오브젝트

CHAPTER 7. 객체 분해

> 추상화에 대한 이야기
> 과거에 사용하던 프로시저 추상화의 문제점을 이해하고 데이터 추상화와 타입 추상화/프로시저 추상화의 차이를 이해하자

- 사람의 단기 기억 정보 갯수는 매우 한정적(5~9개, 조지 밀러의 매직넘버 7 - 7 ±2 규칙)
  장기 기억에는 직접 접근하지 못하고 단기 기억 영역으로 옮긴 후에 처리 가능
- 어떤 문제 해결을 위해서 필요한 정보를 먼저 단기 기억으로 가져오는데, 필요한 요소의 수가 단기 기억 용량을 초과하는 순간 문제해결 능력이 급격하게 떨어지게 됨 ⇢ **인지 과부하**
- 이런 인지 과부하를 방지하는 가장 좋은 방법은 단기 기억안에 보관할 정보의 양을 조절하는 것
  한 번에 다뤄야 하는 정보의 수를 줄이기 위해 **본질적인 정보(핵심)**만 남기고 불필요한 세부 사항을 걸러내며 문제를 단순화하기 ⇢ ***추상화***

분해(decomposition)

- 가장 일반적인 추상화 방법
- 한 번에 해결하기 어려운 문제를 해결 가능한 작은 문제로 나누는 작업
- 목적은 인지 과부하의 부담 없이 단기 기억 안에서 한 번에 처리할 수 있는 규모의 문제로 나누는 것
- 조지 밀러의 단기기억 정보 매직넘버 7 은 가장 작은 단위로서의 개별 항목이 아니라 ***하나의 단위로 취급될 수 있는 논리적인 청크(chunk)를 의미***

  청크는 더 작은 청크를 포함할 수 있으며 연속적으로 분해 가능

  예를 들어,

  - 임의로 조합된 11자리 정수 8개를 한꺼번에 외우는 것 ⇢ 듣기만해도 어려움 🤯
  - 전화번호라는 개념적 청크로 묶어 8명에 대한 전화번호를 기억하는 것 ⇢ 가능 할 듯, 인지능력 향상 😎

결론

- 추상화와 분해는 인간이 세계를 인식하고 반응하기 위해 사용하는 가장 기본적인 사고 도구
- 추상화를 더 큰 규모의 추상화로 압축시킴으로써 단기 기억의 한계를 초월 가능
- 복잡성이 존재하는 곳에 추상화와 분해 역시 함께 존재

  따라서, 가장 복잡한 분야 중 하나인 소프트웨어 개발 영역에서도 문제를 해결하기 위해 사용되어 왔음

1. 프로시저 추상화와 데이터 추상화

프로그램 개발 세계의 추상화의 발전

- 프로그램 개발의 복잡성을 극복하려는 개발자들의 노력
  ⇢ 프로그래밍 언어의 발전
  ⇢ 프로그래밍 언어를 통해 표현되는 추상화의 발전
  ⇢ ***다양한 프로그래밍 패러다임의 탄생***

프로그래밍 패러다임

프로그래밍 패러다임은 아래 두 가지 요소로 결정되며, 모든 프로그래밍 패러다임을 (추상화와 분해의 관점에서) 설명할 수 있다.

- 프로그래밍을 구성하기 위해 사용하는 **추상화** 의 종류
- 추상화를 이용해 소프트웨어를 **분해** 하는 방법

현대적인 프로그래밍 언어의 두 가지 추상화 매커니즘

- 프로시저 추상화(procedure abstraction) : 소프트웨어가 무엇을 해야 하는지를 추상화 ⇢ 데이터 조작
- 데이터 추상화(data abstraction) : 소프트웨어가 무엇을 알아야 하는지를 추상화 ⇢ 정보 표현

소프트웨어는 데이터를 이용해 정보를 표현하고 프로시저를 이용해 데이터를 조작한다.

시스템 분해 방법

현대의 설계 방법에 중요한 영향을 끼치는 프로그래밍 패러다임들은 프로시저 추상화나 데이터 추상화를 중심으로 시스템의 분해 방법을 설명한다.

- 프로시저 추상화를 중심으로 시스템을 분해 ⇢ 기능 분해(functional decomposition) / 알고리즘 분해(algorithmic decomposition)
- 데이터 추상화를 중심으로 시스템을 분해
  - 데이터를 중심으로 **타입을 추상화(type abstraction)** ⇢ 추상 데이터 타입(Abstract Data Type)
  - 데이터를 중심으로 **프로시저를 추상화(procedure abstraction)** ⇢ 객체지향(Object-Oriented)

> 꾸준히 이야기했던 내용인,'객체지향은 역할과 책임을 수행하는 자율적인 객체들의 협력 공동체를 구축하는 것'에서
> '역할과 책임을 수행하는 객체' 이 부분이 객체지향 패러다임이 이용하는 추상화이다.
>
> 기능을 '협력하는 공동체'를 구성하도록 객체들로 나누는 과정이 바로 ***객체지향 패러다임에서의 분해를 의미*** 한다.

