금씩 개선

- Database
  - 데이터구조 변경과 연관된 Business Code 부분

    Database스키마에 대한 구조적 변경과 이에 따른 Data Migration 부분
  - Evolutionary Database Design, Database 리팩토링 기법 적용
    관련도서) 리팩토링 데이터베이스, Scott Ambler, Pramod Sadalage

(* Self Test Code -스스로 성공/실패를 판단하는 테스트)

리팩토링 전략

- A series of small changes
  - 기존 코드를 **조금씩** 개선 (작은 변경)
  - **일련**의 작은 변경(단계)들을 수행함으로써 진행

- slightly better, still leaving in working order
  - 리팩토링 후에도
    코드는 **정상 작동**되어야 하고,
    겉보기 동작은 **그대로 유지** (리팩토링 Hat)

- 일련의 전체 리팩토링 작업이 끝나지 않아도, 언제든 **멈출 수** 있어야 함

리팩토링과 테스트

* 그림 출처) https://www.youtube.com/watch?v=J6yyvGWFCkY,

* 동영상링크) https://youtu.be/_v9eqp_ZzL4

- 개발자의 Test Code vs. 작업자의 **안전장치**

리팩토링 절차

- Step1.
  - 분석 중인 코드의 일부분에 대해,
    기존의 동작을 확인,증명해주는 견고한 Test Code들을 준비한다

- Step2.
  - 코드 스멜을 활용하여, 코드에서 문제를 찾는다. : 잠재적인 문제 찾기

- Step3.
  - 리팩토링 기법을 적용하여 문제를 해결한다
  - 테스트한다 : 리팩토링을 제대로 했음을 확인

리팩토링 Trade-Off

- 좋은 코드의 기준은 도메인, 팀, 개인의 가치에 따라 상이

  ex) 성능 우선의 Application

- 리팩토링 기법들 또한 상충적 관계 존재

  ex) 변수 추출하기 <-> 변수 인라인 하기

  위임 숨기기 <-> 중개자 제거하기

- 각자의 가치와 리팩토링 기법의 장단점을 고려하여,

  투자 대비 효과 측면에서

  자신의 가치에 부합하는 **최적**의 방안을 결정해야 한다.

  어떻게 리팩토링할지에 대해서 정답이 있는 것이 아니다.

코드 스멜

코드 스멜의 정의

"Any characteristic
in the source code of a program
that **possibly indicates a deeper problem.**"

- https://en.wikipedia.org/wiki/Code_smell

- 리팩토링을 언제 시작할지,
  어떤 리팩토링 기법을 사용할지를 알려주는, Heuristic 한 도구
  (특성, 징후, 숙련된 개발자들의 직관)

- 리팩토링의 유발자 역할

- 가독성과 유지 보수성(리팩토링의 목적)을 저해하는 요인에 집중

코드 스멜(Code Smells)

- 기이한 이름 (Mysterious Name)
- 중복 코드 (Duplication Code)
- 긴 함수 (Long Function)
- 긴 매개변수 목록 (Long Parameter List)
- 전역 데이터 (Global Data)
- 가변 데이터 (Mutable Data)
- 뒤엉킨 변경 (Divergent Change)
- 산탄총 수술 (Shotgun Surgery)
- 기능 편애 (Feature Envy)
- 데이터 뭉치 (Data Clumps)
- 기본형 집착 (Primitive Obsession)
- 반복되는 Switch문 (Repeated Switches)
- 반복문 (Loops)
- 성의 없는 요소 (Lazy Element)
- 추측성 일반화 (Speculative Generality)
- 임시 필드 (Temporary Field)
- 메시지 체인 (Message Chains)
- 중개자 (Middle Man)
- 내부자 거래 (Insider Trading)
- 거대한 클래스 (Large Class)
- 다른 인터페이스의 대안 클래스들
  (Alternative Classes with Different Interfaces)
- 데이터 클래스 (Data Class)
- 상속 포기 (Refused Bequest)
- 주석 (Comments)

기이한 이름 (Mysterious Name)

Symptoms

코드만으로는 수행하는 동작을 알 수 없어, 이해하기 위해

Text를 이리저리 맞추어 보아야 하는 경우

Treatment

- 함수 이름을 바꾸기 위한, **함수 선언 바꾸기(Change Function Declaration)**
- 변수 이름을 바꾸기 위한, **변수 이름 바꾸기(Rename Variable)**
- 필드 이름을 바꾸기 위한, **필드 이름 바꾸기(Rename Field)**

※ 혼란스러운 이름을 정리하는 것만으로도 코드는 훨씬 간결해짐

마땅한 이름이 떠오르지 않는다면, 더 근본적인 문제가 있을 가능성이 크다

중복 코드 (Duplication Code)

Symptoms

동일한 코드 구조가 두 군데 이상 있을 때.

Treatment

- 동일한 코드가 한 클래스의 두 함수 안에 존재하는 경우,
  함수 추출하기(Extract Function)
- 비슷한 코드가 여러 군데 존재하는 경우, 문장 슬라이드하기(Slide Statement) 한 후,
  함수 추출하기(Extract Function)
- 중복된 코드가 서브 클래스들에 존재하는 경우,
  메서드 올리기(Pull Up Method) (슈퍼 클래스로 옮기기)

긴 함수 (Long Function) (1/2)

Symptoms

함수의 코드가 긴 경우.

10라인이 넘어가면 의문을 가져보아야 한다. 20줄도 길다.

- R.C Martin (Clean Code)

Treatment (1/2)

- 코드의 길이를 줄이기 위해, **함수 추출하기(Extract Function)**
- 추출한 함수의 **매개변수가 많아질** 경우,
  - **임시 변수를 질의 함수로 바꾸기(Replace Temp with Query)**
    : 함수 추출하기에 앞서, 로컬 변수 제거
  - **매개변수 객체 만들기(Introduce Parameter Object)** 와 **객체 통째로 넘기기(Preserve Whole Object)**
    : 객체를 만들어 매개변수로 전달
- 위의 내용을 적용해도 여전히 매개변수가 너무 많은 경우,
  함수를 명령으로 바꾸기(Replace Function with Command)
  : method 를 분리된 객체로 옮기기

긴 함수 (Long Function) (2/2)

Treatment (2/2)

(조건문과 반복문도 긴 함수 스멜이 많이 검출되는 부분)

- **조건문**은 조건문 분해하기(Decompose Conditional)
- 거대한 switch구문의 경우, case절에 대해, 함수 추출하기(Extract Functions)
- 같은 조건을 기준으로 나누는 switch구문이 여럿인 경우,
  조건부 로직을 다형성으로 바꾸기(Replace Conditional with Polymorphism)

- **반복문**은 함수 추출하기(Extract Function) (추출한 함수에 대해 적절한 이름을 붙임)
- 반복문이 두 가지 이상의 작업을 하는 경우에는, *추출하기에 앞서* 반복문 쪼개기(split Loop)

긴 매개변수 목록 (Long Parameter List)

Symptoms

3~4개 이상의 매개변수가 전달되는 경우.

Treatment

- 다른 함수 호출에서 매개변수의 값을 얻어올 수 있는 경우,
  매개변수를 함수 호출로 바꾸기(Replace Parameter with Query)
- 다른 객체로부터 데이터를 뽑아서 매개변수로 전달하는 경우,
  객체 통째로 넘기기(Preserve Whole Objecct) (객체를 그대로 전달)
- 몇몇 파라미터들이 항상 함께 전달되는 매개변수들의 경우,
  매개변수 객체 만들기(Introduce Parameter Object)
- 함수의 다른 동작을 구분하는 플래그 역할의 매개변수는,
  플래그 인수 제거하기(Remove Flag Argument) (매개변수 제거)

전역 데이터 (Global Data)

Symptoms

코드 베이스 어디 에서든 값을 바꿀 수 있어, 변경한 코드를 찾아내기가 굉장히 어려움.

전역변수, 클래스필드, Singleton 데이터.

Treatment

- 변수 캡슐화하기(Encapsulate Variables) 로 데이터로의 접근, 수정 제한.

가변 데이터 (Mutable Data) (1/2)

Symptoms

의도치 않은 Data의 변경을 야기할 수 있는 경우.

Treatment (1/2)

- 몇몇 정해진 함수에서만 값을 갱신할 수 있도록, **변수 캡슐화하기(Encapsulate Variable)**
  : 값을 변경하는 **경로**를 제한함으로써, 모니터링과 코드 변경 용이
- 변수가 하나 이상의 목적으로 사용되는 경우, **변수 쪼개기(Split Variable)**
- 기존의 로직에서 값을 변경하는 로직을 분리하기 위해,
  문장 슬라이드하기(Slide Statement) 와 **함수 추출하기(Extract Function)**
- **질의 함수와 변경 함수 분리하기(Separate Query from Modifier)** 를 사용하여,
  Caller 에서 불필요한 동작을 하는 Side Effect 방지.

가변 데이터 (Mutable Data) (2/2)

Treatment (2/2)

- 생성자에서만 값을 변경하는 경우, 세터 제거하기(Remove Setting Method)
  : 불필요한 세터를 제거하여 데이터 변경 가능성 차단.
- 값을 계산하는 **시점**과 사용하는 시점이 다른 경우,
  파생 변수를 질의 함수로 바꾸기(Replace Derived Variable with Query)
  : 계산 시점을 사용하는 시점으로 이동
- 가변 데이터가 사용되는 **범위** 통제 방법
  - 여러 함수를 클래스로 묶기(Combine Functions into Class)
    : 데이터를 사용하는 함수들을 하나의 클래스로 묶기.
  - 여러 함수를 변환 함수로 묶기(Combine Functions into Transform)
    : 변환 함수를 통해서 해당 함수들에 호출
- 값을 변경하지 않는 경우, 참조를 값으로 바꾸기(Change Reference to Value)
  : 의도치 않은 변경 가능성 제거

뒤엉킨 변경 (Divergent Change)

Symptoms

하나의 모듈이 서로 다른 이유로 인해 여러 가지 방식으로 변경되는 일이 많은 경우

Treatment

- 여러 맥락이 혼재된 중에도, 이들의 순서가 자연스러운 경우, **단계 쪼개기(Split Phase)**
- 각기 다른 맥락의 함수를 호출하는 빈도가 높은 경우, **함수 옮기기(Move Function)**
  : 맥락별로 적당한 모듈을 만들어서 이동
  - 이때 호출하는 함수 중 여러 맥락의 일에 관여하는 부분이 있는 경우,
    옮기기에 앞서 **함수 추출하기(Extract Function)** 로 맥락 분리
  - 모듈이 클래스라면,
    클래스 추출하기(Extract Class)

산탄총(散彈銃) 수술 (Shotgun Surgery)

Symptoms

코드를 수정할 때마다 여러 클래스에서 수많은 자잘한 부분을 고쳐야 하는 경우.

: Responsibility가 여러 모듈에 흩어져 있는 경우

Treatment

- 함수 옮기기(Move Function) 와 필드 옮기기(Move Field)
  : 같은 맥락의 함수들을 하나의 모듈로 이동, 적절한 클래스 없으면 새로 정의.
- 함수들이 유사한 데이터들을 사용하는 경우,
  여러 함수를 클래스로 묶기(Combine Functions into Class)
- 데이터 구조를 변환하거나 보강하는 함수들이 있는 경우,
  여러 함수를 변환 함수로 묶기(Combine Functions into Transform)
- 코드들의 이동으로 원래의 클래스가 거의 비게 되는 경우
  : 함수 인라인하기(Inline Function) 또는 클래스 인라인 하기(Inline Class)

기능 편애 (Feature Envy) 1/2

Symptoms

어떤 함수가 자신의 속한 모듈보다, **다른** 모듈의 함수나 데이터와 **상호작용**이 더 많은 경우.

