5 w.

OOP

Ivy

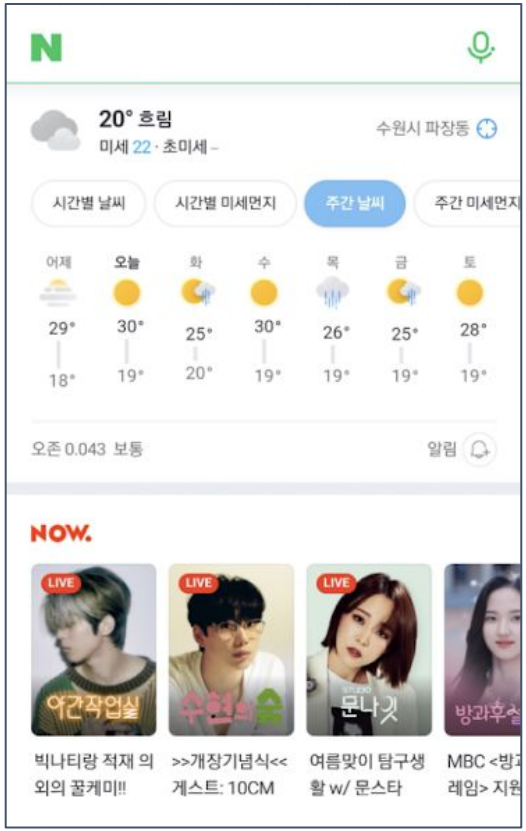
객체지향 사고방식

인간이 사물을 인식하는 방식

- 해
- 구름
- 풀밭
- 나무
- 집
- …

→ 물리적인 실체를 지닌 것을 하나의 단위로 인지할 수 있다

인간이 사물을 인식하는 방식



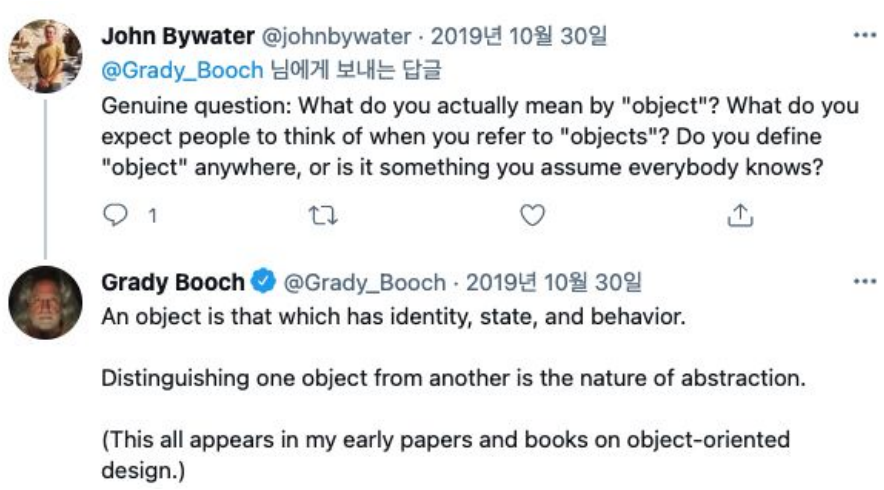
- 어제 날씨 / 오늘 날씨 / 내일 날씨
- NOW 방송 : 야간작업실 / 수현의 숲 / …

→ 물리적인 실체가 없더라도
개념적으로 구분할 수 있는 추상적인
사물도 하나의 단위로 인지할 수 있다

프로그래밍에서의 객체(Object)

Grady Booch(2007) “객체를 상태(state), 행동(behavior), 식별자(identity)를 지닌 실체로 보는 것이 가장 효과적이다”

- ‘객체지향의 사실과 오해’ 발췌



- Grady Booch on Twitter

기계로서의 객체(Object)