프로그래밍 언어의 관점에서 객체지향

기능구현을 위해 필요한 객체를 식별 ⇢ 협력 가능하도록 시스템을 분해 ⇢ 프로그래밍 언어로 실행 가능한 프로그램을 구현
이런 객체를 구현하기 위해 대부분의 객체지향 언어가 제공하는 도구 ⇢ ***클래스***

- 프로그래밍 언어의 관점에서 객체지향이란 데이터를 중심으로 데이터 추상화와 프로시저 추상화를 통합한 객체를 이용해 시스템을 분해하는 방법

  구체적으로, 데이터 추상화와 프로시저 추상화를 함께 포함한 클래스를 이용해 시스템을 분해하는 것

앞으로의 내용

'전통적인 기능 분해 방법보다 객체지향이 효과적' 이라고 말하는 이유와 효과적이란 의미에 대해서 알아보자

전통적인 기능 분해 방법에서 객체지향 분해 방법까지 좌절과 극복의 역사를 살펴본다

2. 프로시저 추상화와 기능 분해

2-1. 메인 함수로서의 시스템

> 기능과 데이터 중 기능은 오랜시간 동안 시스템을 분해하기 위한 기준으로 사용되었으며, 이 같은 시스템 분해 방식을 **알고리즘 분해** 또는 **기능 분해** 라고 부른다.
> 기능 분해의 관점에서 추상화의 단위는 프로시저이며 시스템은 프로시저를 단위로 분해된다.

기능 분해/알고리즘 분해

- 기능이 시스템을 분해하기 위한 기준
- 추상화의 단위는 프로시저
- 시스템을 프로시저 단위로 분해

프로시저

- 반복적으로 실행되거나 거의 유사하게 실행되는 작업들을 하나의 장소에 모아놓음으로써 로직을 재사용하고 중복을 방지할 수 있는 추상화 방법
- 프로시저를 추상화라고 부르는 이유는 내부의 구현 내용을 모르더라도 인터페이스만 알면 프로시저를 사용할 수 있기 때문

  따라서, 프로시저는 잠재적으로 정보은닉(information hiding)의 가능성을 제시

  그러나, 프로시저만으로 효과적인 정보은닉 체계를 구축하는데는 한계가 존재

프로시저 중심의 기능 분해

- 입력 값을 계산해서 출력 값을 반환하는 수학의 함수와 동일

시스템은 필요한 더 작은 작업으로 분해될 수 있는 하나의 커다란 메인 함수

- 하향식 접근법(Top-Down Approach)을 따른다
  - 시스템을 구성하는 가장 최상위(topmost) 기능을 정의하고, 이 최상위 기능을 좀 더 작은 단계의 하위 기능으로 분해해 나가는 방법
  - 분해는 세분화된 마지막 하위 기능이 프로그래밍 언어로 구현 가능한 수준이 될 때까지 계속된다
  - 각 세분화 단계는 바로 위 단계보다 더 구체적이어야 한다
    같은 말로, 정제된 기능은 자신의 바로 상위 기능보다 덜 추상적이어야 한다
  - 상위 기능은 하나 이상의 더 간단하고 더 구체적이며 덜 추상적인 하위 기능의 집합으로 분해된다

2-2. 급여 관리 시스템

기능 분해 방법으로 급여 관리 시스템을 분해

- 최상위의 추상적인 함수 정의에서 출발해 단계적인 정제 절차를 따라 시스템을 구축
- 최상위의 추상적인 함수 정의는 시스템의 기능을 표현하는 하나의 문장으로 나타내며
  이 문장을 구성하는 세부적인 단계의 문장으로 분해해 나가는 방식을 따른다
- 하나의 문장으로 표현된 기능을 여러개의 더 작은 기능으로 분해하는 것

1. 최상위 문장 기술

   > 직원의 급여를 계산한다

   ⇢ 메인 프로시저

2. 세분화된 절차로 구체화

   > 직원의 급여를 계산한다
   > 사용자로부터 소득세율을 입력받는다
   > 직원의 급여를 계산한다
   > 양식에 맞게 결과를 출력한다

3. 각 정제 단계는 이전 문장의 추상화 수준을 감소시켜야 하며,
   각 단계에서 불완전하고 좀 더 구체화 될 수 있는 문장들이 남아있는지 검토하고, 구현이 가능할 정도로 충분히 저수준의 문장이 될 때까지 기능을 분해

   > 직원의 급여를 계산한다
   > **사용자로부터 소득세율을 입력받는다**
   > "세율을 입력하세요: "라는 문장을 화면에 출력한다
   > 키보드를 통해 세율을 입력받는다
   > **직원의 급여를 계산한다**
   > 전역 변수에 저장된 직원의 기본급 정보를 얻는다
   > 급여를 계산한다
   > **양식에 맞게 결과를 출력한다**
   > "이름: {직원명}, 급여: {계산된 금액}" 형식에 따라 출력 문자열을 생성한다

- 기능 분해의 결과는 최상위 기능을 수행하는데 필요한 절차들을 실행되는 시간 순서에 따라 나열한 것이다

- 기본적으로 기능 분해는 책의 목차를 정리하고 그 안에 내용을 채워 넣는 것과 유사하다
- 위의 굵게 표시된 내용이 목차가 되는 것
- 필요하다면 이들 정보는 그 자체로 하위 단계의 목차가 될 수 있으며 그 안에 좀 더 구체적인 내용을 포함할 수 있다
- 급여 관리 시스템을 ***입력을 받아 출력을 생성하는 커다란 하나의 메인 함수로 간주*** 하고 기능 분해를 시작했다는 점에 주목하라
  - 입력 정보 - 직원정보, 소득세율
  - 출력 - 계산된 급여 정보
- 기능을 중심으로 필요한 데이터를 결정한다
  - 데이터는 기능을 보조하는 조연 역할
- 기능 분해를 위한 하향식 접근법은 먼저 필요한 기능을 생각하고 이 기능을 분해하고 정제하는 과정에서 필요한 데이터의 종류와 저장 방식을 식별한다

