Chapter 07 객체 분해

객체 분해

사람의 기억은 단기/ 장기로 구분이 된다.

장기 기억은 경험한 내용을 수개월에서 길게는 평생에 걸쳐 보관하는 장소를 의미.

  큰 용량이 있지만 직접 접근할 수 없음.

  단기 기억으로 이동 시켜야함.

단기 기억은 보관되어 있는 지식에 직접 접근할 수 있다.

하지만 속도와 공간적인 측면 모두에서 제약을 받는다.

컴퓨터라면야 프로그램을 작성할 때, 시간과 공간의 트레이드 오프를 통해 효율을 향상시킬 수 있지만, 사람은 그럴 수 없어!

사람의 단기 기억에 있어서 시간과 공간의 두 측면 모두가 병목지점으로 작용한다!

핵심은?

실제로 문제를 해결 하기 위해 사용하는 저장소는 장기 기억이 아니라 단기 기억이라는 점!

문제해결을 위해선 필요한 정보들을 먼저 단기 기억 안으로 불러들여야 한다!

하지만 문제 해결에 필요한 요소의 수가 기억의 용량을 초과하는 순간 ! 문제 해결능력은 급격하게 떨어진다!

이것이 인지 과부하(Cognitive overload) 이다

그럼 인지과부하를 막으려면 어떻게 해?

가장 좋은 방법?

단 기억 안에 보관할 정보 양을 조절 하면 된다!

정말 필요한 정보만 남겨두고 불필요한 세부 사항을 걸러내면 무네를 단순화할 수 있을것이다 .

이것처럼 불필요한 정보를 제거하고 현재의 문제 해결에 필요한 핵심만 남기는 작업을 추상화라고 한다.

추상화가 뭔데

불필요한 정보를 제거하고 현재의 문제를 해결에 필요한 핵심만 남기는 것!

보통 추상화는 한번에 다뤄야 하는 문제의 크기를 줄이는 것이다.

나뉜 문제들 중에서도 한 번에 해결하기 어려운 문제라면 더 작은 문제로 나누면 된다!

이것을 분해 ! 라고 한다.

분해?...

분해는 문제를 해결 가능한 작은 문제로 나누는 작업을 분해 라고 한다.

분해의 목적 큰 문제를 인지 과부하의 부담 없이 단기 기억 안에서 한번에 처리할 수 있는 규모의 문제로 나누는 것이다

뭘로 나누는데?

청크 (하나의 단위로 취급 될 수 있는 논리적인 의미)

연속적으로 분해 가능하다.

한번에 담기 힘든 추상화의 수에는 한계가 있지만 추상화를 더 큰 규모의 추상화로 압축시킴으로서 단기 기억의 한계를 초월할 수 있다.

프로시저 추상화와 데이터 추상화

프로그래밍 패러다임은 프로그래밍을 구성하기 위해 사용하는 추상화의 종류와 이 추상화를 이용해 소프트웨어를 분해하는 방법의 두 가지 요소로 결정.

바로 추상화와 분해의 관점

현대적인 프로그래밍 언어를 특징 짓는 중요한 두 가지 추상화 메커니즘

1. 프로시저 추상화 2. 데이터 추상화

프로시저 추상화 : 소프트 웨어가 무엇을 해야 하는지는 추상화 한다.

-> 기능분해 = 알고리즘 분해

데이터 추상화 : 소프트 웨어가 무엇을 알아야하는지를 추상화한다.

-> 선택지 두가지

1. 타입을 추상화 = > 추상 데이터 타입
2. 프로시저를 추상화 (데이터를 중심으로) =>객체지향
   - 소프트 웨어는 데이터를 이용해 정보를 표현하고 프로시저를 이용해 데이터를 조작한다.

프로시저 추상화와 기능분해

기능과 데이터의 첫 전쟁에서 신은 기능의 손을 들어주었다?

- 기능은 오랜 시간 동안 시스템을 분해하기 위한 기준으로 사용되었다.
- 기능으로 분해하는 시스템을 분해하는 방식을 알고리즘 분해, 기능 분해 라고 부른다.
- 기능 분해의 관점에서 추상화의 단위는 프로시저 이다.
- 시스템은 프로시저를 단위로 분해된다.

프로시저는 반복적으로 실행되거나 유사하게 실행되는 작업들을 하나의 장소에 모아놓음으로써 로직을 재사용하고 중복을 방지할 수 있는 추상화 방법이다.

프로시저를 추상화라고 부르는 이유 ?

- 내부 구현 내용을 상세하게 몰라도 인터페이스만 알면 프로시저를 사용 가능!

-> 프로시저는 잠재적으로 정보은닉(information hiding)의 가능성을 제시는 하지만!

효과적인 정보은닉 체계를 구축하기는 힘들다.

프로시저 중심의 기능분해의 관점에서 시스템은 입력 값을 계산해서 출력 값을 반환하는 수학의 함수와 동일 => 시스템은 더 작은 작업으로 분해될 수 있는 하나의 큰 메인함수와 같다.

전통적인 기능 분해 방법은?

하향식 접근법 (Top-Down Approach)이다.

- 시스템을 구성하는 가장 최상위 기능을 정의하고,

  이 최상위 기능을 좀 더 작은 단계의 하위 기능으로 분해해 나가는 방법을 말한다.