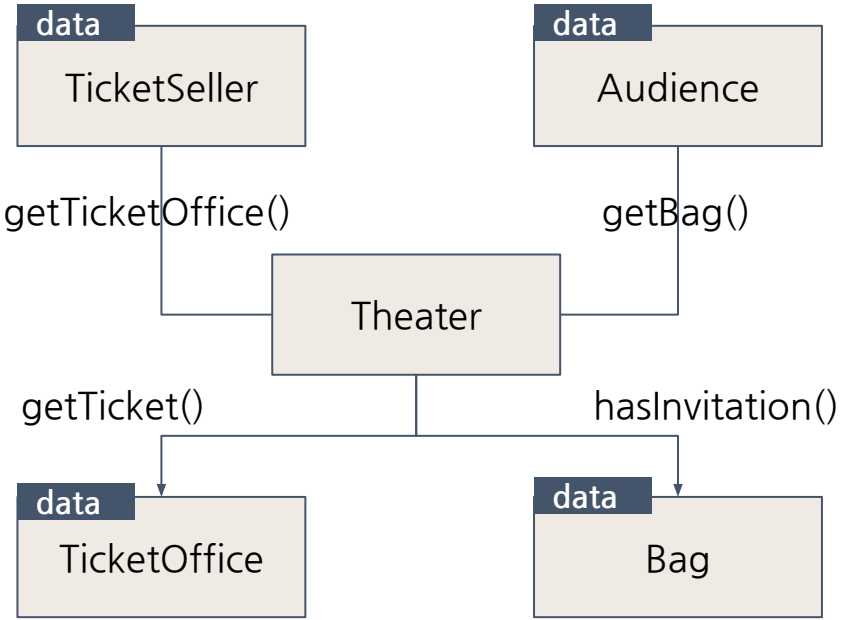
출처 : 탁상시계

(1) 상태
- 상태(시간, 온도, 습도)가 디스플레이 창에 출력된다
- 행동이 상태를 결정한다

(2) 행동
- SET / UP / DOWN 버튼을 누르면 시간(상태)를 변경할 수 있다
- 상단을 탭하면, 날짜 정보를 볼 수 있다.

(3) 식별자
- 여러 시계의 상태가 동일하더라도, 우리는 별개의 객체로 인식한다.

절차지향



출처: 오브젝트 (저자: 조영호) p. 28

```
public class Theater {
  private TicketSeller ticketSeller;

  public Theater(TicketSeller ticketSeller) {
    this.ticketSeller = ticketSeller;
  }

  public void enter(Audience audience) {
    if (audience.getBag().hasInvitation()) {
      Ticket ticket = ticketSeller.getTicketOffice().getTicket();
      audience.getBag().setTicket(ticket);
    } else {
      Ticket ticket = ticketSeller.getTicketOffice().getTicket();
      audience.getBag().minusAmount(ticket.getFee());
      ticketSeller.getTicketOffice().plusAmount(ticket.getFee());
      audience.getBag().setTicket(ticket);
    }
  }
}
```

절차지향

```
public class Theater {
  private TicketSeller ticketSeller;

  public Theater(TicketSeller ticketSeller) {
    this.ticketSeller = ticketSeller;
  }

  public void enter(Audience audience) {
    if (audience.getBag().hasInvitation()) {
      Ticket ticket = ticketSeller.getTicketOffice().getTicket();
      audience.getBag().setTicket(ticket);
    } else {
      Ticket ticket = ticketSeller.getTicketOffice().getTicket();
      audience.getBag().minusAmount(ticket.getFee());
      ticketSeller.getTicketOffice().plusAmount(ticket.getFee());
      audience.getBag().setTicket(ticket);
    }
  }
}
```

출처: 오브젝트 (저자: 조영호) 1장. 객체, 설계

- Audience, TicketSeller가 Theater의 통제를 받는다.
- Theater가 Audience, TicketSeller의 세부 내용을 알아야(기억해야) 한다.
- Audience, TicketSeller를 변경할 경우 Theater도 변경해야 한다.

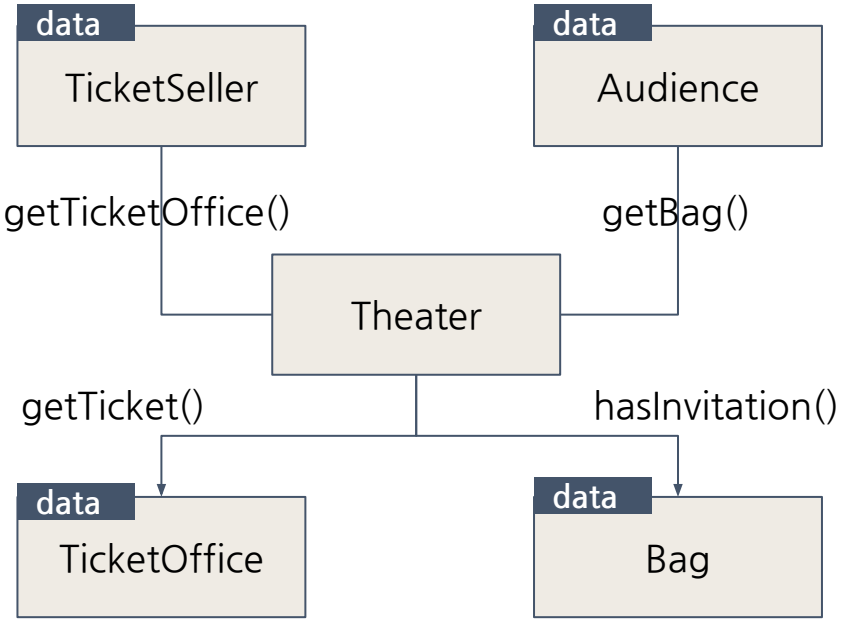
  → 관람객이 가방을 들고 있다는 가정이 바뀐다면?