Treatment

- 함수 옮기기(Move Function)
  : 함수가 위치해야 할 클래스가 명확하다면, Behavior를 Data 근처 class 로 이동
- 함수의 특정 부분만 다른 객체와 상호작용이 많은 경우, 함수를 추출하여 해당 class로 이동
  : 함수 추출하기(Extract Function) 와 함수 옮기기(Move Function)
  - 상호작용하는 모듈이 여러 개이고,이동할 모듈이 명확하지 않은 경우, 가장 많이 상호 작용하는 모듈로 이동
  - 혹은 *이동에 앞서* 함수를 여러 부분으로 분리 추출하여 각각을 적합한 모듈로 이동

기능 편애 (Feature Envy) 2/2

When to Ignore

- 기본 규칙은 Data 와 Behavior 는 같은 모듈에 위치.
- 예외 : Strategy Pattern 과 Visitor Pattern
  Divergent Change 를 해결하기 위해 Behavior 를 재정의하여 기능 확장
  => Data 와 Behavior 를 분리하여 Feature Envy 를 발생시키는 예외 발생시킴.

데이터 뭉치 (Data Clumps)

Symptoms

동일한 3~4개의 데이터 항목이, 여러 위치에 뭉치로 몰려다니는 경우.

Clumps를 Class 로 묶어서 관리.

Treatment

- **클래스 추출하기(Extract Class)**
  : 동일한 데이터 항목들이 class field 를 구성할 경우 field 들을 데이터 클래스로 추출
- **매개변수 객체 만들기(Introduce Parameter Object)**
  : 데이터 뭉치가 함수의 매개변수를 구성하는 경우
- **객체 통째로 넘기기(Preserve Whole Object)**
  : 일부 데이터 뭉치가 다른 함수로 전달되는 경우 전체 데이터 객체를 매개변수로 전달

기본형 집착 (Primitive Obsession)

Symptoms

간단한 작업을 위해 작은 객체를 사용하기 보다, Primitive를 사용하는 경우.

Primitive 값이 조건부 동작을 제어하는 type code로 쓰인 경우.

Data Array에서 필드이름을 상수로 표현하는 경우.(ex. Data[id][NAME], Data[id][ADDR], ..)

```
const string phoneNum = "03112341234";
cout << "Tel : " + phoneNum.substr(0, 3) << "-" << phoneNum.substr(3, 4)
        << "-" << phoneNum.substr(7, 4);
```

Treatment

- **기본형을 객체로 바꾸기(Replace Primitive with Object)**
  : 연관 있는 Primitive field들과 behavior들을 묶어 클래스로 만들기
- Primitive의 값이 조건부 동작을 제어하는 type code로 쓰인 경우, 제어문을 클래스로 처리
  : **타입 코드를 서브클래스로 바꾸기(Replace Type Code with Subclasses)**
  조건부 로직을 다형성으로 바꾸기(Replace Conditional with Polymorphism)
- Primitive 데이터 뭉치가 함께 다니는 경우, 독립된 클래스로 만들기
  : **클래스 추출하기(Extract Class)** 와 **매개변수 객체 만들기(Introduce Parameter Object)**

반복되는 Switch문 (Repeated Switches)

Symptoms

똑같은 조건부 로직, switch문 혹은 길게 나열된 if문이 여러 곳에서 반복되는 경우.

Treatment

- Switch 를 Polymorphism 으로 재정의
  - **함수 추출하기(Extract Function)** 와 **함수 옮기기(Move Function)** 로 재정의할 클래스로 이동
  - **타입 코드를 서브클래스로 바꾸기(Replace Type Code with Subclasses)**
    : Type code를 참조하는 Switch문의 type code 제거하여 클래스로 분리
  - **조건부 로직을 다형성으로 바꾸기(Replace Conditional with Polymorphism)**

※ 예외) Factory method와 같이, instance를 생성하는 로직에서 사용되는 경우

반복문 (Loops)

Symptoms

Collection 탐색을 위해 반복문을 사용하는 경우.

Treatment

- 반복문을 파이프라인으로 바꾸기(Replace Loop with Pipeline)

Cf. 자바

```
List<String> names = new ArrayList<>();
for (Person p : people) {
    if ("programmer".equals(p.job))
        names.add(p.name);
}
```

```
List<String> names = people.stream()
        .filter(p -> "programmer".equals(p.job))
        .map(p -> p.name)
        .collect(Collectors.toList());
```

성의 없는 요소 (Lazy Element)

Symptoms

리팩토링 후 기능이 축소되거나 필요가 없어진 프로그램 요소.

Treatment

- 함수 인라인하기(Inline Function) 와 클래스 인라인하기(Inline Class)
  : 쓸모 없는 구성요소 제거
- 상속을 사용하는 경우, 계층 합치기(Collapse Hierarchy) 를 사용한 상속구조 제거

추측성 일반화 (Speculative Generality)

Symptoms

막연한 추측으로 작성해 두었으나 당장은 필요하지 않고, 이해하거나 관리하기 어려운 코드.

Test Code 말고는 사용되는 곳이 없는 코드.

Treatment

- 하는 일이 거의 없는 클래스의 경우, 계층 합치기(Collapse Hierarchy)
- 불필요한 위임의 경우, 함수 인라인하기(Inline Function) 와 클래스 인라인하기(Inline Class)
- 불필요한 매개변수의 경우, 함수 선언 바꾸기(Change Function Declaration)
- Test Code에서만 사용되는 경우, 죽은 코드 제거하기(Remove Dead Code)

임시 필드 (Temporary Field)

Symptoms

- 특정 상황에서만 값이 설정되는 필드가 있는 경우.

Treatment

- 임시 필드에 대해, **클래스 추출하기(Extract Class)**
  : 분리된 클래스로 이동하여, 필드에 항상 의미 있는 값이 설정
- 임시 필드와 관련된 함수에 대해, **함수 옮기기(Move Function)**
  : 분리된 클래스로 이동
- 조건부 로직이 임시 필드의 유효성을 확인한 후에 동작하는 경우,
  특이 케이스 추가하기(Introduce Special Case)
  : 조건부 로직을 제거하고 단순호출로 대체

메시지 체인 (Message Chains)

Symptoms

다른 객체를 요청하는 작업이, 체인처럼 연쇄적으로 이어지는 코드.

클라이언트가 한 객체에 제2의 객체를 요청하면, 제2의 객체가 제 3의 객체를 요청하고,
제3의 객체가 제 4의 객체를 요청하는 식으로 연쇄적 요청이 발생하는 경우

(ex.- a.getB().getC().getD().getData())

Treatment

- 위임 숨기기(Hide Delegate) : 체인 제거
- 결과 객체가 어느 대상에 사용되는지 알아내고, 객체가 사용되는 코드 부분에 대해,
  함수 추출하기(Extract Function) 를 통해 별도의 함수로 분리 후,
  함수 옮기기(Move function) 를 하여 Chain 아래로 밀어낸다.
- 체인을 구성하는 객체 중 특정 하나를 사용하는 코드가 제법 된다면,
  이 요구를 처리하기 위해 별도의 함수로 분리

※ 주의) 체인을 제거하는 중에, 중간객체가 또 다른 스멜인 중개자가 되기 쉬우므로, 체인의 최종 결과 객체부터 검토

중개자 (Middle Man)

Symptoms

어떤 클래스의 절반 이상의 메서드가 기능을 다른 클래스에 위임하는 경우.

Treatment

- 중개자 제거하기(Remove Middle Man) 을 실시하여 구현된 객체에 직접 접근하기
  - 별 기능이 없는 함수
    : 함수 인라인하기(Inline Function) 으로, 함수의 내용을 호출 객체에 직접 삽입
  - 추가할 기능이 있는 경우
    : 위임을 상속으로 바꾸기(Replace Delegation with Inheritance) 를
    사용하여 Middle Man 을 실제 객체의 서브 클래스로 전환

Proxy pattern 과 같이 의도를 가지고 생성된 경우라면 그대로 유지되어야 한다.

내부자 거래 (Insider Trading)

Symptoms

클래스간의 결합도(coupling)가 높아, 서로의 private 필드와 메서드에 접근하는 경우.

Treatment

- 함수와 필드 이동 (private부분)
  : **필드 옮기기(Move Field)** 와 **함수 옮기기(Move Function)**
- 공통부분이 있는 경우
  : **클래스 추출하기(Extract Class)** - 독립적 모듈
  위임 숨기기(Hide Delegate) - 중간자 모듈
- 상속구조에서 부모와 자식 간에, 지나친 coupling이 있는 경우
  : **상속을 위임으로 바꾸기(Replace Subclass with Delegation, Replace Superclass with Delegation)**

거대한 클래스 (Large Class)

Symptoms

클래스의 멤버(data, method)의 수가 너무 많은 경우.

클래스의 코드 양이 너무 많은 경우.

(클래스가 너무 많은 기능을 담고 있다는 근거가 될 수 있음)

Treatment

- 클래스의 멤버들을 여러 개로 나눌 수 있는 경우
  : **클래스 추출하기(Extract Class)**
- 추출보다 상속 관계로 만드는 것이 좋은 경우
  : **슈퍼클래스 추출하기(Extract Superclass)**
  타입 코드를 서브클래스로 바꾸기(Replace Type Code with Subclasses)

다른 인터페이스의 대안 클래스들 (Alternative Classes with Different Interfaces)

Symptoms

두 개 이상의 클래스가 동일한 동작을 하는 interface를 갖는 경우.

Treatment

- 기능은 같은데, Signature가 서로 다른 경우, Signature 일치시키기
  : **함수 선언 바꾸기(Change Function Declaration)**
- 클래스들 간 인터페이스가 같아질 때까지, 동일한 클래스로 함수를 이동
  함수 옮기기(Move function)
- 클래스들 사이의 공통 부분을, 상속 관계로 만들 수 있는 경우
  : **슈퍼클래스 추출하기(Extract Superclass)**

※ 예외) Signature를 일치시키는 것이 불가능하거나, 무의미할 정도로 어려운 경우

데이터 클래스 (Data Class)

Symptoms

data 필드와 그것에 대한 Getter/Setter 메서드만 있는 클래스.

데이터 보관만 담당하며, 구체적 데이터 조작은 다른 클래스에서 수행

Treatment

- Public field 의 경우, Getter/Setter로만 접근하게 제한
  : **필드 캡슐화하기(Encapsulate Field)**
- 데이터가 collection 타입인 경우, **컬렉션 캡슐화하기(Encapsulate Collection)**
- 변경하면 안 되는 데이터의 경우, **세터 제거하기(Remove Setting Method)**
- 다른 클래스에서 데이터를 조작하는 부분으로, **함수 옮기기(Move Function)**
  - 메서드 전체를 옮겨올 수 없는 경우, 사용되는 부분에 대해서만,
    함수 추출하기(Extract Function) 한 후에 **함수 옮기기(Move Function)**

※ 예외) immutable 로만 사용되는 경우

상속 포기 (Refused Bequest) (1/3)

- **Symptoms**

  자식 클래스가 부모 클래스의 메서드와 필드의 일부만 필요한 경우.