문제점

- 기능 분해 방법은 유지보수에 다양한 문제를 야기

  하향식 기능 분해 방식의 문제점을 이해하는 것은 유지보수 관점에서 객체지향의 장점을 이해할 수 있는 좋은 출발점

2-3. 급여 관리 시스템 구현

ruby 로 구현 (pseudo code 라고 생각해도 무방)

```
def main(name)                                  # 직원의 급여를 계산한다
  taxRate = getTaxRate()                            # 사용자로부터 소득세율을 입력받는다
  pay = calculatePayFor(name, taxRate)              # 직원의 급여를 계산한다
  puts(describeResult(name, pay))                   # 양식에 맞게 결과를 출력한다
end

def getTaxRate()
  PRINT("세율을 입력하세요: ")                        # 문장을 화면에 출력한다
  return gets().chomp().to_f()                      # 키보드를 통해 세율을 입력받는다
end

# ruby 의 전역변수의 이름은 반드시 '$'로 시작해야 한다
$employees = ["직원A", "직원B", "직원C"]
$basePays = [400, 300, 250]

def calculatePayFor(name, taxRate)
  index = $employees.index(name)                    # 전역 변수에 저장된 직원의 기본급 정보를
얻는다
  basePay = $basePays[index]                        # 급여를 계산한다
  return basePay - (basePay * taxRate)
end
```

```
def describeResult(name, pay)
  return "이름: #{name}, 급여: #{pay}"          # 형식에 따라 출력 문자열을 생성한다
end
```

이름이 "직원C"인 직원의 급여를 계산하려면 다음과 같이 프로시저를 호출

```
main("직원C")
```

하향식 기능분해

- 시스템을 최상위의 가장 추상적인 메인 함수로 정의하고, 메인 함수를 구현 가능한 수준까지 세부적인 단계로 분해하는 방법
- 메인 함수를 루트로 하는 '트리(tree)'로 표현할 수 있다
  - 각 노드(node)는 시스템을 구성하는 하나의 프로시저를 의미하고 한 노드의 자식 노드는 부모 노드를 구현하는 절차중의 한 단계를 의미
  - p.225 그림 7.2 참고 `트리 형태로 표현한 급여 관리 시스템의 기능 분해 구조`
- 논리적이고 체계적인 시스템 개발 절차를 제시한다

  ⇢ 구조적이고 체계적이어서 이상적인 방법으로 보이지만 변화무쌍한 소프트웨어 안에서는 다양한 문제에 부딪힌다

2-4. 하향식 기능 분해의 문제점

- 시스템은 하나의 메인함수로 구성돼 있지 않다
- 기능 추가나 요구사항 변경으로 인해 메인 함수를 빈번하게 수정해야 한다
- 비즈니스 로직이 사용자 인터페이스와 강하게 결합된다
- 하향식 분해는 너무 이른 시기에 함수들의 실행 순서를 고정시키기 때문에 유연성과 재사용성이 저하된다
- 데이터 형식이 변경될 경우 파급효과를 예측할 수 없다

> **설계는 코드 배치 방법이며 설계가 필요한 이유는 변경에 대비하기 위한것** 이라는 점을 기억하라.
>
> 변경은 성공적인 소프트웨어가 맞이해야 하는 피할 수 없는 운명이다. 현재의 요구사항이 변하지 않고 코드를 변경할 필요가 없다면 소프트웨어를 어떻게 설계하건 아무도 신경쓰지 않을 것이다. 하지만 불행하게도 소프트웨어는 항상 변경된다.

하향식 접근법과 기능 분해가 가지는 근본적인 문제점

- **변경에 취약한 설계** 를 낳는다는 것

하나의 메인 함수라는 비현실적인 아이디어

- 어떤 시스템도 최초에 릴리즈됐던 당시의 모습을 그대로 유지하지 않음, 지속적으로 새로운 기능을 추가하게 된다

  ⇢ 시스템이 오직 하나의 메인 함수만으로 구현된다는 개념과 모순

- 어느 시점에 이르면 유일한 메인 함수라는 개념의 의미가 없어지고 여러개의 동등한 수준의 함수 집합으로 성장하게 된다
- 모든 기능은 기능성의 측면에서는 동등하게 독립적이고 완결된 하나의 기능을 표현한다
- 하나의 알고리즘을 구현하거나 배치 처리를 구현하기에는 적합하지만 현대적인 상호작용 시스템을 개발하는데는 적합하지 않다

  현대적인 시스템은 동등한 수준의 다양한 기능으로 구성된다 ⇢ ***하향식 접근법의 문제점***

메인 함수의 빈번한 재설계

- 시스템 안에는 여러개의 정상이 존재하기 때문에 결과적으로 하나의 메인 함수를 유일한 정상으로 간주하는 하향식 기능 분해의 경우에는 새로운 기능을 추가할 때마다 매번 메인 함수를 수정해야 한다

  기존 로직과는 아무런 상관이 없는 새로운 함수의 적절한 위치를 확보해야 하기 때문에 메인 함수의 구조를 급격하게 변경할 수 밖에 없는 것

  - 기존 코드 수정 ⇢ 새로운 버그를 만들어낼 확률이 높아짐
- 급여 관리 시스템에 회사에 속한 모든 직원들의 기본급의 총합을 구하는 기능을 추가해 달라는 새로운 요구사항이 접수됐다고 가정하자
  - 발생되는 문제 : 기존의 메인 함수에 새로운 함수를 넣을자리가 마땅치 않다
  - 구현 방법
    - 기존 main 에서 동작하던 로직을 새로운 calculatePay 함수를 만들어 옮긴다
    - main 함수에서 인자로 두 작업중 어느 것을 수행할지 지정하는 값을 받아 구현한다

```
# 새로운 요구사항
def sumOfBasePays()
  result = 0
  for basePay in $basePays
    result += basePay
  end
  puts(result)
end

# 기존 main 에서 동작하던 로직을 옮김
def calculatePay(name)
  taxRate = getTaxRate()
  pay = calculatePayFor(name, taxRate)
  puts(describeResult(name, pay))
end

# 변경된 main 함수
def main(operation, args={})
```

```
  case(operation)
  when :pay then calculatePay(args[:name])
  when :basePays then sumOfBasePays()
  end
end
```