객체 사이의 의존성(dependency),
어떤 객체가 변경될 때 그 객체에 의존하는 다른 객체도 함께 변경될 수 있다.
→ **과한 경우 결합도(coupling)가 높다**고 한다.

프로시저는 무엇인가?

- 루틴 : 어떤 절차를 묶어 호출 가능하도록 이름을 부여한 기능 모듈
  a. 프로시저 : 정해진 절차에 따라 내부의 상태를 변경하는 루틴
  b. 함수 : 어떤 절차에 따라 필요한 값을 계산해서 **반환**하는 루틴

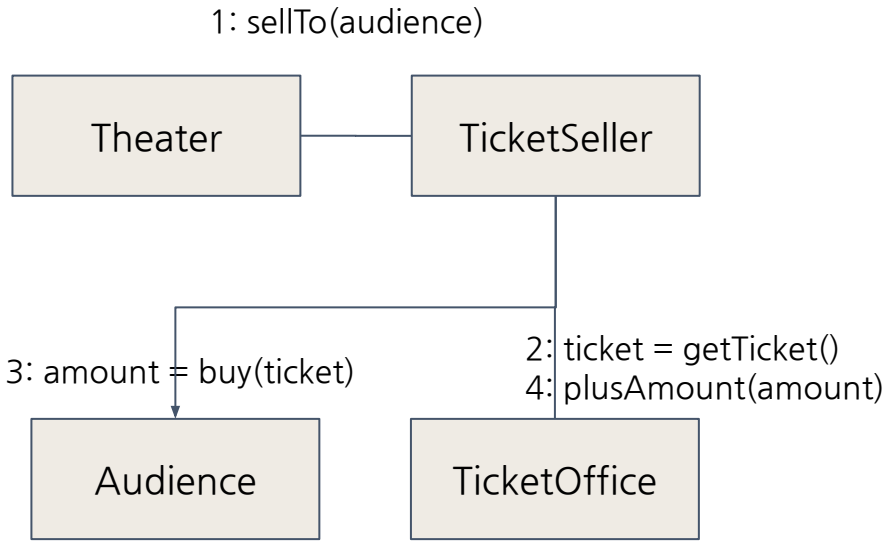
절차지향 프로그래밍 (Procedural Programming)



- 프로시저와 데이터를 별도의 모듈에 위치시키는 방식
- 프로시저가 필요한 모든 데이터에 의존해야하기 때문에 변경에 취약하다
- 여러 프로시저가 동일한 데이터를 공유하는 상황에서, 데이터가 변경되는 경우를 생각해보자

출처: 오브젝트 (저자: 조영호) p. 28

객체지향



출처: 오브젝트 (저자: 조영호) p. 28

```java
public class Theater {
  private TicketSeller ticketSeller;

  public Theater(TicketSeller ticketSeller) {
    this.ticketSeller = ticketSeller;
  }

  public void enter(Audience audience) {
    ticketSeller.sellTo(audience);
  }
}
```

객체지향
여러 객체가 협력하는 구조를 만드는 것
→ 객체 사이의 **결합도**를 낮추고, **변경하기 쉬운 코드**를 작성할 수 있도록 설계해야 한다

객체지향

```java
public class Theater {
  private TicketSeller ticketSeller;

  public Theater(TicketSeller ticketSeller) {
    this.ticketSeller = ticketSeller;
  }

  public void enter(Audience audience) {
    if (audience.getBag().hasInvitation()) {
      Ticket ticket = ticketSeller.getTicketOffice().getTicket();
      audience.getBag().setTicket(ticket);
    } else {
      Ticket ticket = ticketSeller.getTicketOffice().getTicket();
      audience.getBag().minusAmount(ticket.getFee());
      ticketSeller.getTicketOffice().plusAmount(ticket.getFee());
      audience.getBag().setTicket(ticket);
    }
  }
}
```