  불필요한 메서드는 아예 사용되지 않거나, 재정의, Exception 발생시키는 경우

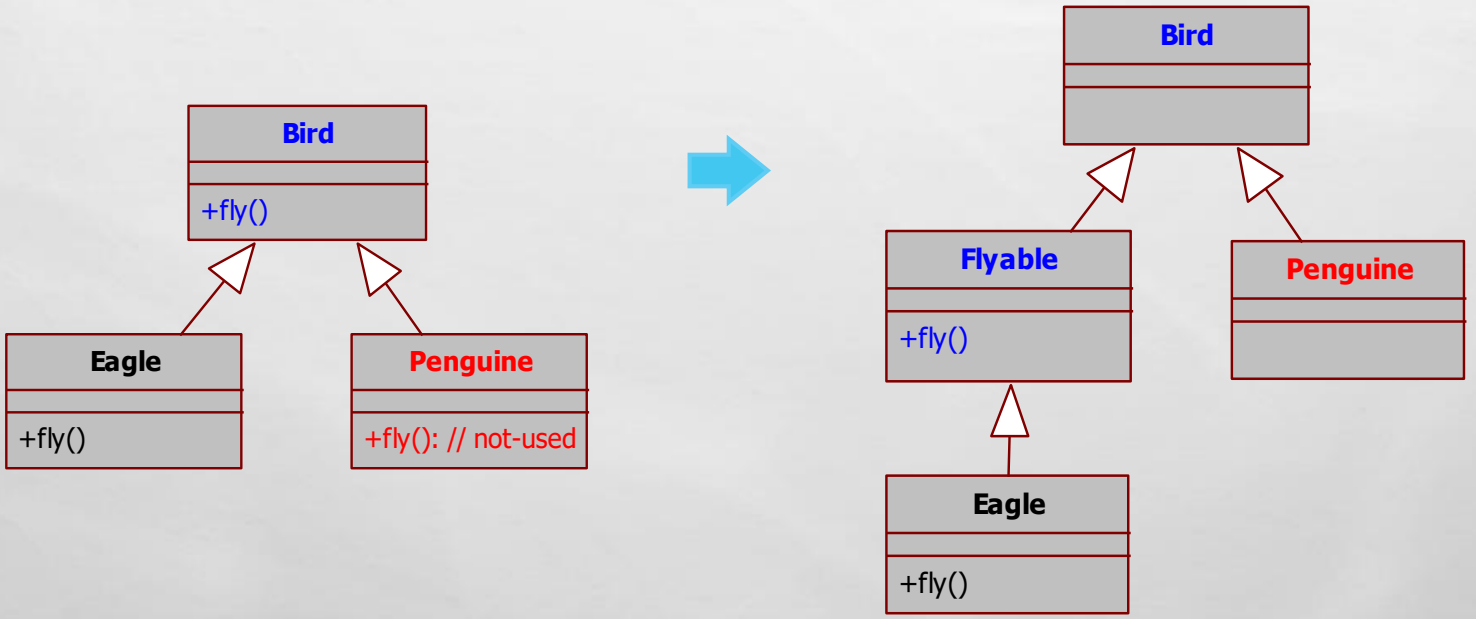
  잘못된 상속 관계에서 발생

  리스코프 치환의 원칙(슈퍼 클래스를 서브 클래스로 대체할 수 있어야 한다.) 위배

상속 포기 (Refused Bequest) (2/3)

Treatment (1/2)

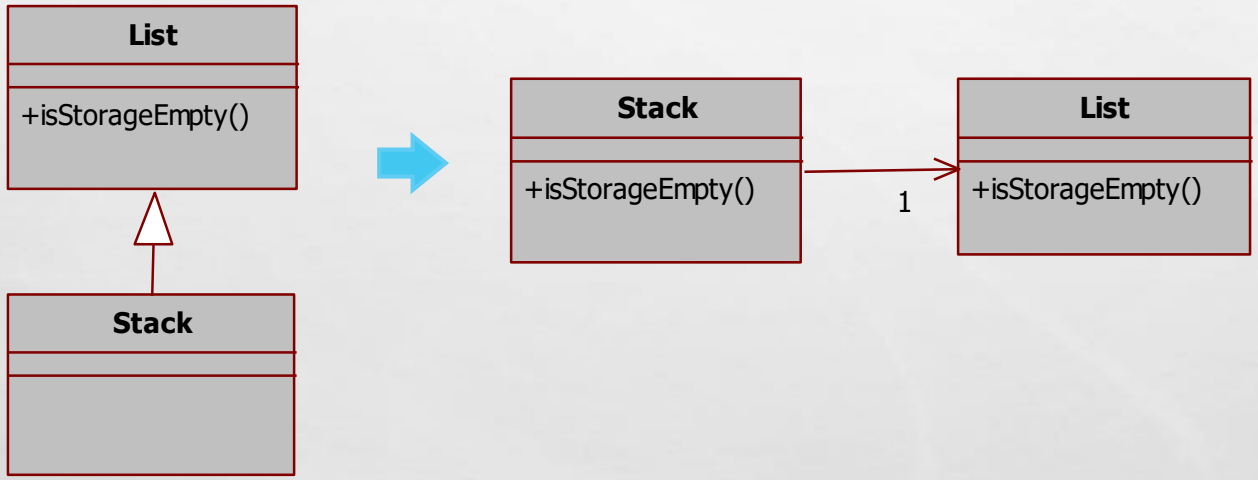
- 형제 클래스를 새로 만든 후, 필드와 함수 이동
  : 메서드 내리기(Push Down Method), 필드 내리기(Push Down Field)
  상속의 계층구조를 조정하여 해결하므로,
  자신은 축소된 부모 클래스를 상속받아 불필요한 멤버나 메소드를 상속받지 않음



상속 포기 (Refused Bequest) (3/3)

Treatment (2/2)

- 자식 클래스가 기능은 재사용하지만, 부모 클래스의 인터페이스를 지원하지 않는 경우
  : **상속을 위임으로 바꾸기(Replace Superclass with Delegation)**
  자식 클래스는 필요한 동작만 구현하고 상속 구조 제거



※ **"상속보다는 컴포지션을 사용하라"** (*처음에는 상속으로 접근한 후, 문제가 생기면 위임으로 변경*)

주석 (Comments)

Symptoms

함수가 코드를 설명하는 주석으로 채워져 있는 경우.

Treatment

- 코드 블록에 대한 주석이 있는 경우, **함수 추출하기(Extract Function)**
- 함수, 변수, 필드를 설명하는 주석이 필요한 경우, 좋은 이름으로 변경
  : **함수 선언 바꾸기(Change Function Declaration)**,
  변수 이름 바꾸기(Rename Variable), **필드 이름 바꾸기(Rename Field)**
- 시스템의 필수적인 상태에 관해 약간의 규칙을 설명하는 주석의 경우
  : **어서션 추가하기(Introduce Assertion)**

When you feel the need to write a comment,
first try to refactor the code so that any comment becomes superfluous.

리팩토링 기법

Category

- 기본적인 기법
- 캡슐화 연관 기법
- 기능이동 연관 기법
- 데이터 조직화 연관 기법
- 조건부 로직 간소화 연관 기법
- API 연관 기법
- 상속 연관 기법

기본적인 기법

- 함수 추출하기
- 함수 인라인하기
- 변수 추출하기
- 변수 인라인하기
- 함수 선언 바꾸기
- 변수 캡슐화하기
- 변수 이름 바꾸기
- 매개변수 객체 만들기
- 여러 함수를 클래스로 묶기
- 여러 함수를 변환 함수로 묶기
- 단계 쪼개기
- (category로...)

함수 추출하기(Extract Function) (<-> Inline Function) [기본기법 #1]

You have a **code fragment that can be grouped** together.

Turn the fragment into a function whose name explains the purpose of the function.

```
void printOwing(double amount) {
  printBanner();

  // Print details.
  cout << "name: " << name << endl;
  cout << "amount: " << amount << endl;
}
```

```
void printOwing(double amount) {
  printBanner();
  printDetails(amount);
}

void printDetails(double amount) {
  cout << "name: " << name << endl;
  cout << "amount: " << amount << endl;
}
```

- Related to

  Move Function, Introduce Parameter object, Parameterize Function

- **Smells**

  Duplication Code, Long Function, Mutable Data, Divergent Change, Feature Envy, Repeated Switches, Message Chains, Data Class, Comments

함수 인라인하기(Inline Function) (<-> Extract Function) [기본기법 #2]

A method's body is just as clear as its name.

*Put the method's body into the body of its callers and **remove the method**.*

```
int getRatings(const Driver& driver) {
    return moreThanFiveLateDeliveries(driver) ? 2 : 1;
}

bool moreThanFiveLateDeliveries(const Driver& driver) {
    return driver.numberOfLateDeliveries > 5;
}
```

```
int getRatings(const Driver& driver) {
    return driver.numberOfLateDeliveries > 5 ? 2 : 1;
}
```

- **Smells**

  Shotgun Surgery, Lazy Element, Speculative Generality, Middle Man

변수 추출하기(Extract Variable) (<-> Inline Variable) [기본기법 #3]

You have a **complicated expression**.

*Put the result of the expression, or parts of the expression, in **a local variable** with a name that explains **the purpose.***

```
return order.quantity * order.itemPrice -
    max(0.0, order.quantity - 500) * order.itemPrice * 0.05 +
    min(order.quantity * order.itemPrice * 0.1, 100.0);
```

```
const double basePrice = order.quantity * order.itemPrice;
const double quantityDiscount
    = max(0.0, order.quantity - 500) * order.itemPrice * 0.05;
const double shipping
    = min(order.quantity * order.itemPrice * 0.1, 100.0);
return basePrice - quantityDiscount + shipping;
```

- Related to

  Extract Function, Replace Temp with Query, Replace Function with Command

- **Smells**

  Comments

변수 인라인하기(Inline Variable) (<-> Extract Variable) [기본기법 #4]

You have a **variable** whose **name doesn't really communicate more than** the expression itself, and the var iable is getting in the way of other refactorings.

Replace all references to that variable with the expression.

```
double basePrice
    = order.quantity * order.itemPrice;
return basePrice > 1000;
```

```
return order.quantity * order.itemPrice > 1000;
```

- Related to

  Replace Temp with Query, Extract Function

함수 선언 바꾸기(Change Function Declaration) [기본기법 #5]

The **name** of a method does not reveal its purpose.
Parameters of a function does not fit in with the rest of its world.

Change the signature of the function.

```
Circle(radius) { ... }

string inNewEngland(Customer& aCustomer) { ... }
```

```
Circle(center, radius) { ... }

string inNewEngland(string stateCode) { ... }
```

- **Smells**

  Mysterious Name, Speculative Generality, Alternative Classes with Different Interfaces, Comments

변수 캡슐화하기(Encapsulate Variable) [기본기법 #6]

There is a **public** field.

*Make it **private** and provide **accessors.***

```
class Customer{
public:
    string forename;
private:
};
```

```
class Customer{
public:
    string getForename() const {
        return m_forename;
    }

    void setForename(const string& forename){
        m_forename= forename;
    }

private:
    string m_forename;
};
```

- **Smells**

  Global Data, Mutable Data, Data Class

변수 이름 바꾸기(Rename Variable) [기본기법 #7]

Name of a variable didn't explain its purpose

Rename a variable.

```
double a = height * width;
```

```
double area = height * width;
```

- **Smells**

  Mysterious Name, Data Class

매개변수 객체 만들기(Introduce Parameter Object) [기본기법 #8]

You have **a group of parameters** that naturally go together.

*Replace them with **an object**.*

```
amountInvoiced(sDate, eDate) { ... }
amountReceived(sDate, eDate) { ... }
amountOverdue(sDate, eDate) { ... }
```

```
amountInvoiced(dateRange) { ... }
amountReceived(dateRange) { ... }
amountOverdue(dateRange) { ... }
```

- Related to

  Preserve Whole Object

- **Smells**

  Long Function, Long Parameter List, Data Clumps, Primitive Obsession

여러 함수를 클래스로 묶기(Combine Functions into Class) [기본기법 #9]

You see a **group of functions** that **operate closely together on a common body of data**.

Group them in a class.

```
double base(Reading reading) {...}
double taxableCharge(Reading reading) {...}
double calculateBaseCharge(Reading reading)
{...}
```

```
class Reading{
public:
    double base() {...}
    double taxableCharge() {...}
    double calculateBaseCharge() {...}
}
```

- Related to

  Combine Functions into Transform

- **Smells**

  Mutable Data, Shotgun Surgery

여러 함수를 변환 함수로 묶기(Combine Functions into Transform) [기본기법 #10]

Software involves feeding data into programs that **calculate various derived information** from it.

*Use a data transformation function that takes the **source data as input** and **calculates all the derivations**, putting each derived value as a field in the output data*

```
double base(Reading reading) {...}
double taxableCharge(Reading reading) {...}
```

```
Reading enrichingReading(Reading argReading) {
    Reading aReading(argReading);
    aReading.m_baseCharge = calculateBaseCharge(aReading);
    aReading.m_taxableCharge = taxableCharge(aReading);
    return aReading;
}
```

- Related to

  Combine Functions into Transform

- **Smells**

  Mutable Data, Shotgun Surgery

단계 쪼개기(Split phase) [기본기법 #11]

Run into code that's dealing with two different things.