- 동작 호출

```
# 기본급의 총합 호출
main(:basePays)

# 이름이 "직원A"인 직원의 급여를 계산하기 위한 호출
main(:pay, name:"직원A")
```

문제점

- 예제를 통해 하나의 정상(top)인 메인 함수에서 출발한다는 하향식 접근법의 기본 가정에 대한 문제를 알 수 있다
- 시스템은 여러개의 정상으로 구성되기 때문에 새로운 정상(함수)을 추가할 때마다 하나의 정상이라고 간주했던 main 함수의 내부 구현을 수정할 수 밖에 없다
  결과적으로 기존 코드의 빈번한 수정으로 인한 버그 발생 확률이 높아지기 때문에 시스템은 변경에 취약해질 수밖에 없다

비즈니스 로직과 사용자 인터페이스의 결합

- 하향식 접근법은 비즈니스 로직을 설계하는 초기 단계부터 입력 방법과 출력 양식을 함께 고민하도록 강요

  결과적으로 비즈니스 로직과 사용자 인터페이스 로직이 밀접하게 결합된다

  - 발생되는 문제 : 비즈니스 로직과 사용자 인터페이스가 변경되는 빈도가 다르며,

    사용자 인터페이스를 변경하는 경우 비즈니스 로직까지 변경에 영향을 받게 됨 ⇢ 변경에 불안정한 설계
- 변경 빈도
  - 사용자 인터페이스 : 시스템 내에서 가장 자주 변경됨
  - 비즈니스 로직 : 사용자 인터페이스에 비해 변경이 적음
- 기존 급여 관리 시스템의 사용자 인터페이스를 GUI 기반으로 변경한다고 가정하자
  - 발생되는 문제 : 전체 구조 재설계 필요

  문제점

  - 기능을 분해하는 과정에서 사용자 인터페이스의 관심사와 비즈니스 로직 관심사를 동시에 고려하도록 강요하기 때문에 `관심사의 분리` 라는 아키텍처 설계의 목적을 달성하기 어렵다

성급하게 결정된 실행 순서

- 하향식으로 기능을 분해하는 과정은 하나의 함수를 더 작은 함수로 분해하고, 분해된 함수들의 실행 순서를 결정하는 작업으로 요약할 수 있는데,

  이것은 설계를 시작하는 시점부터 시스템이 **무엇(what)** 을 해야하는지가 아니라 **어떻게(how)** 동작해야 하는지에 집중하도록 만든다
- 처음부터 구현을 염두에 두기 때문에 자연스럽게 함수들의 실행 순서를 정의하는 시간 제약(temporal constraint)을 강조한다

  함수들의 실행 순서를 미리 결정하지 않는 한 기능 분해를 진행할 수 없다
- 기능 분해 방식은 **중앙집중 제어 스타일의 형태**
  - 중요한 제어 흐름의 결정이 상위함수에서 이루어짐
  - 하위 함수는 상위 함수의 흐름에 따라 적절한 시점에 호출

  문제점
  - 중요한 설계 결정사항인 함수의 제어 구조가 **빈번한 변경의 대상** 인 점

    ⇢ 결과적으로 기능을 추가하거나 변경하는 작업은 매번 기존에 결정된 함수의 제어구조를 변경하게 만든다

  해결방법
  - 안정적인 논리적 제약(logical constraint)을 설계의 기준으로 삼는 것

    ⇢ 객체지향 설계 방식

    어떤 한 구성요소로 제어가 집중되지 않고 여러 객체들 사이로 제어 주체가 분산된다
- 함수의 재사용이 어렵다
  - 모든 함수는 상위 함수를 분해하는 과정에서 필요에 따라 식별되며, 상위 함수가 강요하는 문맥(context) 안에서만 의미를 갖기 때문
  - 함수가 재사용 가능하기 위해서는 상위 함수보다 더 **일반적** 이어야 한다

    하지만 하향식 접근법을 따를 경우 분해된 하위 함수는 항상 상위 함수보다 문맥에 더 종속적이다
- 모든 문제의 원인은 **결합도**
  - 함수는 상위 함수가 강요하는 문맥에 강하게 결합된다
  - 함수는 함께 절차를 구성하는 다른 함수들과 시간적으로 강하게 결합돼 있다
  - 강한 결합도는 시스템을 변경에 취약하게 만들고 이해하기 어렵게 만든다
  - 가장 큰 문제는 전체 시스템의 핵심적인 구조를 결정하는 함수들이 데이터와 강하게 결합된다는 점이다

데이터 변경으로 인한 파급효과

- 하향식 기능 분해의 가장 큰 문제점은 어떤 데이터를 어떤 함수가 사용하고 있는지를 추적하기 어렵다는 것
  - 데이터 변경으로 인해 어떤 함수가 영향을 받을지 예상하기 어렵다
  - 의존성과 결합도의 문제, 테스트의 문제
  - 데이터의 변경으로 인한 영향은 데이터를 직접 참조하는 모든 함수로 퍼져나간다
- 급여 관리 시스템에 새로운 기능으로 아르바이트 직원에 대한 급여 계산을 추가해 보자
  - 조건 : 아르바이트 직원은 일한 시간에 시급을 곱한 금액만큼을 지급받는다