  분해는 마지막 하위 기능이 프로그래밍 언어로 구현 가능한 수준이 될 때까지 계속된다.

  각 세분화 단계는 바로 위 단계보다 더 구체적이어야 한다.

  다시 말해 정제된 기능은 자신의 바로 상위 기능보다 덜 추상적이어야 한다.

  즉 상위 기능은 하나 이상의 더 간단하고 구체적이며 덜 추상적인 하위 기능의 집합으로 분해된다.

급여 관리 시스템

- 연초 회사는 매달 지급해야하는 기본급에 대해 직원과 협의해서 12개월동안 동이랗게 지급한다.
- 회사는 급여 지급 시 소득세율에 따라 일정 금액의 세금을 공제한다.

  급여 = 기본급 - (기본급 * 소득세율)

- 기능 분해 방법을 이용한다.
- 하향식 접근법을 따른다. (최상위의 추상적인 함수 정의에서 출발해서 단계적인 정제 절차를 따라 시스템 구축)
- 최상위 추상 함수 정의는 시스템의 기능을 표현하는 하나의 문장으로 표현.
- 좀 더 세부적인 단계의 문장으로 분해해 나가는 방식을 따른다.
- 기능 분해의 초점은 하나의 문장으로 표현된 기능을 여러 개의 더 작은 기능으로 분해하는 것.

추상적인 최상위 문장을 기술한다. (급여관리 시스템을 시작하는 메인 프로시저로 구현될 것이다.)

- 직원의 급여를 계산한다.

기능 분해 방법에 따라 이 프로시저를 실제로 급여를 계산하는데 필요한 좀 더 세부적인 절차로 구체화 될 수 있다.

- 직원의 급여를 계산한다

  사용자로부터 소득 세율을 입력받는다.

  직원의 급여를 계산한다.

  양식에 맞게 결과를 출력한다.

각 정제 단계는 이전 문장의 추상화 수준을 감소시켜야 한다.

(모든 문장이 정제 과정을 거치면서 하나 이상의 좀 더 단순하고 구체적인 문장들의 조합으로 분해되어야 한다.)

개발자는 각 단계에서 불완전하고 좀 더 구체화될 수 있는 문장들이 있나 검토.

저수준의 문장이 될 때 까지 기능을 분해해야 한다.

직원의 급여를 계산하기 위해서는 소득세율뿐 아니라 직원 기본급 정보 역시 필요하다.

직원의 목록과 개별 직원에 대한 기본급 데이터를 시스템 내부에 보관하기로 했다고 한다.

급여 계산 결과는 “이름:{직원명}, 급여 :{계산된 금액} “ 형식에 따라 출력 문자열을 생성.

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

기본적으로 기능 분해는 책의 목차를 정리하고 그 안에 내용을 채워넣는것과 유사.

급여 관리 시스템을 입력 받아 출력을 생성하는 커다란 하나의 메인함수로 간주하고 기능 분해를 시작했다는 점에 주목하자. 입력정보는 직원정보와 소득세율이고 출력은 계산된 급여 정보이다.

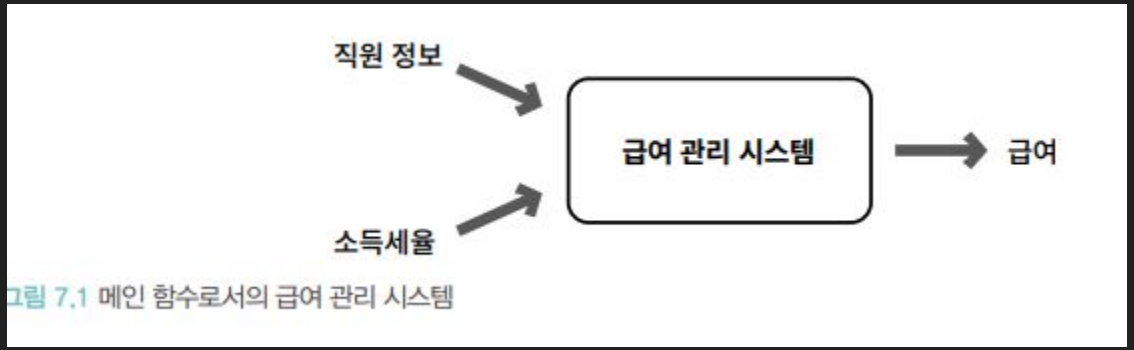

그림 7.1 메인 함수로서의 급여 관리 시스템

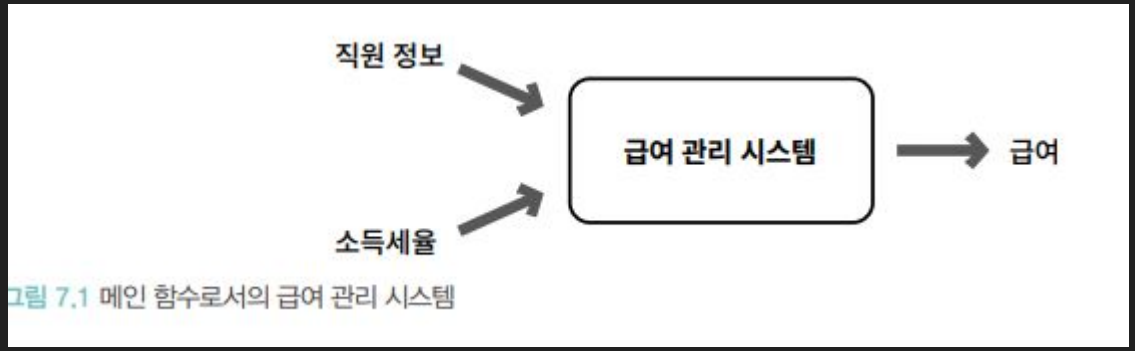

그림 7.1 메인 함수로서의 급여 관리 시스템

기능 분해 방법에서는 기능을 중심으로 필요한 데이터를 결정한다.