Split it into separate modules.

```
std::vector<std::string> orderData = split(orderString, R"([₩s,]+)");
int productPrice
    = priceList[atoi(split(orderData[0], R"([-]+)")[1].c_str())];
int orderPrice = atoi(orderData[1].c_str()) * productPrice;
```

```
Order orderRecord = Order(orderString);
int orderPrice = orderRecord.price(orderRecord, priceList);

Order(string aString){
    std::vector<std::string> value = split(aString, R"([₩s,]+)");
    productID = atoi(split(value[0], R"([-]+)")[1].c_str());
    quantity = atoi(value[1].c_str());
}

int price(Order order, int* priceList) {
    return order.quantity * priceList[order.productID];
}
```

- **Smells**

  Divergent Change, Shotgun Surgery

캡슐화 연관 기법

- 모듈화의 가장 중요한 기준 중 하나인, 캡슐화와 연관된 리팩토링 기법.

- 필드 캡슐화하기
- 컬렉션 캡슐화하기
- 기본형을 객체로 바꾸기
- 임시 변수를 질의 함수로 바꾸기
- 클래스 추출하기
- 클래스 인라인하기
- 위임 숨기기
- 중개자 제거하기
- 알고리즘 교체하기
- (category로...)

필드 캡슐화하기(Encapsulate Field) [캡슐화부분 #1]

You need to interface with a record structure in a traditional programming environment.

Make a dumb data object for the record.

```
typedef struct ORGANIZATION
{
    string name;
    string country;
} ORGANIZATION;
```

```
class Organization {
public:
    Organization(string* data) {
        m_name = data[0];
        m_country = data[1];
    }

    string getName() { return m_name;}
    string getCountry() { return m_country;}
    void setName(string arg) { m_name = arg;}
    void setCountry(string arg) { m_country = arg;}

private:
    string m_name;
    string m_country;
};
```

- Related to

  Encapsulate field

컬렉션 캡슐화하기(Encapsulate Collection) [캡슐화부분 #2]

A method returns a collection.

*Make it return a **read-only view** and provide **add/remove methods.***

```
class  Person {
    Course getCourses() { return courses; }
    void setCourses(aList) {
        courses = aList; }
}
```

```
class Person{
public:
    vector<Course> getCourses(){
        return m_courses;
    }
    void addCourse(const Course& aCourse) {  }
    void removeCourse(const Course& aCourse) {  }

private:
    vector<Course> m_courses;
};
```

- **Smells**

  Data Class

기본형을 객체로 바꾸기(Replace Primitive with Object)

You have simple data items such as **numbers or strings** that **needs additional data or behavior.**

Turn the data item into an object

```
for (auto& item : orders) {
    if (item.getPriority() == "high" || item.getPriority() == "rush")
    {
            highPriorityOrders.emplace_back(item);
    }
}
```

```
for (auto& item : orders) {
    if (item.getPriority().higherThan(Priority("normal")))
    {
        highPriorityOrders.emplace_back(item);
    }
}
```

- Related to

  Extract Class, Introduce Parameter Object, Replace Function with Command

- **Smells**

  Primitive Obsession

임시 변수를 질의 함수로 바꾸기(Replace Temp With Query)

You are using a local variable to **hold the result of an expression**.

Extract the expression into a method.

Replace all references to the temp with the expression.

The new method can then be used in other methods.

```
double basePrice = m_quantity * m_itemPrice;
double discountFactor;

if (basePrice > 1000)
    discountFactor = 0.98;
else
    discountFactor = 0.95;

return basePrice * discountFactor;
```

```
double getPrice() {
    double discountFactor;

    if (basePrice() > 1000)
        discountFactor = 0.98;
    else
        discountFactor = 0.95;
    return basePrice() * discountFactor;
}
inline double basePrice() { return m_quantity * m_itemPrice; }
```

- Related to

  Split Variable, Extract Function, Separate Query from Modifier

- **Smells**

  Duplication Code, Long Function

클래스 추출하기(Extract Class) (<-> Inline Class)

You have **one** class doing work that should be done **by two**.

*Create **a new class** and move the relevant fields and methods from the old class into the new class.*

```
class Person {
public:
    string getName(){  }
    string getTelephoneNumber() {}
private:
    string officeAreaCode(){}
    string officeNumber(){}
};
```

```
class Person {
public:
    Person(TelephoneNumer& telephoneNumber)
        :m_telephoneNumber(telephoneNumber) {
    }
    TelephoneNumer getTelephoneNumer(){
        return m_telephoneNumber; }

private:
    string m_name;
    TelephoneNumer& m_telephoneNumber;
};
```

```
class TelephoneNumer {
public:
    string getAreaCode() {   }
    string getNumber() {   }
    void setAreaCode(string arg) {   }
    void setNumber(string arg) {   }
};
```

- Related to

  Extract Subclass, Replace Primitive with Object

- **Smell**

  Divergent Change, Data Clumps, Primitive Obsession, Temporary Field, Insider Trading, Large Class

클래스 인라인하기(Inline Class) (<-> Extract Class) [캡슐화부분 #6]

A class isn't doing very much.

Move all its features into another class and delete it.

```
class Person {
public:
    Person(TelephoneNumer& telephoneNumber)
        :m_telephoneNumber(telephoneNumber) {
    }
    TelephoneNumer getTelephoneNumer(){
        return m_telephoneNumber; }

private:
    string m_name;
    TelephoneNumer& m_telephoneNumber;
};

class TelephoneNumber {    };
```

```
class Person {
public:
    string getName(){  }
    string getOfficeAreaCode(){ }
    string getOfficeNumber(){ }

private:
};
```

- **Smells**

  Shotgun Surgery, Lazy Element, Speculative Generality

위임 숨기기(Hide Delegate) (<-> Remove Middle Man) [캡슐화부분 #7]

A client is calling a delegate class of an object.

Create methods on the server to hide the delegate.

```
Person manager = aPerson->getDepartment()->getManager();

class Person {
public:
    Person(string name, shared_ptr<Department> department)
        :m_name(name)) {
    }
    ...
}

class Department {
public:
    shared_ptr<Person> getManager() {
        return m_manager
}
```

```
Person manager = aPerson.getManager();

class Person {
public:
   string getManager(){
        return m_department->getManager();
   }
   shared_ptr<Department> getDepartment(){
        return m_department;
   }

private:
   shared_ptr<Department> m_department;
}
```

- **Smells**

  Message Chains, Insider Trading

중개자 제거하기(Remove Middle Man) (<-> Hide Delegate) [캡슐화부분 #8]

A class is doing **too much simple delegation**.

Get the client to call the delegate directly.

```
Person manager = aPerson.getManager();

class Person {
public:
    string getManager(){
        return m_department->getManager();
    }
    shared_ptr<Department> getDepartment(){
        return m_department;
    }
private:
    shared_ptr<Department> m_department;
}
```

```
Person manager = aPerson->getDepartment()->getManager();

class Person {
public:
    Person(string name, shared_ptr<Department> department)
        :m_name(name)) {
    }
    ...
}

class Department {
public:
    shared_ptr<Person> getManager() {
        return m_manager
}
```

- **Smells**

  Middle Man

알고리즘 교체하기(Substitute Algorithm) [캡슐화부분 #9]

You want to replace an algorithm with one that is clearer.

Replace the body of the method with the new algorithm.

```
int foundPerson(vector<string> people) {
    for(int i=0; i<people.size(); i++) {
        if(people[i]=="Don") {
            return 1;
        }
        if(people[i]=="John") {
            return 2;
        }
        if(people[i]=="Kent") {
            return 3;
        }
    }
    return 0;
}
```

```
int foundPerson(vector<string> people) {
    map<string, int> candidates = {{"Don", 1}, {"John", 2}, {"Kent", 3}};

    for(vector<string>::iterator iter = people.begin(); iter != people.end(); iter++) {
        if (candidates[*iter] != 0)
            return candidates[*iter];
    }
    return 0;
}
```

- **Smells**

  Duplication Code, Long Function

기능이동 연관 기법

- 프로그램 요소를 다른 큰 요소로 옮기는 것과 연관된 리팩토링 기법.

- 함수 옮기기
- 필드 옮기기
- 문장을 함수로 옮기기
- 문장을 호출한 곳으로 옮기기
- 인라인 코드를 함수 호출로 옮기기
- 문장 슬라이드하기
- 반복문 쪼개기
- 반복문을 파이프라인으로 바꾸기
- 죽은 코드 제거하기
- (category로...)

함수 옮기기(Move Function) [기능이동 #1]

A function is, or will be, using or used by more features of another class than the class on which it is define d.

Create a new function with a similar body in the class it uses most.
Either turn the old method into a simple delegation, or remove it altogether.

```
class ClassA {
    aMethodA() { ... }
}
class ClassB {
}
```

```
class ClassA {
}
class ClassB {
    aMethodA() { ... }
}
```

- Related to

  Extract Function, Move Field, Extract Class, Inline Class, Introduce Parameter Object

- **Smells**

  Divergent Change, Shotgun Surgery, Feature Envy, Repeated Switches, Temporary Field, Message Chains, Insider Trading, Alternative Classes with Different Interfaces, Data Class

필드 옮기기(Move Field) [기능이동 #2]

A field is, or will be, used by another class more than the class on which it is defined.

Create a new field in the target class, and change all its users.

```
class ClassA {
private:
    aDataFieldA;
}
class ClassB {
}
```

```
class ClassA {
}
class ClassB {
private:
    aDataFieldA;
}
```

- Related to

  Extract Class, Inline Class

- **Smells**

  Shotgun Surgery, Insider Trading

문장을 함수로 옮기기(Move Statements into Function)(<-> Move Statements to Callers)

Same code executed every time I call a particular function.

Combine that repeating code into the function itself.

```
result.append("Title: " + aPerson.getPhoto()->getTitle() + "₩n");
    result.append(photoData(photo));

string photoData(shared_ptr<Photo> photo) {
    string retString;
    retString.append("Name: " + photo->getLocation() + "₩n");
    retString.append("Filetype: " + photo->getFiletype() + "₩n");
    return retString;
}
```

```
aPerson.setPhoto(photo);

string photoData(shared_ptr<Photo> photo) {
    string retString;
    retString.append("Title : " + photo->getTitle() + "₩n");
    retString.append("Name: " + photo->getLocation() + "₩n");
    retString.append("Filetype: " + photo->getFiletype() + "₩n");
    return retString;
}
```

- Related to

Extract Function, Inline Function, Change Function Declaration, Slide Statements

문장을 호출한 곳으로 옮기기(Move Statements to Callers)(<->Move Statements into Function)

Common behavior used in several places **needs to vary** in some of its calls.

Move the varying behavior out of the function to its callers.

```
emitPhotoData(photo);

void emitPhotoData(shared_ptr<Photo> photo) {
    cout << "Title: " << photo->getTitle() << endl;
    cout << "Name: " << photo->getLocation() << endl;
    cout << "Filetype: " << photo->getFiletype() << endl;
}
```

```
emitPhotoData(photo);
cout << " Filetype : " << photo-> getFiletype() << endl;

void emitPhotoData(shared_ptr<Photo> photo) {
    cout << "Title: " << photo->getTitle() << endl;
    cout << "Name: " << photo->getLocation() << endl;

}
```

- Related to

Extract Function, Inline Function, Change Function Declaration

인라인 코드를 함수 호출로 옮기기(Replace Inline Code With Function Call)

Inline code that's doing **the same thing** that I have in an **existing function**.

Replace that inline code with a function call.

```
bool appliesToMass = false;
for (auto s : states) {
    if ("MA"==s)
        appliesToMass = true;
}
```

```
appliesToMass = states.contains("MA");
```

문장 슬라이드하기(Slide Statements) [기능이동 #6]

Several lines of code access the same data structure.

Make them to be together rather than intermingled with code accessing other data structures.

```
Plan pricingPlan = retrievePricingPlan();
Order order = retreiveOrder();
double charge;
double chargePerUnit = pricingPlan.unit;
```

```
Plan pricingPlan = retrievePricingPlan();
double chargePerUnit = pricingPlan.unit;

Order order = retreiveOrder();
double charge;
```

- **Smells**

  Duplication Code, Mutable Data

반복문 쪼개기(Split Loop) [기능이동 #7]

Two different things in the same loop.