- 구현 : 직원 정보를 관리하던 데이터 변수($employees, $basePays)를 수정하고, 업무 누적 시간정보 데이터($timeCards) 등도 추가한다
- 결과적으로 발생된 문제 : 기존의 직원들의 모든 기본급을 더한 sumOfBasePays 함수 결과가 변경되었다
  - 원인 : 직원 정보를 관리하던 데이터에 아르바이트 직원의 데이터가 포함되어 발생
  - 수정방안 : 아르바이트 직원의 시급을 총합에서 제외해야 함

확인 할 수 있는 것

- 시스템 안에 구현된 모든 함수를 분석해서 영향도를 파악하는 것은 쉽지 않다
- 코드가 성장하고 라인 수가 증가할수록 문제가 커진다

해결 방법

- 데이터와 함께 변경되는 부분과 그렇지 않은 부분을 명학하게 분리 한다

  데이터와 함께 변경되는 부분을 하나의 구현 단위로 묶고 외부에서는 제공되는 함수만 이용해 데이터에 접근한다

  ⇢ 잘 정의된 ***퍼블릭 인터페이스*** 를 통해 데이터에 대한 접근을 통제해야 한다 ⇢ ***의존성 관리의 핵심***

  > 초기 소프트웨어 개발 분야의 선구자 중 한명인 데이비드 파나스는 기능 분해가 가진 본질적인 문제를 해결하기 위해 이 같은 개념을 기반으로 한 **정보 은닉** 과 **모듈** 이라는 개념을 제시하기에 이르렀다.

2-5. 언제 하향식 분해가 유용한가?

- 설계의 다양한 측면을 논리적으로 설명하고 문서화하기에 용이하다

  그러나 설계를 문서화하는데 적절한 방법이 좋은 구조를 설계할 수 있는 방법과 동일한 것은 아니다
- 하향식 분해는 새로운 것을 개발하고, 설계하고 발견하는데는 적합한 방법이 아니다

  하향식은 이미 완전히 이해된 사실을 서술하기에 적합한 방법이다 - 마이클 잭슨

하향식 설계가 가지는 문제점

- 하나의 함수에 제어가 집중되기 때문에 확장이 어렵다
- 프로젝트 초기에 설계의 본질적인 측면을 무시하고 사용자 인터페이스 같은 비본질적인 측면에 집중하게 만든다
- 과도하게 함수에 집중하게 함으로써 데이터에 대한 영향도를 파악하기 어렵게 만든다
- 하향식 분해를 적용한 설계는 재사용하기 어렵다

3. 모듈

3-1. 정보 은닉과 모듈

시스템의 변경을 관리하는 기본적인 전략

- 함께 변경되는 부분을 하나의 구현 단위로 묶고 퍼블릭 인터페이스를 통해서만 접근하도록 만드는 것
- 기능을 기반으로 시스템을 분해하는 것이 아니라 **변경의 방향** 에 맞춰 시스템을 분해하는 것

정보 은닉(information hiding)

- 시스템을 ***모듈 단위로 분해*** 하기 위한 기본원리
- 시스템에서 자주 변경되는 부분을 상대적으로 덜 변경되는 안정적인 인터페이스 뒤로 감추는 것이 핵심

모듈

- 변경될 가능성이 있는 비밀을 내부로 감추고, 잘 정의되고 쉽게 변경되지 않을 퍼블릭 인터페이스를 외부에 제공해서 내부의 비밀에 함부로 접근하지 못하게 한다
- 다음 두가지 비밀을 감춰야 한다
    - **복잡성** : 모듈이 너무 복잡한 경우 이해하고 사용하기 어렵다. 외부에 모듈을 추상화할 수 있는 간단한 인터페이스를 제공해서 모듈의 복잡도를 낮춘다.
    - **변경 가능성** : 변경 가능한 설계 결정이 외부에 노출될 경우 실제로 변경이 발생했을 때 파급효과가 커진다. 변경 발생 시 하나의 모듈만 수정하면 되도록 변경 가능한 설계 결정을 모듈 내부로 감추고 외부에는 쉽게 변경되지 않을 인터페이스를 제공한다.

모듈과 기능분해

상호 배타적인 관계가 아니다. 시스템을 모듈로 분해한 후에는 각 모듈 내부를 구현하기 위해 기능 분해를 적용할 수 있다.

- 기능 분해 : 하나의 기능을 구현하기 위해 필요한 기능들을 순차적으로 찾아가는 **탐색의 과정**
- 모듈 분해 : 감춰야 하는 비밀을 선택하고 비밀 주변에 안정적인 보호막을 설치하는 **보존의 과정**

비밀을 결정하고 모듈을 분해한 후에는 기능 분해를 이용해 모듈에 필요한 퍼블릭 인터페이스를 구현할 수 있다.