기능 분해라는 무대!

주연은 기능!

데이터는 조연역할이다.

기능 분해를 위한 하향식 접근법은 먼저 필요한 기능을 생각하고 이 기능을 분해하고 정제하는 과정에서 필요한 데이터의 종류와 저장 방식을 식별한다.

급여 관리 시스템 구현

최상위 문장은 다음과 같다.

[직원의 급여를 계산한다 .]

이 문장이 메인함수로 매핑되고 앞에서 급여를 계산하는 데 필요한 소득세율은 사용자로부터 입력받고 직원의 기본급 정보는 시스템에 저장된 값을 참조.

직원에 대한 정보를 찾기 위해 필요한 직원의 이름은 함수의 인자로 받는다.

결과

```
def main(name)
end
```

구현을 위해 세분화 해보자

직원의 급여를 계산한다

사용자로부터 소득세율을 입력 받는다.

직원의 급여를 계산한다.

양식에 맞게 결과를 출력한다.

- 위는 더 작은 세부적인 단계로 분해가능하니 단계를 프로시저를 호출하는 명령문으로 변환할 수 있다.

```
def main(name)
    taxRate = getTaxRate()
    pay = calculatePayFor(name,taxRate)
puts(describeResult(name,pay))
end
```

사용자로부터 소득세율을 입력받는 getTaxRate 함수는 다음과 같은 두 개의 절차로 분해할 수 있다.

직원의 급여를 계산한다.

사용자로부터 소득세율을 입력 받는다.

“세율을 입력하세요:” 라는 문장을 화면에 출력한다.

키보드를 통해 세율을 입력 받는다.

직원의 급여를 계산한다.

양식에 맞게 결과를 출력한다.

```
def getTaxRate( )
  print (“세율을 입력하세요:”)
  return gets( ).chomp( ).to_f( )
end
```

급여를 계산하는 코드는 기본급 정보를 이용해 급여를 계산하는 두 개의 단계로 구현가능하다.

직원의 급여를 계산한다.

사용자로부터 소득세율을 입력받는다.

직원의 급여를 계산한다.

전역 변수에 저장된 직원의 기본급 정보를 얻는다.

급여를 계산한다.

양식에 맞게 결과를 출력한다.

1.

직원 목록과 기본급에 대한 정보를 유지하고 있어야 한다.

- 직원 목록은 $employees라는 전역 변수
- 직원별 기본급은 $basePays 라는 전역 변수

$employees = [“직원 A” , “직원B”, “직원C”]

$basePays = [400, 300, 250]

급여를 계산하는 calculatePayFor 함수는 파라미터로 전달된 직원의 이름을 이용해 $employees 배열 안에서의 인덱스를 알아낸 후 $basePays 의 해당 인덱스에 위치한 기본급 정보를 얻는다. 지급될 급여는 '기본급 - (기본급 * 소득세율)' 의 공식에 따라 계산된 후 반환한다.

```
def calculatePayFor(name, taxRate)
    index = $employees.index(name)
    basePay = $basePays[index]
    return basePay - (basePay * taxRate)
end
```

급여를 계산했으므로 마지막으로 급여 내역을 출력양식을 맞게 포매팅한 후 반환하면 모든 작업이 완료된다.

직원의 급여를 계산한다.

사용자로부터 소득세율을 입력 받는다.

직원의 급여를 계산한다.

양식에 맞게 결과를 출력한다.

“이름 : {직원명}, 급여 : {계산된 금액}” 형식에 따라 출력 문자열을 생성한다.

describeResult 함수는 이름과 급여 정보를 이용해 출력 포맷에 따라 문자열을 조합해 후 반환한다.

```
def describeResult(name, pay)
  return “이름 :#{name}, 급여 :#{pay}”
 end
```

```
def describeResult (name, pay)
    return “이름 : #{name} , 급여 : #{pay}”
end
```

이름이 “직원C”인 직원의 급여를 계산하려면 다음과 같이 프로시저를 호출하면 된다.

```
main (“직원C”)
```

예제에서 알 수 있는 것처럼 하향식 기능 분해는 시스템을 최상위의 가장 추상적인 메인 함수로 정의하고, 메인 함수를 구현 가능한 수준까지 세부적인 단계로 분해하는 방법이다.

하향식 기능 분해 방식으로 설계한 시스템은 메인 함수를 루트로 “트리(tree)”로 표현할 수 있다.

트리에서 각 노드는 시스템 구성하는 하나의 프로시저의 의미하고 하고 한 노드의 자식 노드는 부모 노드를 구현하는 절차 중의 한단계를 의미한다.