Split the loop.

```
double totalSalary = 0;
double averageAge = 0;

for (auto& e : m_employees) {
    averageAge += e.getAge();
    totalSalary += e.getSalary();
}
averageAge /= m_employees.size();
```

```
double totalSalary = 0;
for (auto& e : m_employees) {
    totalSalary += e.getSalary();
}

double averageAge = 0;
for (auto& e : m_employees) {
    averageAge += e.getAge();
}
averageAge = averageAge / people.length;
```

Smells

Long Function, Loops

죽은 코드 제거하기(Remove Dead Code) [기능이동 #9]

Unused code is still a significant burden when trying to understand how the software works.

Delete it.

```
If (false) {
    doSomethingThatUsedToMatter();
}
```

- **Smells**

  Speculative Generality

데이터 조직화 연관 기법

데이터 구조와 연관된 리팩토링 기법.

- 변수 쪼개기
- 필드 이름 바꾸기
- 파생 변수를 질의 함수로 바꾸기
- 참조를 값으로 바꾸기
- 값을 참조로 바꾸기
- 매직 리터럴 바꾸기
- (category로...)

변수 쪼개기(Split Variable) [데이터조직화 #1]

A value that's used **for different purposes** is a breeding ground for **confusion and bugs.**

*Use Split Variable to **separate the usages.***

```
int temp = 2 * (height + width);
cout << temp << endl;

temp = height * width;
cout << temp << endl;
```

```
int perimeter = 2 * (height + width);
cout << perimeter << endl;

int  area = height * width;
cout << area << endl;
```

Smells

Mutable Data

필드 이름 바꾸기(Rename Field)

Name of a field **didn't explain** its purpose.

Rename the field.

```
class Organization {
private:
    string name;
    string getName() {...}
}
```

```
class Organization {
private
    string title;
    string getTitle() {...}
}
```

- **Smells**

  Mysterious Name

파생 변수를 질의 함수로 바꾸기(Replace Derived Variable with Query)

Data changes can often **couple** together parts of **code** **in awkward ways**, with changes in one part **leading** to knock-on effects that are **hard to spot**.

Remove any variables *that could be just as easily calculate.*

```
int discountedTotal() {
    return discountedTotal;
}

void setDiscount(int aNumber) {
    int old = discount;
    discount = aNumber;
    discountedTotal += old - aNumber;
}
```

```
int discountedTotal() {
    return baseTotal - discount;
}

void setDiscount(int aNumber) {
    discount = aNumber;
}
```

- **Smells**
  - Mutable Data

참조를 값으로 바꾸기(Change Reference to Value)(<->Change Value to Reference)

A field is treated as a value which is **immutable**, **not** want to be **shared** between several objects.

Replace the entire inner object with a new one that has the desired property.

```
class Product {
public:
    void applyDiscount(int arg) {
        price.amount -= arg;
    }
private:
    Money price;
}
```

```
class Product {
public:
    void applyDiscount(int arg) {
        price = Money(price.m_amount - arg, price.m_currency);
    }
private:
    Money price;
};
```

- **Smells**

  Mutable Data

값을 참조로 바꾸기(Change Value to Reference)(<->Change Reference to Value)

Physical copies of the same logical data need to be updated.

*Create **only one object for an entity**, and then retrieve it from the repository.*

```
Customer c(customerData);
```

```
Customer c =
        customerRepository.get(customerData.id);
```

매직 리터럴 바꾸기(Replace Magic Literal)

You have a literal number with a particular meaning.

Create a constant, name it after the meaning, and replace the number with it.

```
double potentialEnergy(double mass,
double height) {
  return mass * height * 9.81;
}
```

```
static const double GRAVITATIONAL_CONST = 9.81;

double potentialEnergy(double mass, double height) {
  return mass * height * GRAVITATIONAL_CONST;
}
```

조건부 로직 간소화 연관 기법

- 프로그램의 힘을 강화하는데 크게 기여하지만, 혼란과 버그를 유발하는 주요 원인이 되는 부분인, 조건문과 연관된 리팩토링 기법.

  - 조건문 분해하기
  - 조건식 통합하기
  - 중첩 조건문을 보호 구문으로 바꾸기
  - 조건부 로직을 다형성으로 바꾸기
  - 특이 케이스 추가하기
  - 어서션 추가하기
  - 제어 플래그를 탈출문으로 바꾸기
  - (category로...)

조건문 분해하기(Decompose Contional) [조건 #1]

You have a **complicated conditional** (if-then-else) statement.

*Extract functions **from the condition**, then **part**, and **else parts**.*

```
if ((cur_t < WINTER_START_T) || (cur_t > WINTER_END_T)) {
    charge = quantity * winterRate + winterServiceCharge;
}
else {
    charge = quantity * summerRate;
}
```

```
if (isSummer(cur_t)) {
    charge = summerCharge(quantity);
}
else {
    charge = winterCharge(quantity);
}
```

- **Smells**
  - Long Function

조건식 통합하기(Consolidate Conditional Expression) [조건 #2]

A series of conditional checks where each check is different yet the resulting **action** is the **same**.

*Combine them **into a single conditional expression** and extract it.*

```
double disabilityAmount() {
  if (seniority < 2)   return 0;
  if (monthsDisabled > 12) return 0;
  if (isPartTime)   return 0;
  ...
}
```

```
double disabilityAmount() {
  if (isNotEligibleForDisability())   return 0;
    ...
}
```

- **Smells**

  Duplication Code

중첩 조건문을 보호 구문으로 바꾸기 (Replace Nested Conditional With Guard Clauses) [조건#3]

A method has conditional behavior that does not make clear the normal path of execution.

*Use **guard clauses for all the special cases.***

```
double getPayAmount() {
  double result;
  if (_isDead)
    result = deadAmount();
  else {
    if (_isSeparated){
      result = separatedAmount();
    }
    else {
      if (_isRetired)    result = retiredAmount();
      else    result = normalPayAmount();
    }
  }
  return result;
}
```

```
double getPayAmount() {
  if (_isDead)    return deadAmount();
  if (_isSeparated) return separatedAmount();
  if (_isRetired)    return retiredAmount();
  return normalPayAmount();
}
```

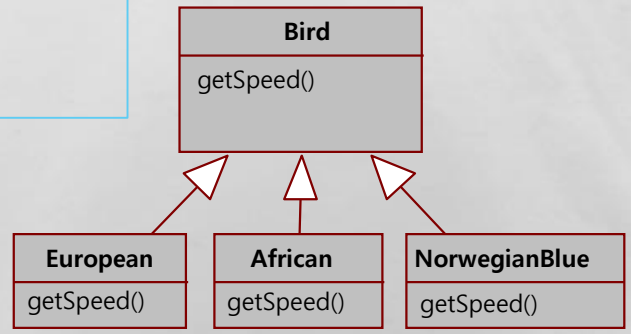
조건부 로직을 다형성으로 바꾸기(Replace Conditional with Polymorphism)

You have a conditional that chooses **different behavior depending on the type of an object.**

*Move **each leg of the conditional** to **an overriding method** in a subclass.*
Make the original method abstract.

```
class Bird {
public:
  double getSpeed() {
    switch (type) {
      case EUROPEAN:
          return getBaseSpeed();
      case AFRICAN:
          return getBaseSpeed()-getLoadFactor().;
      case NORWEGIAN:
          return (isNailed)?0:getBaseSpeed(voltage);
    }
  }
}
```

```
class Bird {
public:
      virtual double getSpeed()=0;
}
class European : public Bird {
      double getSpeed() override { return ...; }
}
class African : public Bird {
      double getSpeed() override { return ...; }
}
class Norwegian : public Bird {
      double getSpeed() override { return ...; }
}
```



Smells

Long Function, Primitive Obsession, Repeated Switches

특이 케이스 추가하기(Introduce Special Case) [조건 #5]

Many parts of the code base having **the same reaction to a particular value**.

Use Special Case pattern creating a special-case element
that captures all the common behavior
*to replace **most of the special-case checks** with **simple calls.***

```
if (aCustomer == null)
  customerName = "occupant";
else
  customerName = customer.getName();
```

```
class UnknownCustomer : public Customer {
    string getName() {
        return "occupant";
    }
}

customerName = customer.getName();
```

- Related to

  Replace Conditional with Polymorphism

- **Smells**

  Temporary Field

어서션 추가하기(Introduce Assertion) [조건 #6]

A section of code assumes something about the state of the program.

*Make the **assumption explicit** with an assertion.*

```
double getExpenseLimit_p() {
        // should have either expense limit or a primary project
        return (_expenseLimit != NULL_EXPENSE) ?
                _expenseLimit :
                _primaryProject->getMemberExpenseLimit();
    }
```

```
double getExpenseLimit() {
        assert(_expenseLimit != NULL_EXPENSE || _primaryProject != nullptr);

        return (_expenseLimit != NULL_EXPENSE) ?
                _expenseLimit :
                _primaryProject->getMemberExpenseLimit();
    }
```

- **Smells**
  - Comments

제어 플래그를 탈출문으로 바꾸기(Replace Control Flag with Break) [조건 #7]

You have a variable that is acting as a control flag for a series of boolean expressions.

Use a break or return instead.

```
bool found = false;
for( auto& p : people ) {
    if( !found ) {
        if(p == "Don" ) {
            sendAlert();
            found = true;
        }
    }
}
```

```
for( auto& p : people ) {
    if(p == " Don " ) {
        sendAlert();
        break;
    }
}
```

API 연관 기법

- S/W블록을 서로 연결하는 API와 관련된 리팩토링 기법.
  API는 다른 작업들과 마찬가지로 잘 만드는 것이 중요한 동시에 어려운 부분.

- 질의 함수와 변경 함수 분리하기
- 함수 매개변수화 하기
- 플래그 인수 제거하기
- 객체 통째로 넘기기
- 매개변수를 질의 함수로 바꾸기
- 질의 함수를 매개변수로 바꾸기
- 세터 제거하기
- 생성자를 팩터리 함수로 바꾸기
- 함수를 명령으로 바꾸기
- 명령을 함수로 바꾸기
- 수정된 값 반환하기
- 오류 코드를 예외로 바꾸기
- 예외를 사전확인으로 바꾸기
- (category로...)

질의 함수와 변경 함수 분리하기(Separate Query from Modifier)

You have a method that **returns a value** but **also changes the state** of an object.

*Create two methods, one **for the query** and one **for the modification**.*

```
string miscreant = alertForMiscreant(passengers);

string alertForMiscreant(vector<string> people) {
    for (const string &p : people) {
        if (p == "Don") {
            setOffAlarms();
            return "Don";
        }
        if (p == "John") {
            setOffAlarms();
            return "John";
        }
    }
    return "";
}
```

```
if (findMiscreant(passengers) != "")
    setOffAlarms();
string miscreant = alertForMiscreant(passengers);

string alertForMiscreant(vector<string> people) {
    for (const string &p : people) {
        if (p == "Don") {
            return "Don";
        }
        if (p == "John") {
            return "John";
        }
    }
    return "";
}
```

- Related to

  Replace Temp with Query

- **Smells**

  Mutable Data

함수 매개변수화 하기(Parameterize Function) [API부분 #2]

Several methods do similar things but with different values contained in the method body.

***Remove the duplication** by using a single function **with parameters** for the different values.*

```
int bottomBand(int usage) {
    return min(usage, 100);
}

int middleBand(int usage) {
    return usage > 100 ? min(usage, 200) - 100 : 0;
}

int topBand(int usage) {
    return usage > 200 ? usage - 200 : 0;
}
```

```
int withinBand(int usage, int bottom, int top) {
    return usage > bottom ? min(usage, 200) - bottom : 0;
}
```

- Related to

  Extract Method

- **Smells**

  Duplication Code

플래그 인수 제거하기(Remove Flag Argument <-> Parameterized Method)

You have a function that **runs different code depending on** the values of an enumerated **parameter.**

*Create a **separate function** for each value of the parameter.*

```
void setValues(string name, int value) {
  if (name == "height")) {
    height = value;
    return;
  }
  if (name == "width")) {
    width = value;
    return;
  }

}
```

```
void setHeight(int arg) {
  height = arg;
}
void setWidth(int arg) {
  width = arg;
}
```

Smells

Long Parameter List

객체 통째로 넘기기(Preserve Whole Object)

You are getting **several values from an object** and passing these values as parameters in a method call.