급여 관리 시스템 개선

- 급여 관리 시스템에서 외부로 감춰야 하는 비밀은 직원 정보(데이터)와 관련된 것이다
    - 보통 시스템의 가장 일반적인 비밀은 데이터이다

      하지만, **데이터 캡슐화 ≠ 정보 은닉** 임을 인지하자, 복잡한 로직이나 변경 가능성이 큰 자료 구조일 수도 있다

> 데이터와 메서드를 하나의 단위로 통합하고 퍼블릭 메서드를 통해서만 접근하도록 허용하는 방법을 **데이터 캡슐화(data encapsulation)** 라고 한다. 정보 은닉과 데이터 캡슐화는 동일한 개념이 아니다.
> 변경과 관련된 비밀을 감춘다는 측면에서 정보 은닉과 캡슐화는 동일 개념을 가리키는 두 가지 다른 용어지만 데이터 캡슐화는 비밀의 한 종류인 데이터를 감추는 캡슐화의 한 종류일 뿐이다

- 모듈을 이용해 직원 정보라는 비밀을 내부로 감추고 외부에 대해서는 퍼블릭 인터페이스만 노출한다
    - ruby 언어에서 제공하는 module 이라는 키워드를 이용해 구현
    - 전역 변수였던 $employees, $basePays .. 등을 Employees 라는 모듈 내부로 숨긴다
        - 모듈 외부에서는 직원 정보를 관리하는 데이터에 직접 접근 할 수 없게 만듦
    - 외부에서는 Employyes 모듈이 제공하는 퍼블릭 인터페이스에 포함된 함수들을 통해서만 내부 변수를 조작할 수 있다
        - 모듈 외부에서는 모듈 내부에 어떤 데이터가 존재하는지조차 알지 못한다

3-2. 모듈의 장점과 한계

개선된 예제를 통해 알 수 있는 모듈의 장점

- 모듈 내부의 변수가 변경되더라도 모듈 내부에만 영향을 미친다 ⇢ 높은 응집도/낮은 결합도, SRP 만족
- 비즈니스 로직과 사용자 인터페이스에 대한 **관심사를 분리** 한다 ⇢ 관심사의 분리(separation of concerns, SoC)로 복잡성 극복
- 전역 변수와 전역 함수를 제거함으로써 네임스페이스 오염(namespace pollution)을 방지한다 ⇢ 이름 충돌 위험 완화

모듈

- 정보 은닉이라는 개념을 통해 데이터를 설계의 중심요소로 부각시켰다 ⇢ 모듈의 핵심은 데이터
- 메인 함수를 정의하고 필요에 따라 더 세부적인 함수로 분해하는 하향식 기능 분해와 달리 모듈은 감춰야 할 데이터를 결정하고 이 데이터를 조작하는데 필요한 함수를 결정한다 ⇢ 데이터를 중심으로 시스템을 분해
- 데이터와 함수가 통합된 한차원 높은 추상화를 제공하는 설계 단위

모듈의 한계

- 프로시저 추상화 보다는 높은 추상화 개념을 제공하지만 태생적으로 변경을 관리하기 위한 구현기법이기 때문에 추상화 관점에서의 한계점이 명확하다

  ⇢ 모듈의 가장 큰 단점은 인스턴스의 개념을 제공하지 않는다는 점
- 개선된 예제의 Employees 모듈의 좀 더 높은 수준의 추상화를 위해서는 직원 전체가 아니라 개별 직원을 독립적인 단위로 다룰 수 있어야 한다

  ⇢ 다수의 직원 인스턴스가 존재하는 추상화 매커니즘이 필요한 것 ⇢ 이를 만족시키기 위해 등장한 개념이 ***추상 데이터 타입***

4. 데이터 추상화와 추상 데이터 타입

4-1. 추상 데이터 타입

타입(type)

- 변수에 저장할 수 있는 내용물의 종류와 변수에 적용될 수 있는 연산의 가짓수를 의미
- 저장된 값에 대해 수행될 수 있는 연산의 집합을 결정하기 때문에 변수의 값이 어떻게 행동할 것이라는 것을 예측할 수 있게 한다
- 예를 들어, 정수타입의 변수는 덧셈 연산을 이용해 값을 더할 수 있고 문자열 타입 변수는 연결 연산을 이용해 두 문자열을 하나로 합칠 수 있다
- 프로그래밍 언어는 다양한 형태의 내장 타입(built-in type)을 제공한다
    - 기능 분해의 시대에 사용되던 절차형 언어들은 적은 수의 내장 타입만을 제공했으며 새로운 타입을 추가하는 것이 불가능하거나 제한적이었다
      이 시대의 주된 추상화는 프로시저 추상화였다
      시간이 흐르면서 사람들은 프로시저 추상화로는 프로그램의 표현력을 향상시키는데 한계가 있다는 사실을 발견했다

프로시저 추상화의 한계 보완

- 바바라 리스코프(Barbara Liskov)는 프로시저 추상화를 보완하기 위해 **데이터 추상화(data abstraction)**의 개념을 제안했다
    - SOLID 원칙 중 LSP 의 그 리스코프 맞다 😲
    - 리스코프는 소프트웨어를 이용해 표현할 수 있는 추상화의 수준을 한 단계 높였다

> 안타깝게도 프로시저만으로는 충분히 풍부한 추상화의 어휘집을 제공할 수 없다. ... 이것은 언어 설계에서 가장 중요한 추상 데이터 타입(Abstract Data Type)의 개념으로 우리를 인도했다. 추상 데이터 타입은 추상 객체의 클래스를 정의한 것으로 추상 객체에 사용할 수 있는 오퍼레이션을 이용해 규정된다. 이것은 오퍼레이션을 이용해 추상 데이터 타입을 정의할 수 있음을 의미한다. ... 추상 데이터 객체를 사용할 때 프로그래머는 오직 객체가 외부에 제공하는 행위에만 관심을 가지며 행위가 구현되는 세부적인 사항에 대해서는 무시한다. 객체가 저장소 내에서 어떻게 표현되는지와 같은 구현 정보는 오직 오퍼레이션을 어떻게 구현할 것인지에 집중할때만 필요하다. 객체의 사용자는 이 정보를 알거나 제공받을 필요가 없다.
>
> - 리스코프