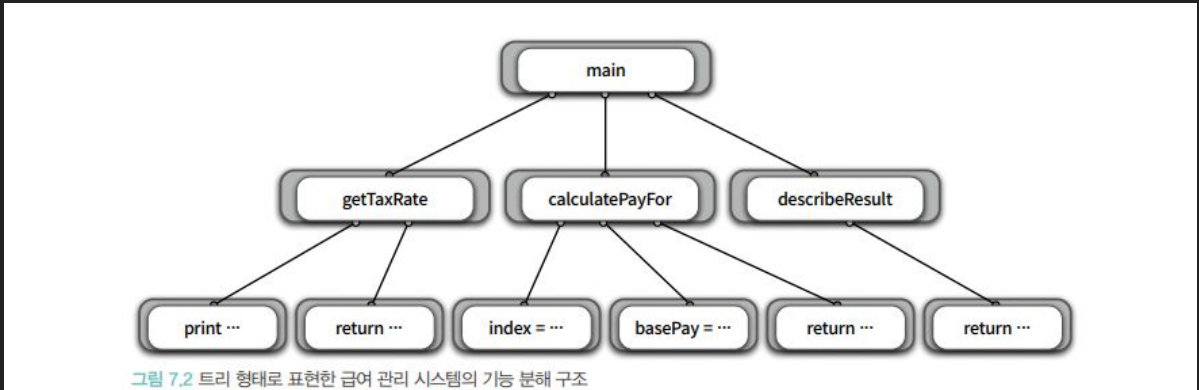

그림 7.2 트리 형태로 표현한 급여 관리 시스템의 기능 분해 구조

이처럼 하향식 기능 분해는 논리적이고 체계적인 시스템 개발 절차를 제시한다.

커다란 기능을 좀 더 작은 기능으로 단계적으로 정제해 가는 과정은 구조적이며 체계적인 동시에 이상적인 방법으로까지 보일것이다.

하향식 기능 분해의 문제점

- 시스템의 하나의 메인 함수로 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 고정 시키기 때문에 유연성을 재사용성이 저하된다.
- 데이터 형식이 변경될 경우 파급효과를 예측할 수 없다.

설계는 코드 배치 방법이며 설계가 필요한 이유는 변경에 대비하기 위한 것이라는 점을 기억하라.

하나의 메인 함수라는 비현실적인 아이디어

어떤 시스템도 최초에 릴리스됐던 당시 모습을 그대로 유지하지는 않는다.

대부분의 경우 추가되는 기능은 최초에 배포된 메인 함수의 일부가 아닐 것이다.

결국 처음에는 중요하게 생각됐던 메인 함수는 동등하게 중요한 여러 함수들 중 하나로 전락하고 만다.

어느 시점에 이르면 유일한 메인 함수라는 개념은 의미가 없어지고 시스템은 여러 개의 동등한 수준의 함수 집합으로 성장하게 될 것이다.

대부분의 시스템에서 하나의 메인 기능이란 개념은 존재하지 않는다.

모든 기능들은 규모라는 측면에서 차이가 이을수 있겠지만 기능성의 측면에서는 동등하게 독립적이고 완결된 하나의 기능을 표현한다.

메인 함수의 빈번한 재설계

기존 코드를 수정하는 것은 항상 새로운 버그를 만들어낼 확률을 높인다는 점에 주의하라.

추가된 코드와는 아무런 상관도 없는 위치에서 빈번하게 발생하는 버그는 새로운 기능을 추가하거나 기존 코드를 수정하는데 필요한 용기를 꺾는다.

결과적으로 기존 코드의 빈번한 수정으로 인한 버그 발생 확률이 높아지기 때문에 시스템은 변경에 취약해질 수 밖에 없다.

비즈니스로직과 사용자 인터페이스의 결합

하향식 접근법은 비즈니스 로직을 설계하는 초기 단계부터 입력 방법과 출력 양식을 함께 고민하도록 강요한다.

문제는 비즈니스 로직과 사용자가 인터페이스가 변경되는 빈도가 다르다는 것이다. 사용 인터페이스는 시스템 내에서는 가장 자주 변경되는 부분이다.

반면 비즈니스 로직은 사용자 인터페이스가 변경되는 빈도가 다르다는 것이다.

사용자 인터페이스는 시스템 내에서 가장 자주 변경 되는 부분이다. 반면에 비즈니스 로직은 사용자가 인터페이스에 비해 변경이 적게 발생한다.

따라서 하향식 접근법은 근본적으로 변경에 불안정한 아키텍처를 낳는다.

하향식 접근법은 기능을 분해하는 과정에서 사용자 인터페이스의 관심사와 비즈니스 로직의 관심사와 비즈니스 로직의 관심사를 동시에 고려하도록 사용하기 때문에 “관심사의 분리” 라는 아키텍쳐 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 있다.

  이것은 설계를 시작하는 시점부터 시스템이 무엇을 해야 하는지가 아니라 어떻게 동작해야 하는지에 집중하도록 만든다.

하향식 접근법의 설계는 처음부터 구현을 염두에 두기 때문에 자연스럽게 함수들의 실행순서를 정의하는 시간 제약을 강조한다.

메인 함수가 작은 함수들로 분해되기 위해서는 우선 함수들의 순서를 결정해야 한다.

실행 순서나 조건, 반복과 같은 제어 구조를 미리 결정하지 않고는 분해 진행할 수 없기 때문에 기능 분해 방식은 중앙집중 제어 스타일의 형태를 띨 수 밖에 없다.