*Send the **whole object** instead.*

```
int low = daysTempRange.getLow();
int high = daysTempRange.getHigh();
If (plan.withinRange(low, high))
{ ... }
```

```
If (plan.withinRange(daysTempRange))
{ ... }
```

- Related to

  Introduce Parameter Object, Replace Parameter with Method Call

- **Smells**

  Long Function, Long Parameter List, Data Clumps

매개변수를 질의 함수로 바꾸기 (Replace Parameter with Query<->Replace Query with Parameter)

Calling a query method and passing its results as the parameters of another method, while that method could call the query directly.

Instead of passing the value through a parameter, try placing a query call inside the method body..

```
void availableVacation(Employee& anEmployee, int grade) {
        // 연휴 계산
}
```

```
void availableVacation(Employee& anEmployee) {
        const int grade =    anEmployee .m_grade;
}
```

- **Smells**

  Long Parameter List

질의 함수를 매개변수로 바꾸기(Replace Query with Parameter<->Replace Parameter with Method Call

References to something in the function's scope that you want to move away - might be a reference to a gl obal variable, or to an element in the same module. You want to make the function **no longer dependent on the element**.

*Shift the **responsibility** of resolving the reference **to the caller** of the function.*

```
targetTemperature(plan);

void targetTemperature(Plan& plan) {
    currentTemperature =
      thermostat.currentTemperature;
    // rest of function...;
}
```

```
targetTemperature(plan,
        thermostat.currentTemperature);

void targetTemperature(Plan& plan,
        double currentTemperature) {
    // rest of function...
}
```

Smells

Long Parameter List

세터 제거하기(Remove Setting Method) [API부분 #7]

A field should be set at creation time and **never altered.**

Remove any setting method for that field.

```
class Person {
    string getImmutableValue() { ... }
    setImmutableValue() { ... }
}
```

```
class Person {
    getImmutableValue() { ... }
}
```

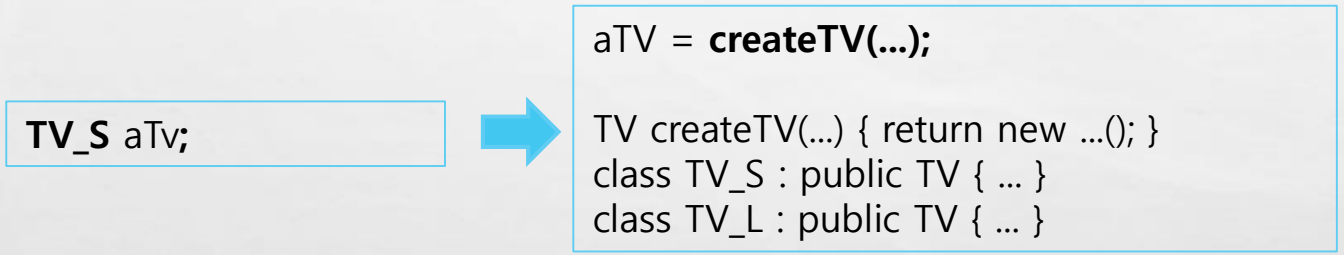
- **Smells**

  Mutable Data, Data Class

생성자를 팩터리 함수로 바꾸기(Replace Constructor With Factory Function)

You want to do more than simple construction when you create an object.

Replace the constructor with a factory method.



- Related to

  Change Value to Reference, Replace Type Code with Subclasses

함수를 명령으로 바꾸기 (Replace Function with Command) (<->Replace Command with Func.)

You have a long function that uses **local variables** in such a way that you **cannot apply Extract Method**.

*Turn **the function** into **its own (command) object***
*so that all the local variables become **fields** on that object.*
You can then decompose the function into other methods on the same object.

```
double score(candidate, medicalExam, scoringGuide)
{
    double result=0;
    double healthLevel=0;
    // Perform long computation.
 }
```

```
double score(candidate, medicalExam, scoringGuide) {
    return Scorer(candidate, medicalExam, scoringGuide).execute();
 }

class Scorer{
public:
    Scorer(int candidate, int medicalExam, int scoringGuide) :
        m_candidate(candidate),
        m_medicalExam(medicalExam),
        m_scoringGuide(scoringGuide) {
    }

    double execute() {
        m_result = 0;
        m_healthLevel= 0;
        // Perform long computation.
    }
...
};
```

- Related to

  Extract Class, Extract Function

- **Smells**

  Long Function

명령을 함수로 바꾸기(Replace Command with Function)(<->Replace Func. with Command)

A function **isn't too complex**, then a command object is more trouble than its worth.

Turned into a regular function.

```
class ChargeCalculator {
public:
ChargeCalculator(const Customer& customer, const int usage) {
        m_customer = customer;
        m_usage = usage;
    }
    double execute() {
        return m_customer.m_rate * m_usage;
    }
...
}
```

```
double charge(const Customer& customer, int usage) {
    return customer.m_rate * usage;
}
```

- Related to
  - Inline Class

수정된 값 반환하기(Return Modified Value)

The nastiest coupling that can exist between code fragments is when **multiple fragments** are reading and writing **a block of data**.

Have functions update variables by returning a value that the caller assigns to the underlying variable.

```
totalAscent = 0;
calculateAscent();

void calculateAscent() {
   for (int i = 1; i < points.length; i++) {
      totalAscent += ...;
   }
}
```

```
totalAscent = calculateAscent();

int calculateAscent() {
   int ret_value = 0;
   for (int  i = 1; i < points.length; i++) {
      ret_value += ...;
   }
   return ret_value;
}
```

오류 코드를 예외로 바꾸기(Replace Error Code with Exception)

A method returns a special code to indicate an error.

Throw an exception instead.

```
int withdraw(int amount) {
    if (amount > m_balance)
        return -1;
    else
        m_balance -= amount;
    return 0;
}
```

```
void withdraw(int amount) {
    if (amount > m_balance) {
        throw -1;
    }
    m_balance -= amount;
}

try {
    account.withdraw(100000);
    cout << account.getBalance() << endl;
} catch(int errorId) {
    cout << errorId << endl;
}
```

예외를 사전확인으로 바꾸기 (Replace Exception with Precheck)

You are throwing a checked exception on a condition the caller could have checked first.

*Change the caller to make **the test first.***

```
double getValueForPeriod (int periodNumber) {
    return m_vector.at(periodNumber);
}

try {
    cout << mylist.getValueForPeriod(2) << endl;
}
catch (exception& e) {
    cout << e.what() << endl;
}
```

```
double getValueForPeriod (int periodNumber) {
    if (periodNumber >= m_vector.size())
        return 0;
    return m_vector.at(periodNumber);
}
```

상속 연관 기법

- 객체 지향 프로그래밍에서 가장 잘 알려진 특성인 상속과 관련된 리팩토링 기법. 다른 메커니즘과 마찬가지로 유용한 동시에 오용하기 쉬운 부분으로, 잘못 사용하고 있어도 이를 알아차리는 것이 쉽지 않은 부분.

  - 서브클래스 제거하기
  - 슈퍼클래스 추출하기
  - 계층 합치기
  - 서브클래스를 위임으로 바꾸기
  - 슈퍼클래스를 위임으로 바꾸기
  - (category로...)
  - 메서드 올리기
  - 필드 올리기
  - 생성자 본문 올리기
  - 메서드 내리기
  - 필드 내리기
  - 타입 코드를 서브클래스로 바꾸기

메서드 올리기(Pull Up Method) (<-> Push Down Method)

You have methods with identical results on subclasses.

Move them to the superclass.

```
class Employee { ... }

class SalesPerson : public Employee {
    string getId() {  ... }
}
class Engineer : public Employee {
    string getId() {  ... }
}
```

```
class Employee {
    string getId() {  ... }
}

class SalesPerson : public Employee { ... }
class Engineer : public Employee { ... }
```

- Related to

  Form Template Method

- **Smells**

  Duplication Code

필드 올리기(Pull Up Field) (<-> Push Down Field) [상속부분 #2]

Two subclasses have the same field.

Move the field to the superclass.

```
class Employee { ... }

class SalesPerson : public Employee {
private:
    string id;
}
class Engineer : public Employee {
private:
    string id;
}
```

```
class Employee {
private:
    string id;
}

class SalesPerson : public Employee { ... }
class Engineer : public Employee { ... }
```

Smells

Duplication Code

생성자 본문 올리기(Pull Up Constructor Body)

You have constructors on subclasses with **mostly identical** bodies.

Create a superclass constructor; call this from the subclass methods.

```
class Party { ... }

class Employee : public Party {
public:
    Employee(int id, string name, int monthlyCost):
        m_id(), m_name(name), m_monthlyCost(monthlyCost)
    {}
};
```

```
class Party {
public:
    Party(){}
    Party(string name):
        m_name(name){
    }
};

class Employee : public Party {
public:
    Employee(int id, string name, int monthlyCost):
        m_id(),  m_monthlyCost(monthlyCost){
            Party(name);
    }
};
```

- Related to

  Pull Up Method

- **Smells**

  Duplication Code

메서드 내리기(Push Down Method) (<-> Pull Up Method) [상속부분 #4]

Behavior on a superclass is relevant only for some of its subclasses.

Move it to those subclasses.

```
class Employee {
    string getQuota() {  ... }
}

class SalesPerson : public Employee { ... }
class Engineer : public Employee { ... }
```

```
class Employee { ... }

class SalesPerson : public Employee {
    string getQuota() {  ... }
}
class Engineer : public Employee {
    string getQuota() {  ... }
}
```

- Related to

  Push Down Field, Extract Subclass

- **Smells**

  Refused Bequest

필드 내리기(Push Down Field) (<-> Pull Up Field) [상속부분 #5]

A field is used only by some subclasses.

Move the field to those subclasses.

```
class Employee {
private:
    string quota;
}

class SalesPerson : public Employee { ... }
class Engineer : public Employee { ... }
```

```
class Employee { ... }

class SalesPerson : public Employee {
private:
    string quota;
}
class Engineer : public Employee {
private:
    string quota;
}
```

- Related to

  Push Down Method, Extract Subclass

- **Smells**

  Refused Bequest

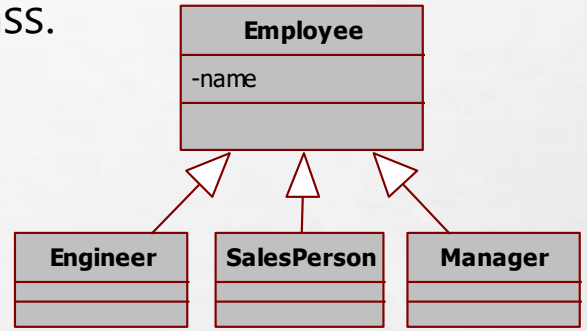
타입 코드를 서브클래스로 바꾸기 (Replace Type Code with Subclasses) (<-> Remove Subclass) (1/2)

[상속부분 #6]

You have an **immutable type code** that affects the behavior of a class.