- 추상 데이터 타입은 프로시저 추상화 대신 데이터 추상화를 기반으로 소프트웨어를 개발하게 한 최초의 발걸음이다

추상 데이터 타입을 구현하기 위해 프로그래밍 언어가 지원해야 하는 특성

- 타입 정의를 선언할 수 있어야 한다
- 타입의 인스턴스를 다루기 위해 사용할 수 있는 오퍼레이션의 집합을 정의할 수 있어야 한다
- 제공된 오퍼레이션을 통해서만 조작할 수 있도록 데이터를 외부로부터 보호할 수 있어야 한다
- 타입에 대해 여러 개의 인스턴스를 생성할 수 있어야 한다

추상 데이터 타입 구현

- 예제에서 사용된 ruby 는 추상 데이터 타입을 흉내 낼 수 있는 Struct 라는 구성 요소를 제공한다

  추상 데이터 타입을 구현하는 방법은 언어마다 다르기 때문에 책에서는 개념적인 수준에서 설명한다는 점을 유의하자

- 급여 관리 시스템 개선

  1. 직원이라는 데이터 추상화가 필요한지 질문한다 ⇢ 개별 직원을 위한 추상 데이터 타입을 구현한다 **= *내부에 캡슐화할 데이터를 결정***

     이름, 기본급, 아르바이트 직원 여부, 작업시간을 비밀로 가지는 추상 데이터 타입인 Employee 를 선언한다

  2. 추상 데이터 타입에 적용할 수 있는 ***오퍼레이션을 결정***

     Employee 타입의 주된 행동은 직원의 유형에 따라 급여를 계산하는 것이므로 calculatePay 오퍼레이션을 추가한다

     외부에서 인자로 전달받던 직원의 이름은 이제 Employee 타입의 내부에 포함돼 있으므로 calculatePay 오퍼레이션의 인자로 받을 필요가 없다

     ⇢ 직원을 지정해야 했던 모듈 방식보다 추상 데이터 타입에 정의된 오퍼레이션의 시그니처가 더 간단하다는 것을 알 수 있다

  3. 두 번째 오퍼레이션으로 개별 직원의 기본급을 계산, 아르바이트 직원의 경우 기본급 개념이 없기때문에 0을 반환

     ⇢ Employee 추상 데이터 타입에 대한 설계 완료

  4. 추상 데이터 타입을 사용하는 클라이언트 코드 작성

     직원들의 인스턴스 생성

  5. 기존 코드에서 개선한 오퍼레이션을 호출하도록 수정

추상 데이터 타입이 가능하게 한 것

- 사람들이 세상을 바라보는 방식에 좀 더 근접해지도록 추상화 수준을 향상시켰다
- 예제의 Employees 모듈보다 Employee 추상 데이터 타입이 좀 더 개념적으로 사람들의 사고방식에 가깝다

추상 데이터 타입의 한계

- 시스템의 상태를 저장할 데이터를 표현하지만 데이터를 이용해서 기능을 구현하는 핵심 로직은 추상 데이터 타입 외부에 존재한다
- 데이터에 대한 관점을 설계의 표면으로 끌어올리기는 하지만 여전히 데이터와 기능을 분리하는 절차적인 설계의 틀에 갇혀 있는 것이다
- 프로그래밍 언어의 관점에서 추상 데이터 타입은 언어의 내장 데이터 타입과 동일하다

  단지 타입을 개발자가 정의할 수 있다는 점이 다를 뿐이다 ⇢ 혼란을 유발하는 관점 그리고 클래스와 관련된 의문 (클래스는 추상 데이터 타입인가?)

5. 클래스

5-1. 클래스는 추상 데이터 타입인가?

클래스와 추상 데이터 타입의 비슷한 부분

- 데이터 추상화를 기반으로 시스템을 분해한다
- 외부에서는 객체의 내부 속성에 직접 접근할 수 없으며 오직 퍼블릭 인터페이스를 통해서만 외부와 의사소통 할 수 있다

명확한 의미에서는 동일하지 않다

가장 핵심적인 차이는 **상속** 과 **다형성** 의 지원 여부이다.

- 상속과 다형성을 지원한다 ⇢ 클래스, 객체지향 프로그래밍(Object-Oriendted Programming)
- 상속과 다형성을 지원하지 못한다 ⇢ 추상 데이터 타입, 객체기반 프로그래밍(Object-Based Programming)

프로그래밍 언어 관점에서의 차이

- 추상 데이터 타입 : 타입을 추상화한 것(type abstraction)
  - 오퍼레이션을 기준으로 타입을 묶는다
  - 예제의 Employee 내부에 정규 직원과 아르바이트 직원이라는 두개의 타입이 공존한다
- 클래스 : 절차를 추상화한 것(procedural abstraction)
  - 타입을 기준으로 오퍼레이션을 묶는다
  - 객체지향은 정규 직원과 아르바이트 직원이라는 두개의 타입을 명시적으로 정의하고 두 직원 유형과 관련된 오퍼레이션의 실행 절차를 두 타입에 분배한다

> p.246 그림 7.4 와 그림 7.5 를 참고하자

추상화와 분해의 관점에서의 차이

- 추상 데이터 타입 ⇢ 오퍼레이션을 기준으로 타입들을 추상화 (type abstraction)
- 클래스 ⇢ 타입을 기준으로 절차들을 추상화 (procedural abstraction)