결과적으로 모든 중요한 제어 흐름의 결정이 상위 함수에서 이뤄지고 함위 함수는 상위 함수의 흐름에 따라 적절한 시점에 호출된다.

문제는 중요한 설계 결정사항인 함수의 제어 구조가 빈번한 변경의 대상이라는 점이다. 기능이 추가되거나 변경될 때마다 초기에 결정된 함수들의 제어 구조가 올바르지 않다는 것이 판명 된다.결과 적으로 기능을 추가하거나 변경하는 작업은 매번 기존에 결정된 함수의 제어 구조를 변경하도록 만든다.

이를 해결할 수 있는 한 가지 방법은 자주 변경되는 시간적인 제약에 대한 미련을 버리고 좀 더 안정적인 논리적 제약을 설계의 기준으로 삼는 것이다.

객체지향 함수 간의 호출 순서가 아니라 객체 사이의 논리적인 관계를 중심으로 설계를 이끌어 나간다. 결과적으로 전체적인 시스템은 어떤 한 구성요소로 제어가 집중되지 않고 여러 객체들 사이로 제어 주체가 분산한다.

하향식 접근법을 통해 분해한 함수들은 재사용하기도 어렵다.

모든 함수는 상위 함수를 분해하는 과정에서 필요에 따라 식별되며 그에 따라 상위 함수가 강요하는 문맥 안에서만 의미를 가지기 때문이다.

하향식의 설계과 관려된 모든 문제의 원인은 결합도다.

데이터 변경으로 인한 파급효과

- 하향식 기능 분해의 가장 큰 문제점은 어떤 데이터를 어떤 함수가 사용하고 있는지를 추적하기 어렵다는 것이다.
- 이것은 코드 안에서 텍스트를 검색하는 단순한 문제가 아니다.

  이것은 의존성과 결합도의 문제다. 그리고 테스트의 문제이기도 하다.

  하향식 기능 분해 방법이 데이터 변경에 얼마나 취약한지를 이해하기 위해 급여 관리 시스템에 새로운 기능을 추가해보자.

- 데이터 변경으로 인한 영향을 최소화하려면 데이터와 함께 변경되는 부분과 그렇지 않은 부분을 명확하게 분리한다. 데이터와 함께 변경되는 부분을 하나의 구현 단위로 묶고 외부에서는 대한 접근을 통제해야 하는 것이다.

데이터와 함께 변경되는 부분과 그렇지 않은 부분을 명확하게 분리해야 한다.

이것이 바로 의존성 관리의 핵심이다. 변경에 대한 영향을 최소화하기 위해 영향을 받는 부분과 받지 않는 부분을 명확하게 분리하고 잘 정의된 퍼블릭 인터페이스를 통해 변경되는 부분에 대한 접근을 통제하라.

초기 소프트웨어 개발 분야의 선구자 중에 한명인 데이비드 파나스는 기능 분해가 가진 본질적인 문제를 해결하기 위해 이 같은 개념을 가빈으로 한 정보 은닉과 모듈이라는 개념을 제시하기에 이르렀다.

언제 하향식 분해가 유용하다.

물론 하향식 분해가 유용한 경우도 있다. 하향식 아이디어가 매력적인 이유는 설계가 어느 정도 안정화된 후에는 설계의 다양한 측면을 논리적으로 설명하고 문서화하기에 용이하기 때문이다.

그러나 설계를 문서화하는 데 적절한 방법이 좋은 구조를 설계할 수있는 방법을 동일한 것은 아니다.

마이클 잭슨은 System Development 에서 하향식 방법이 지니고 있는 태생적인 한계와 사람들이 하향식 방법에 관해 오해하는 부분.

하향식 분해는 작은 프로그램과 개별 알고리즘을 위해서는 유용한 패러다임으로 남아 있다. 특히 프로그래밍 과정에서 이미 해결된 알고리즘 문서화하고 서술하는 데는 훌륭한 기법이다.

그러나 실제로 동작하는 커다란 소프트웨어를 설계하는 데 적합한 방법은 아니다.

지금까지 하향식 설계가 가지는 문제점을 살펴봤다.

하향식 분해 방식으로 설계된 소프트웨어는 하나의 함수에 제어가 집중 되기 때문에 확장이 어렵다.

하향식 분해는 프로젝트 초기에 설계의 본질적인 측면을 무시하고 사용자 인터페이스 같은 비본질적인 측면에 집중하게 만든다. 과도하게 함수에 집중하게 함으로써 소프트웨어의 중요한 다른 측면인 데이터에 대한 영향도를 파악하기 어렵다. 또한 하향식 분해를 적용한 설계는 근본적으로 재사용하기 어렵다.

모듈 - 정보은닉과 모듈

- 데이비드 파나스
  소프트웨어 개발의 가장 중요한 원리인 동시에 가장 많은 오해를 받고 있는 정보은닉의 개념을 소개했다.

- 정보 은닉은 시스템을 모듈 단위로 분해하기 위한 기본원리로 시스템에서 자주 변경되는 부분을 상대적으로 덜 변경되는 안정적인 인터페이스 뒤로 감춰야 한다는 것이 핵심이다.