*Replace the **type code with subclasses.***

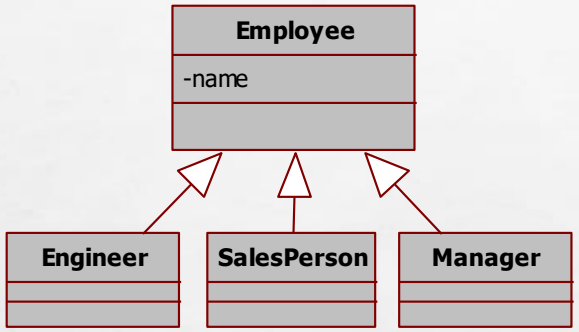
- *Polymorphism*



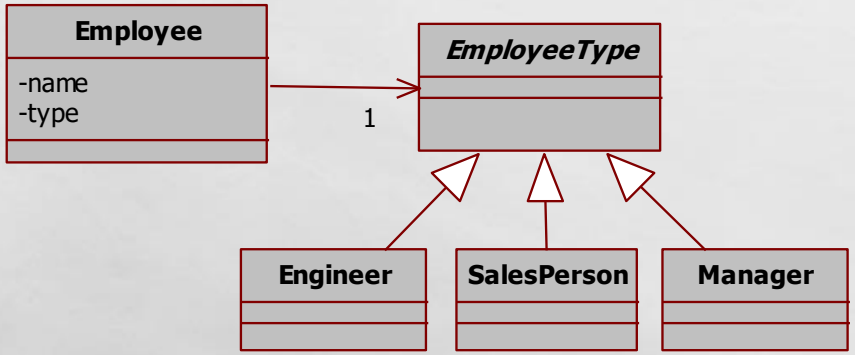

```
Employee ceateEmp(string type, string name){
    return Employee(type, name);
}

class Employee {
public:
    Employee(string type, string name):
        m_type(type), m_name(name){
    }

private:
    int m_type;
    string m_name;
};
```

```
shared_ptr<Employee> aEmployee = ceateEmp(type,name);
shared_ptr<Employee> ceateEmp(int type, string name){
    switch(type) {
        case ENGINEER:
            return make_shared<Engineer>(name);
        case SALESPERSON:
            return make_shared<SalesPerson>(name);
        case MANAGER:
            return make_shared<Manager>(name);
        default:
            throw(type);
    }
}

class Engineer : public Employee { ... }
class SalesPerson : public Employee { ... }
class Manager : public Employee { ... }
```

타입 코드를 서브클래스로 바꾸기 (Replace Type Code with Subclasses) (<-> Remove Subclass) (2/2)



- *Delegation*



[상속부분 #6]

```
Employee aEmployee = Employee(type);

Employee::Employee(int arg)
{ setType(arg); };

void Employee::setType(int type) {
    m_type = EmployeeType::create(type);
}

class EmployeeType {
public:
    static shared_ptr<EmployeeType> create(int);
};

shared_ptr<EmployeeType> EmployeeType::create(int type) {
    if( type == EmployeeType::ENGINEER )
        return std::make_shared<Engineer>();
    if( type == EmployeeType::SALESMAN )
        return std::make_shared<Salesman>();
    if( type == EmployeeType::MANAGER )
        return std::make_shared<Manager>();

    throw TypeException();
}

class Engineer : public EmployeeType { ... }
class SalesPerson : public EmployeeType { ... }
class Manager :public EmployeeType { ... }
```

Smells

Primitive Obsession, Repeated Switches, Large Class

서브클래스 제거하기(Remove Subclass)(<-> Replace Type Code with Subclasses)

A subclass that does too little incurs a cost in understanding that is no longer worthwhile.

Change the methods to superclass fields and eliminate the subclasses.

```
class Person {
public:
    virtual int genderCode();
}
class Male : public Person {
public:
    virtual int genderCode() { return MALE;}
}
class Female public Person {
public:
   virtual int genderCode() {return FEMALE; }
}
```

```
class Person {
public:
    int genderCode() {return m_genderCode;
private:
    int m_genterCode;    // MALE or FEMALE}
}
```

슈퍼클래스 추출하기(Extract Superclass) [상속부분 #8]

You have **two classes with similar features.**

Create a superclass and move the common features to the superclass.

```
class Department {
    double totalAnnualExpenses() {...}
    string name() {...}
    int headCount() {...}
}
class Employee {
    double annualExpenses() {...}
    string name() {...}
    string id() {...}
}
```

```
class Party {
    string name() {...}
    double annualExpenses() {...}
}

class Department : public Party {
    double annualExpenses() {...}
    int headCount() {...}
}
class Employee : public Party {
    double annualExpenses() {...}
    string id() {...}
}
```

Smells

Large Class, Alternative Classes with Different Interfaces

계층 합치기(Collapse Hierarchy) [상속부분 #9]

A superclass and subclass are **not very different.**

***Merge** them together.*

```
class Employee {
    // A
}
class Salesman : public Employee {
    // B
}
```

```
class SalesEmployee {
    // A+B
}
```

- Related to

  Inline Class

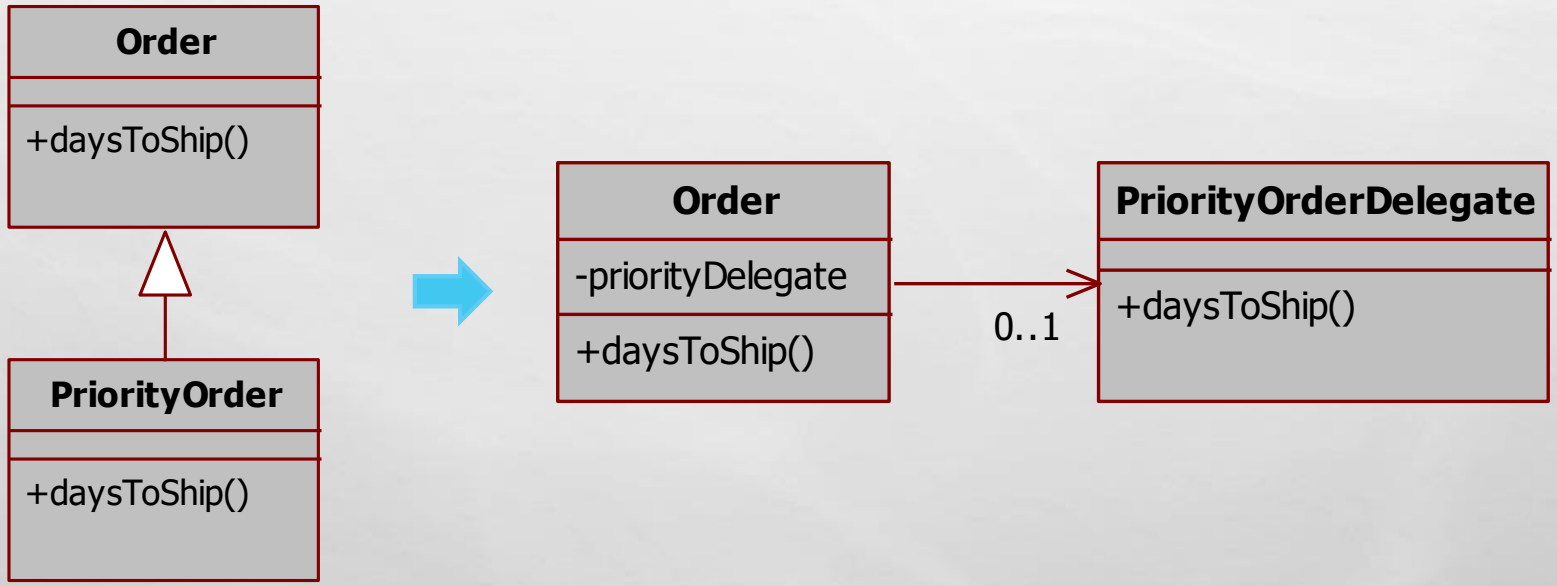
- **Smells**

  Lazy Element, **Speculative Generality**

서브클래스를 위임으로 바꾸기(1/2)(Replace Subclass with Delegation)[상속부분 #10]

You have an object **whose behavior more than one reason** to vary, or **very close relationship** between cla sses in same inheritance.

*Create a class for **delegate,** move functions to delegate then **remove the subclasses.***



서브클래스를 위임으로 바꾸기(2/2)(Replace Subclass with Delegation)


```
class Order {
public:
    int daysToShip() {
        return warehouse.daysToShip;
    }
private:
    Warehouse warehouse;
};

class PriorityOrder : public Order {
public:
    int daysToShip() {
        return priorityPlan.daysToShip;
    }
private:
    PriorityPlan priorityPlan;
};
```

```
class Order {
public:
    int daysToShip() {
        return (priorityDelegate != nullptr ? priorityDelegate->daysToShip() : warehouse.daysToShip);
    }
    PriorityOrderDelegate *priorityDelegate = nullptr;
private:
    Warehouse warehouse;
};

class PriorityOrderDelegate{
public:
    int daysToShip() {
        return priorityPlan.daysToShip;
    }
private:
    PriorityPlan priorityPlan;
};
```

- **Smells**

  Middle Man, Insider Trading, Refused Bequest

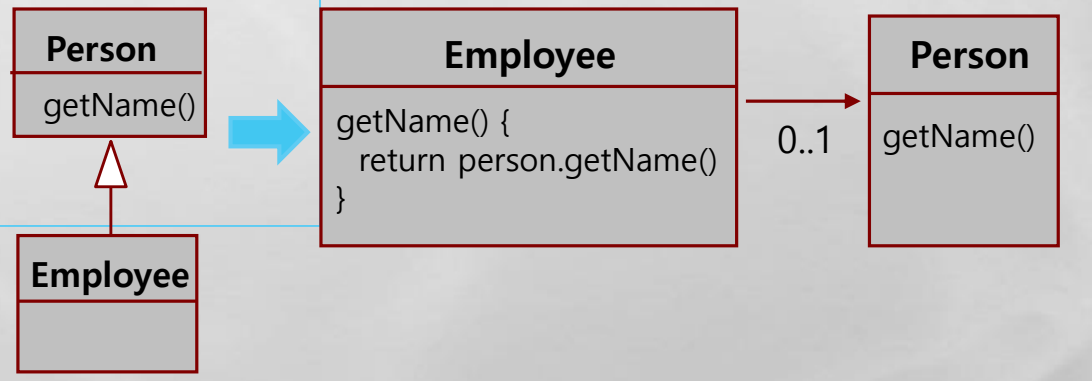
슈퍼클래스를 위임으로 바꾸기(Replace Superclass with Delegation)

A subclass uses only a **part of a superclass interface** or does not want to inherit superclass data.

Create a field for the superclass, adjust methods to delegate to the superclass, and remove inheritance.

```
class Person{
    getName(){...}
}

class Employee : public Person {...}
```

```
class Person{...}

class Employee{
public:
    Employee(const string name)     {
        m_person = Person(name);
    }
    string getName() {
        return m_person.getName();
    }
private:
    Person m_person;
};
```



Smells

Middle Man, Insider Trading, Refused Bequest

위임을 상속으로 바꾸기(Replace Delegation with Inheritance)

A class contains many simple methods that delegate to all methods of another class.

Make the class a delegate inheritor, which makes the delegating methods unnecessary.

```
class Person{...}

class Employee{
public:
    Employee(const string name)     {
        m_person = Person(name);
    }
    string getName() {
        return m_person.getName();
    }
private:
    Person m_person;
};
```

```
class Person{
   getName(){...}
}

class Employee : Person {...}
```

Smells

Middle Man, Insider Trading, Refused Bequest

Conclusion

- Object Calisthenics : 9 steps to better software design today, by Jeff Bay
  1. One level of indentation per method : loop, long function
  2. Don't use the ELSE keyword : long function
  3. Wrap all primitives and Strings : primitive obsession
  4. First class collections : collection 의 불변성, 가독성, 관리 용이
  5. One dot per line : message chain
  6. Don't abbreviate : mysterious name
  7. Keep all entities small : large module
  8. No classes with more than two instance variables : class cohesion 높인다.
  9. No getters/setters/properties : Data class

- "Rather than construction,

  Software is more like Gardening - it is more organic then concrete" *- Pragmatic Programmer -*

More Information

- Michael Feathers, 레거시 코드 활용전략, 2004
- Scott Ambler, Pramod Sadalage, 리팩토링 데이터베이스, 2006
- Joshua Kerievsky, 패턴을 활용한 리팩토링, 2005

- Quick Reference Guide (Smell to Refactoring)
  - https://refactoring.guru/files/refactoring-cheat-sheet.pdf
- Changes in 2nd Ed. (Martin Fowler, Refactoring Improving the Design of Existing Code)
  - https://martinfowler.com/articles/refactoring-2nd-changes.html
- Code Refactoring in IntelliJ IDEA
  - https://www.jetbrains.com/help/idea/refactoring-source-code.html

참고 문헌

- Martin Fowler, Refactoring Improving the Design of Existing Code 2nd Ed., 2020
- www.refactoring.guru/ , www.refactoring.com/
- William C. Wake, Refactoring Workbook, 2001
- Robert C Martin, Clean Code, 2013
- Kent Beck, Test-Driven Development By Example New Ed., 2014
- Andrew Hunt, The Pragmatic Programmer : Your Journey to master

C 언어 refactoring

-Test-Driven Development

for Embedded C, Jammes W.Grenning

Code smell

- Duplicate Code
- Bad Names
- Bad Pasta
- Long Function
- Abstraction Distraction
- Bewildering Boolean
- Swith Case Disgrace
- Duplicate Swith Case
- Nefarious nesting
- Feature Envy
- Long parameter List
- Willy-Nilly Initialization
- Gobal Free-for All
- Comments
- Commented-Out Code
- Conditional Compilation

Code smell

Bad Names

- C 언어 프로그램에 전체 단어나 모음을 사용하는 것이 가혹한가?