5-2. 추상 데이터 타입에서 클래스로 변경하기

클래스를 이용해 급여 관리 시스템을 구현

- 각 직원 타입을 독립적인 클래스로 구현함으로써 두 개의 타입이 존재한다는 사실을 명시적으로 표현한다
- Employee 클래스를 추상클래스로 정의하고, calculatePay 와 monthlyBasePay 메서드를 추상 메서드로 정의한다

- 정규 직원 타입을 SalariedEmployee 클래스로 아르바이트 직원 타입을 HourlyEmployee 클래스로 구현한다

  각 클래스의 메서드는 오직 각자의 타입과 관련된 로직만을 구현한다

- 모든 직원 타입에 대해 Employee 의 인스턴스를 생성해야 했던 추상 데이터 타입의 경우와 달리 클래스를 이용해서 구현한 코드의 경우에는 클라이언트가 원하는 직원 타입에 해당하는 클래스의 인스턴스를 명시적으로 지정하여 생성한다

  ⇢ 객체를 생성하고 나면 객체의 클래스가 무엇인지는 중요하지 않다

  클라이언트의 입장에서는 SalariedEmployee 와 HourlyEmployee 의 인스턴스를 모두 부모 클래스인 Employee 의 인스턴스인 것처럼 다룰 수 있다

5-3. 변경을 기준으로 선택하라

- 타입을 기준으로 절차를 추상화하지 않았다면 클래스를 구현 단위로 사용하더라도 객체지향 분해가 아니다
- 객체지향에서는 타입 변수를 이용한 조건문을 **다형성으로 대체** 한다

  클라이언트가 객체의 타입을 확인한 후 적절한 메서드를 호출하는 것이 아니라 객체가 메시지를 처리할 적절한 메서드를 선택한다

  - 흔히 '객체지향이란 조건문을 제거하는 것'이라는 다소 오해(편협한 견해)가 널리 퍼진 이유
- 조건문을 사용하는 방식을 기피하는 이유는 **변경** 때문이다

  추상 데이터 타입을 기반으로 한 Employee 에 새로운 직원 타입을 추가하기 위해서는 hourly 의 값을 체크하는 클라이언트의 조건문을 하나씩 다 찾아 수정해야 한다

  이에 반해 객체지향은 새로운 직원 유형을 구현하는 클래스를 Employee 상속 계층에 추가하고 필요한 메서드를 오버라이딩하면 된다

  새로 추가된 클래스의 메서드를 실행하기 위한 어떤 코드도 추가할 필요가 없다

  ⇢ 새로운 로직을 추가하기 위해 클라이언트 코드를 수정할 필요가 없다는 것을 의미 👍

  - 기존 코드에 아무런 영향도 미치지않고 새로운 객체 유형과 행위를 추가할 수 있는 객체지향의 특성을 **개방-폐쇄 원칙(Open-Closed Principle, OCP)** 라고 부른다

    이것이 객체지향 설계가 전통적인 방식에 비해 변경하고 확장하기 쉬운 구조를 설계할 수 있는 이유다
- 설계는 **변경** 과 관련된 것이다

  설계의 유용성은 변경의 방향성과 발생 빈도에 따라 결정된다

  추상 데이터 타입과 객체지향 설계의 유용성은 설계에 요구되는 변경의 압력이 '타입 추가'에 관한 것인지, '오퍼레이션 추가'에 관한 것인지에 따라 달라진다

  - 타입 추가 > 오퍼레이션 추가 ⇢ 객체지향 선택
  - 오퍼레이션 추가 > 타입 추가 ⇢ 추상 데이터 타입 선택

> 변경의 축을 찾아라.
>
> 객체지향적인 접근법이 모든 경우에 올바른 해결 방법인 것은 아니다.

> **데이터-주도 설계**

> 레베카 워프스브록은 추상 데이터 타입의 접근법을 객체지향 설계에 구현한 것을 데이터 주도 설계라고 부른다.
> 워프스브록이 제안한 책임 주도 설계는 데이터 주도 설계 방법을 개선하고자 하는 노력의 산물이었다.
> 티모시 버드는 모듈과 추상 데이터 타입이 데이터 중심적인 관점(data centered view)을 취하는데 비해 객체지향은 서비스 중심적인 관점(service centered view)을 취한다는 말로 둘 사이의 차이점을 깔끔하게 설명했다.

5-4. 협력이 중요하다

- 객체지향에서 중요한 것은 ***역할, 책임, 협력*** 이다

  객체지향은 기능을 수행하기 위해 객체들이 협력하는 방식에 집중한다

  협력이라는 문맥을 고려하지 않고 객체를 고립시킨 채 오퍼레이션의 구현 방식을 타입별로 분배하는 것은 올바른 접근법이 아니다
- 객체를 설계하는 방법은 3장에서 설명했던 **책임 주도 설계의 흐름을 따른다** 는 점을 기억하자
- 객체가 참여할 **협력을 결정** 하고 협력에 필요한 **책임** 을 수행하기 위해 어떤 객체가 필요한지에 관해 고민하라

  그 책임을 다양한 방식으로 수행해야 할 때만 **타입 계층 안에 각 절차를 추상화** 하라

  타입 계층과 다형성은 협력이라는 문맥 안에서 책임을 수행하는 방법에 관해 고민한 결과물이어야 하며 그 자체가 목적이 되어서는 안 된다

참고

관심사 분리(separation of concerns) - 위키

리스코프 치환 원칙(LSP) - 위키