- 데이비드 파나스는 시스템을 모듈로 분할하는 원칙 외부에 유출돼서는 안 되는 비밀의 윤곽을 따라야 한다고 주장한다.

정보 은닉은 외부에 감춰야 하는 비밀에 따라 시스템을 분할하는 모듈 분할 원리다.

모듈은 변경될 가능성이 있는 비밀을 내부로 감추고 , 잘 정의되고 쉽게 변경되지 않을 퍼블릭 인터페이스를 외부에 제공해서 내부의 비밀에 함수로 접근하지 못하게 한다

모듈과 기능 분해는 상호 배타적인 관계가 아니다.

시스템을 모듈로 분해한 후에는 각 모듈 내부를 구현하기 위해 기능 분해를 적용할 수 있다.

기능 분해가 기능을 구현하기 위해 필요한 기능들을 순차적으로 찾아가는 탐색의 과정이라면 모듈 분해는 감춰야 하는 비밀을 선택하고 비밀 주변에 안정적인 보호막을 설치하는 보존의 과정이다. 비밀 결정하고 모듈을 분해한 후에는 기능 분해를 이용해 모듈에 필요한 퍼블릭 인터페이스를 구현할 수 있다.

시스템을 모듈 단위로 어떻게 분해할 것인가?

시스템이 감춰야 하는 비밀을 찾아라, 외부에서 내부의 비밀에 접근하지 못하도록 커다란 방어막을 쳐서 에워싸라. 이 방어막이 바로 퍼블릭 인터페이스가 된다.

모듈은 다음과 같은 두 가지 비밀을 감춰야 한다.

- 복잡성 : 모듈이 너무 복잡한 경우 이해하고 사용하기가 어렵다.
  외부에 모듈을 추상화할 수 있는 간단한 인터페이스를 제공해서 모듈의 복잡도를 낮춘다.

- 변경 기능성 : 변경 가능한 설계 결정이 외부에 노출될 경우 실제로 변경이 발생했을 때 파급효과가 커진다. 변경 발생 시 하나의 모듈만 소중하면 되도록 변경 가능한 설계 결정을 모듈 내부로 감추고 외부에는 쉽게 변경되지 않을 인터페이스를 제공한다.

모듈의 장점과 한계

- 모듈 내부의 변수가 변경되더라도 모듈 내부에만 영향을 미친다.
- 비즈니스 로직과 사용자 인터페이스에 대한 관심사를 분리한다.
- 전역 변수와 전역 함수를 제거함으로써 네임스페이스 오염을 방지한다.

모듈은 기능이 아니라 변경의 정도에 따라 시스템을 부내하게 한다.

각 모듈은 외부에 감춰야 하는 비밀과 관련성 높은 데이터의 함수의 집합이다.

따라서 모듈 내부는 높은 응집도를 유지한다.

모듈과 모듈 사이에는 퍼블릭 인터페이스를 통해서만 통신해야 한다.

따라서 낮은 낮은 결합도를 유지한다.

정부 은닉이라는 개념을 통해 데이터라는 존재를 설계의 중심 요소를 부각 시켰다는 것이다.

모듈에 있어서 핵심은 데이터다.

메임함수를 정의하고 필요에 따라 더 세부적인 함수로 분해하는 하향식 기능 분해와

달리 모듈은 감춰야 할 데이터를 이 데이터를 조작하는 데 필요한 함수를 결정한다.

데이터 추상화와 추상 데이터 타입

- 추상 데이터 타입

프로그래밍 언어에서 타입은 이란 변수에 저장할 수 있는 내용물의 종류와 변수에 적용 될 수 있는 연산의 가짓수를 의미를 한다.

프로그래밍 언어는 다양한 형태의 내장 타입을 제공한다.

기능 분해의 시대에 사용되던 절차형 언어들은 적은 수의 내장 타입만을 제공했으며 설상가상으로 새로운 타입을 추가하는 것이 불가능하거나 제한적이었다.

리스코프는 프로시저 추상화를 보완하기 위해 데이터 추상화의 개념을 제안했다.

인용문에는 지금까지 설명했던 데이터 추상화, 정보 은닉 , 데이터 캡슐화, 인터페이스 - 구현 분리의 개념들이 모두 다 녹아들어 있다.

리스코프의 업적은 소프트웨어를 이용해 표현할 수 있는 추상화의 수준을 한단계 높였다는 점이다. 추상 데이터 타입은 프로시저 추상화 대신 데이터를 추상화를 기반으로 소프트웨어를 개발하게 한 최초의 발걸음인다.

추상 데이터 타입을 구현하려면 다음과 같은 특성을 위한 프로그래밍 언어의 지원이 필요하다.

- 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 타입을 정의하기 위해서 제시한 언어적인 메커니즘을 오퍼레이션 클러스터(operation cluster)라고 불렀다.

추상 데이터 타입을 구현할 수 있는 언어적인 장치를 제공하지 않는 프로그래밍 언어에서도 추상 데이터타입을 구현해 왔다.

실제로 과거의 많은 프로그래머들은 모듈의 개념을 기반으로 추상 데이터 타입을 구현해 왔다.

추상 데이터 타입을 구현하려면 다음과 같은 특성을 위한 프로그래밍 언어의 지원이 필요하다.