- 코드를 처음 읽기에도 다시 읽기에도 쉽도록
  - 읽을 수 있는 이름: Avoid abbreviation and acronyms.

    예) lht_sched => LightScheduler

  - 내부 작업이 아닌 의도된 결과

    예) BinarySearch => Find, BinarySearchForScheduledEvent => FindScheduledEvent

    * library 로는 BinarySearch도 가능

Code smell

Long Function

- make them smaller.
- 새로운 기능이 추가되어 길어지기 시작하면 새로운 function으로 분리하라.
- 블럭에 주석을 추가하는 대신 설명하는 이름을 사용하여 코드를 이동하라.

  => “Extract Method”

Abstraction Distraction

- 각 function은 일관된 레벨의 추상화
- Primitive Obsession에 의한 high-level idea를 잃지 않도록
- Function extraction으로 Long function smell 제거

Code smell

Bewildering Boolean

- 해석하는 데 얼마나 걸리나요

```
if (!(day == EVERYDAY || day == today
    ||(day == WEEKEND && (SATURDAY == today
    || SUNDAY == today)) || (day == WEEKDAY
    && today >= MONDAY && today <= FRIDAY)))
    return;
```

```
if (!matchesToday(day))
    return;
```

Switch Case Disgrace

- Determining the case and then doing something simple or delegating to something to do the work.

Code smell

Duplicate Switch Case

- Switch Case 가 중복되지만 다른 동작을 수행한다면

=> Open/Closed principle 을 적용하여

```
void LightController_TurnOn(int id)
{
    LightDriver driver = lightDrivers[id];
    if(NULL == driver)
        return;

    switch(driver->type)
    {
    case X10:
        X10LightDriver_TurnOn(driver);
        break;
    case AcmeWireless:
        AcmeWirelessLightDriver_TurnOn(driver);
        break;
```

```
void LightController_TurnOn(int id)
{
    LightDriver_TurnOn(lightDrivers[id];
}
```

Code smell

- Nefarious Nesting
  - nested conditional logic이 있는 loop가 있다면

  => 내포 된 것을 함수로 만드는 것을 고려

- Feature Envy
  - C에서는 특히 Data structure가 globally 전달되거나 접근될때

  => OO concepts과 Multiple-instance Module을 사용하여 모듈성을 향상, 중복을 줄임.

Code smell

- Long Parameters List
  - 어느 정도가 long? => 때에 따라 다르다?
  - 동일한 파라미터가 여러 function signature에 중복된다면

  => 새로운 data structure

  => Module의 initialize function

  => 새로 정의된 구조의 pointer

- Willy-Nilly Initialization
  - initialization and running의 구분이 없다

  =>연관된 initialization code를 한 곳에 모아 분명히 한다.

Code smell

- Global Free-For-All
  - Willy-Nilly Initialization, strong coupling 문제 발생

  => Function 내로 encapsulation을 고려

  => Module의 initialize function

  => Global data가 structure 라면 abstract data type으로 변환을 고려

- Commented-Out Code

  =>삭제한다.(언제든지 복구 가능)

Code smell

Comments

- 리팩터링의 목표는 코드가 스스로 말하는 체계화된 코드
- 언제 사용해야할까?

=>좋은 이름과 구조로도 명확하지 않을 때.

=> API 영역, 최적화 등으로 불분명해진 경우,

- 기존에 있는 Comment는?

=> 기존 설명을 코드를 재구성하는 방법에 대한 힌트로 사용하고, 좋은 이름의 함수로 대체하고, 무의미한 코멘트를 삭제한다.

Embeded C refactoring 예제

<LightScheduler_WakeUp >

```
void LightScheduler_WakeUp(void){
    int i;
    Time time;
    TimeService_GetTime(&time);
    Day td = time.dayOfWeek;
    int min = time.minuteOfDay;
    for (i = 0; i < MAX_EVENTS; i++)
    {
        ScheduledLightEvent * se = &eventList[i];
        if (se->id != UNUSED)
        {
            Day d = se->day;
            if ( (d == EVERYDAY) || (d == td) || (d == WEEKEND &&
                    (td == SATURDAY || td == SUNDAY)) ||
                    (d == WEEKDAY && (td >= MONDAY
                                && td <= FRIDAY)))
            {
                    /* it's the right day */
                    if (min == se->minuteOfDay + se->randomMinutes)
                    {
                        if (se->event == TURN_ON)
                            LightController_TurnOn(se->id);
                        else if (se->event == TURN_OFF)
                            LightController_TurnOff(se->id);
                        if (se->randomize == RANDOM_ON)
                            se->randomMinutes = RandomMinute_Get();
                        else
                            se->randomMinutes = 0;
                    }
                }
            }
        }
    }
}
```

=> 1. 함수추출하기

Embeded C refactoring 예제

<processEventsDueNow >

```
static void processEventsDueNow(Time * time, ScheduledLightEvent * event){
    Day today = time->dayOfWeek;
    int minuteOfDay = time->minuteOfDay;
    if (event->id != UNUSED)
    {
        Day day = event->day;
        /* if (isEventDueNow()) */
        if ( (day == EVERYDAY) || (day == today) || (day == WEEKEND &&
                (today == SATURDAY || today == SUNDAY)) ||
                (day == WEEKDAY && (today >= MONDAY
                        && today <= FRIDAY)))
      {
                /* it's the right day */
                 if (min == se->minuteOfDay + se->randomMinutes)
                {
                    if (se->event == TURN_ON)
                        LightController_TurnOn(se->id);
                    else if (se->event == TURN_OFF)
                        LightController_TurnOff(se->id);
                    if (se->randomize == RANDOM_ON)
                        se->randomMinutes = RandomMinute_Get();
                    else
                        se->randomMinutes = 0;
                }
            }
        }
    }
}
```

=> 2. 함수추출하기(IsEventDueNow, operateLight, resetRandomize)

Embeded C refactoring 예제

<processEventsDueNow >

```
static void processEventsDueNow(Time * time, ScheduledLightEvent * event){
    if (event->id != UNUSED)
    {
        if (isEventDueNow(time, event))
        {
            operateLight(event);
            resetRandomize(event);
        }
    }
}
```

=> 3. 추상화 레벨 평탄화

Embeded C refactoring 예제

<processEventsDueNow >

```
static BOOL isInUse(ScheduledLightEvent * event){
    return event->id != UNUSED;
}

static void processEventsDueNow(Time * time, ScheduledLightEvent * event){
    if (isInUse(event))
    {
        if (isEventDueNow(time, event))
        {
            operateLight(event);
            resetRandomize(event);
        }
    }
}
```

```
if (!isInUse(event))
        return;

if (isEventDueNow(time, event))
{
    operateLight(event);
    resetRandomize(event);
}
```

Embeded C refactoring 예제

<isEventDueNow >

```
static BOOL isEventDueNow(Time * time, ScheduledLightEvent * event){
    Day today = time->dayOfWeek;
    int minuteOfDay = time->minuteOfDay;
    Day day = event->day;

    if ( (day == EVERYDAY) || (day == today)
            || (day == WEEKEND &&
            (today == SATURDAY || today == SUNDAY))
            || (day == WEEKDAY && (today >= MONDAY
            && today <= FRIDAY)))
    {
        if (minuteOfDay != event->minuteOfDay + event->randomMinutes)
            return TRUE;
    }
    return FALSE;
```

=> 4. 중첩 조건 분리

Embeded C refactoring 예제

<isEventDueNow >

```
static BOOL isEventDueNow(Time * time, ScheduledLightEvent * event){
    Day today = time->dayOfWeek;
    int minuteOfDay = time->minuteOfDay;
    Day day = event->day;

    if (minuteOfDay != event->minuteOfDay + event->randomMinutes)
            return FALSE;

    if ( (day == EVERYDAY) || (day == today)
            || (day == WEEKEND &&
            (today == SATURDAY || today == SUNDAY))
            || (day == WEEKDAY && (today >= MONDAY
            && today <= FRIDAY)))
    {
        return TRUE;
}
```

=> 4. 함수 추출(daysMatch)

Embeded C refactoring 예제

<isEventDueNow >

```
static BOOL daysMatch(Day scheduledDay, Day today){
    if ((day == EVERYDAY) || (day == today)
            || (day == WEEKEND &&
               (today == SATURDAY || today == SUNDAY))
            || (day == WEEKDAY && (today >= MONDAY
            && today <= FRIDAY)))
        return TRUE;
    return FALSE;
}

static BOOL isEventDueNow(Time * time, ScheduledLightEvent * event){
    Day today = time->dayOfWeek;
    int minuteOfDay = time->minuteOfDay;
    Day day = event->day;
    if (minuteOfDay != event->minuteOfDay + event->randomMinutes)
        return FALSE;
    if (!daysMatch(today, day))
        return FALSE;    return TRUE;
}
```

=> 5. daysMatch 정리

Embeded C refactoring 예제

<daysMatch >

```
static BOOL daysMatch(Day scheduledDay, Day today){
    if (scheduledDay == EVERYDAY)
        return TRUE;
    if (scheduledDay == today)
        return TRUE;
    if (scheduledDay == WEEKEND && (today == SATURDAY || today == SUNDAY))
        return TRUE;
    if (scheduledDay == WEEKDAY && (today >= MONDAY && today <= FRIDAY))
        return TRUE;
    return FALSE;
}
```

=> 6. LightScheduler 보다는 TimeService와 연관도를 높임.

Embeded C refactoring 예제

<Time_MatchesDayOfWeek >

```
BOOL Time_MatchesDayOfWeek(Time * time, Day day){
    int today = time->dayOfWeek;
    if (day == EVERYDAY)
        return TRUE;
    if (day == today)
        return TRUE;
    if (day == WEEKEND && (today == SATURDAY || today == SUNDAY))
        return TRUE;
    if (day == WEEKDAY && today >= MONDAY && today <= FRIDAY)
        return TRUE;
    return FALSE;
}
static BOOL isEventDueNow(Time * time, ScheduledLightEvent * event){
    Day today = time->dayOfWeek;
    int minuteOfDay = time->minuteOfDay;
    Day day = event->day;
    if (minuteOfDay != event->minuteOfDay + event->randomMinutes)
        return FALSE;
    if (!Time_MatchesDayOfWeek(time, day))
        return FALSE;
}
```

=> 7. minuteOfDay 도 TimeService로

Embeded C refactoring 예제

<Time_MatchesDayOfWeek >

```
BOOL Time_MatchesMinuteOfDay(Time * time, int minuteOfDay){
     return time->minuteOfDay == minuteOfDay;
}

static BOOL isEventDueNow(Time * time, ScheduledLightEvent * event){
    int todaysMinute = event->minuteOfDay + event->randomMinutes;
    Day day = event->day;

    if (!Time_MatchesMinuteOfDay(time, todaysMinute))
        return FALSE;
    if (!Time_MatchesDayOfWeek(time, day))
        return FALSE;
}
```

=> 8. 파일 time.c 로 분리

참고 문헌

- James W.Grenning, Test-Driven Development for Embedded C, 2011

THANK YOU.