- 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 인스턴 생성할 수 있어야 한다.

```
Employee = Struct.new(:name, :basePay, :hourly, :timeCard) do
End
```

```
Employee = Struct.new(:name, :basePay, :hourly, :timeCard) do
  def calculatePay(taxRate)
    if (hourly) then
      return calculateHourlyPay(taxRate)
    end
    return calculateSalariedPay(taxRate)
  end

private
  def calculateHourlyPay(taxRate)
    return (basePay * timeCard) - (basePay * timeCard) * taxRate
  end
```

```
  def calculateSalariedPay(taxRate)
    return basePay - (basePay * taxRate)
  end
end
```

```
  Employee = Struct.new(:name, :basePay, :hourly, :timeCard) do
   def monthlyBasePay()
     if (hourly) then return 0 end
     return basePay
   end
  end
```

```
$employees = [
  Employee.new("직원A", 400, false, 0),
  Employee.new("직원A", 300, false, 0),
  Employee.new("직원C", 250, false, 0),
  Employee.new("아르바이트D", 1, true, 120),
  Employee.new("아르바이트E", 1, true, 120),
  Employee.new("아르바이트F", 1, true, 120),
]
```

```
def calculatePay(name)
  taxRate = getTaxRate()
  for each in $employees
    if (each.name == name) then employee = each; break end
  end
```

```
  pay = employee.calculatePay(taxRate)
  puts(describeResult(name, pay))
end
```

```
def sumOfBasePays()
  result = 0
  for each in $employees
    result += each.monthlyBasePay()
  end
  puts(result)
end
```

지금까지 살펴본 것처럼 추상 데이터 타입은 사람들이 세상을 바라보는 방식에 좀 더 근접해지도록 추상화는 수준을 향상 시킨다.

추상 데이터 타입 정의를 기반으로 객체를 생성하는 것은 가능하지만 여전히 데이터와 기능을 분리해서 바라본다는 점에 주의하라.

추상 데이터 타입은 말 그대로 시스템의 상태를 저장할 데이터를 표현한다.

추상데이터 타입으로 표현된 데이터를 이용해서 기능을 구현하는 핵심 로직은 추상 데이터 타입 외부에 존재한다.

리스코프가 이야기한 것처럼 추상 데이터 타입의 기본 의도를 프로그래밍 언어가 제공하는 타입처럼 동작하는 사용자 정의 타입을 추가할 수 있게 하는 것이다.

프로그래밍 언어의 관점에서 추상 데이터 타입은 프로그래밍 언어의 내장 데이터 타입과 동일하다.

클래스

클래스는 추상 데이터 타입인가?

- 명확한 의미에서 추상 데이터 타입과 클래스는 동일하지 않다.
- 가장 핵심적인 차이는 클래스는 상속과 다형성 지원하는 데 비해 추상 데이터 타입은 지원하지 못하다는 점이다.

  상속과 다형성을 지원하는 객체지향 프로그래밍과 구분하기 위해 상속과 다형성을 지원하지 않는 추상 데이터 타입 기반의 프로그래밍을 패러다임을 객체기반 프로그래밍이라고 부르기도 한다.

- 추상 데이터 타입은 타입을 추상화한 것이고 클래스는 절차를 추상화 한 것.
- 타입 추상화와 절차 추상화의 차이점을 이해하기 위해 먼저 추상 데이터 타입으로 물리적으로는 하나의 타입이지만 개념적으로 정규 직원과 아르바이트 지구언이라는 두 개의 개별적인 개념을 포괄하는 복합 개념이다.

추상 데이터 타입이 오퍼레이션을 기준으로 타입을 묶는 방법이라면 객체지향은 타입을 기준으로 오퍼레이션을 묶는다.

공통 로직을 제공할 수 있는 가장 간단한 방법을 공통 로직을 포함할 부모 클래스를 정의하고 두 직원 유형의 클래스가 부모 클래스를 상속받게 하는 것이다.

이제 클라이언트는 부모 클래스의 참조자에 대해 메시지를 전송하면 실제 클래스가 무엇인가에 따라 적절한 절차가 실행된다

동일한 메시지에 대해 서로 다르게 반응한다. 이것이 바로 다형성이다.

클라이언트의 관점에서 두 클래스의 인스턴스는 동일하게 보인다는 것에 주목하라.

실제로 내부에서 수행되는 절차는 다르지만 클래스를 이용한 다형성은 절차에 대한 차이점을 감춘다. 다시 말해 객체지향은 절차 추상화다.

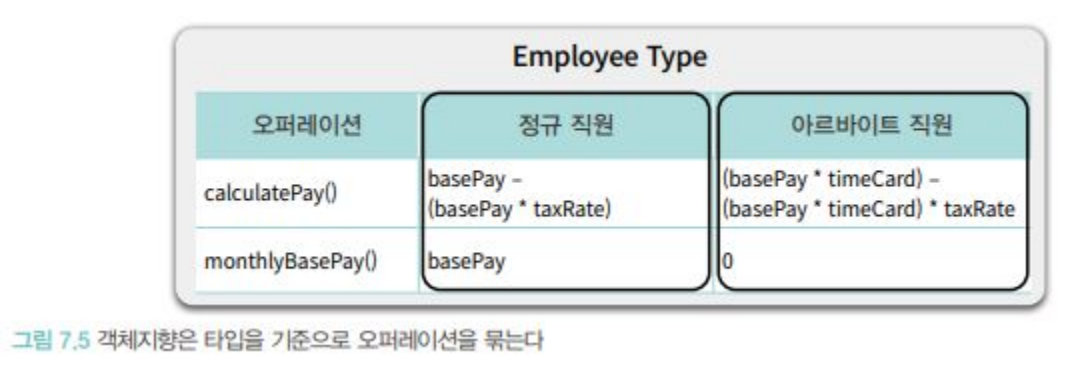

그림 7.5 객체지향은 타입을 기준으로 오퍼레이션을 묶는다

추상 데이터 타입은 오퍼레이션을 기준으로 타입들을 추상화한다. 클래스는 타입을 기준으로 절차들을 추상화와 분해의 관점에서 추상 데이터 타입과 클래스의 다른 점이다.

추상 데이터 타입에서 클래스로 변경하기

```
class Employee
  attr_reader :name, :basePay

  def initialize(name, basePay)
    @name = name
    @basePay = basePay
  end
```

```
  def calculatePay(taxRate)
    raise NotImplementedError
  end

  def monthlyBasePay()
    raise NotImplementedError
  end
end
```

```
class SalariedEmployee < Employee
  def initialize(name, basePay)
    super(name, basePay)
  end

  def calculatePay(taxRate)
    return basePay - (basePay * taxRate)
  end

  def monthlyBasePay()
    return basePay
  end
end
```

```
class HourlyEmployee < Employee
  attr_reader :timeCard
  def initialize(name, basePay, timeCard)
    super(name, basePay)
    @timeCard = timeCard
  end

  def calculatePay(taxRate)
    return (basePay * timeCard) - (basePay * timeCard) * taxRate
  end

  def monthlyBasePay()
    return 0
  end
end
```

```
def sumOfBasePays()
  result = 0
  for each in $employees
    result += each.monthlyBasePay()
  end
  puts(result)
end
```

```
$employees = [
  SalariedEmployee.new("직원A", 400),
  SalariedEmployee.new("직원B", 300),
  SalariedEmployee.new("직원C", 250),
  HourlyEmployee.new("아르바이트D", 1, 120),
  HourlyEmployee.new("아르바이트E", 1, 120),
  HourlyEmployee.new("아르바이트F", 1, 120),
]
```

변경을 기준으로 선택하라.

- 단순히 클래스를 구현단위로 사용한다는 것이 객체지향 프로그래밍을 한다는 것을 의미하지는 않는다.
- 타입을 기준으로 절차를 추상화하지 않았다면 그것은 객체지향 분해가 아니다. 비록 클래스를 사용하고 있더라도 말이다.
- 클래스가 추상 데이터 타입의 개념을 따르는지를 확인할 수 있는 가장 간단한 방법은 클래스 내부에 인스턴스의 타입을 표현하는 변수가 있는지를 살펴보는 것이다.
- 객체지향에서는 타입 변수를 이용한 조건문을 다형성으로 대체한다.
  클라이언트가 객체의 타입을 확인한 후 적절한 메서드를 호출하는 것이 아니라 객체가 메시지를 처리할 적절한 메서드를 선택한다.
- 모든 설계 문제가 그런 것처럼 조건문을 사용하는 방식을 기피하는 이유 변경 때문이다.

- 이처럼 기존 코드에 아무런 영향도 미치지 않고 새로운 객체 유형과 행위를 추가할 수 있는 객체지향의 특성을 개방-폐쇄 원칙 이라고 부른다.

- 대부분의 객체지향 서적에서는 추상 데이터 타입을 기반으로 어플리케이션을 설계하는 방식을 잘못된 것으로 설명한다.
- 설계는 변경과 관련된 것이다.
- 설계의 유용성은 변경의 방향성과 발생 빈도에 따라 결정된다.
  그리고 추상 데이터 타입과 객체지향 설계의 유용성은 변경의 방향성과 발생 빈도에 따라 결정된다. 그리고 추상 데이터 타입과 객체지향 설계의 유용성은 설계에 요구되는 변경의 압력이 타입추가 에 관한 것인지.

  아니면 오퍼레이션 추가에 관한 것인지 따라 달라진다.

  타입 추가라는 변경의 압력이 더 강한 경우에는 객체지향의 손을 들어줘야 한다.

  이에 반해 변경의 주된 압력이 오퍼레이션을 추가하는 것이라면 추상 데이터 타입의 승리를 선언해야한다.

새로운 타입을 빈번하게 추가해야 한다면 객체지향의 클래스 구조가 더 유용하다.

새로운 오퍼레이션을 빈번하게 추가해야 한다면 추상 데이터 타입을 선택하는 것이 현명한 판단이다.

변경의 축을 찾아라.

데이터-주도 설계

레베카 워프스브록은 추상 데이터 타입의 접근법을 객체지향 설계에 구현한 것을 데이터 주도 설계라고 부른다[Wirfs-Brock89]. 워프스브록이 제안한 책임 주도 설계는 데이터 주도 설계 방법을 개선하고자 하는 노력의 산물이었다. 티모시 버드(Timothy Budd)는 모듈과 추상 데이터 타입이 데이터 중심적인 관점(data centered view)을 취하는 데 비해 객체지향은 서비스 중심적인 관점(service centered view)을 취한다는 말로 둘 사이의 차이점을 깔끔하게 설명했다[Budd01].

협력이 중요하다