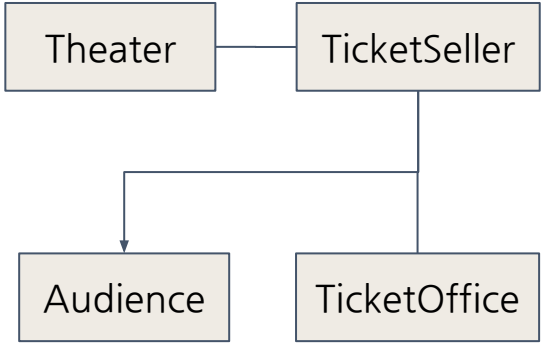
ticketSeller의 interface에만 의존한다

```java
public class Theater {
  private TicketSeller ticketSeller;

  public Theater(TicketSeller ticketSeller) {
    this.ticketSeller = ticketSeller;
  }

  public void enter(Audience audience) {
    ticketSeller.sellTo(audience);
  }
}
```

캡슐화(encapsulation)
객체 내부의 세부적인 사항을 감추는 것
→ 목적 : **변경하기 쉬운 객체**를 만든다
→ 결과 : 의존성을 제거할 수 있다.

출처: 오브젝트 (저자: 조영호) 1장. 객체, 설계

객체지향




```
public class Theater {
  private TicketSeller ticketSeller;

  public Theater(TicketSeller ticketSeller) {
    this.ticketSeller = ticketSeller;
  }

  public void enter(Audience audience) {
    if (audience.getBag().hasInvitation()) {
      Ticket ticket = ticketSeller.getTicketOffice().getTicket();
      audience.getBag().setTicket(ticket);
    } else {
      Ticket ticket = ticketSeller.getTicketOffice().getTicket();
      audience.getBag().minusAmount(ticket.getFee());
      ticketSeller.getTicketOffice().plusAmount(ticket.getFee());
      audience.getBag().setTicket(ticket);
    }
  }
}
```

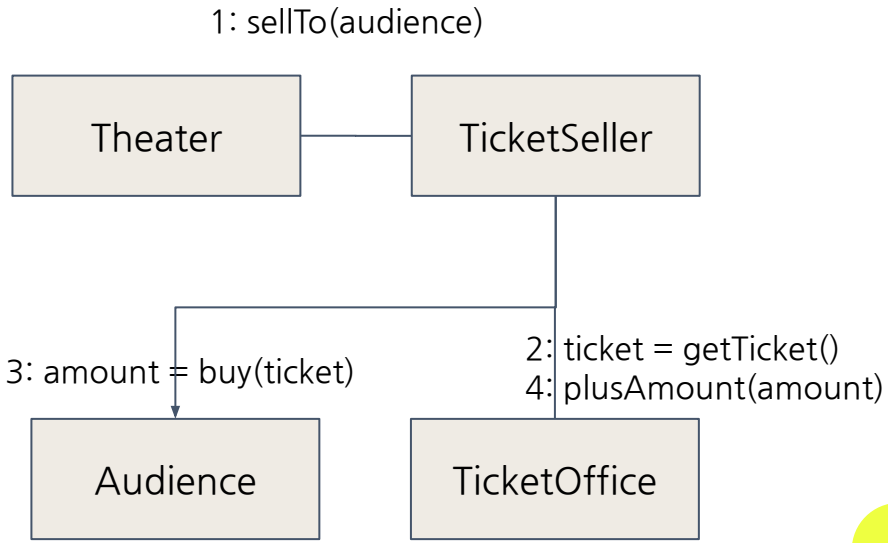
```
public class Theater {
  private TicketSeller ticketSeller;

  public Theater(TicketSeller ticketSeller) {
    this.ticketSeller = ticketSeller;
  }

  public void enter(Audience audience) {
    ticketSeller.sellTo(audience);
  }
}
```

```
public class TicketSeller {
  private TicketOffice ticketOffice;

  public TicketSeller(TicketOffice ticketOffice) {
    this.ticketOffice = ticketOffice;
  }

  public void sellTo(Audience audience) {
    ticketOffice.plusAmount(audience.buy(ticketOffice.getTicket()));
  }
}
```

```
public class Audience {
  private Bag bag;

  public Audience(Bag bag) {
    this.bag = bag;
  }

  public Long buy(Ticket ticket) {
    if (bag.hasInvitation()) {
      bag.setTicket(ticket);
      return 0L;
    } else {
      bag.setTicket(ticket);
      bag.minusAmount(ticket.getFee());
      return ticket.getFee();
    }
```

출처: 오브젝트 (저자: 조영호) 1장. 객체, 설계

객체지향 프로그래밍 (Object-Oriented Programming)



출처: 오브젝트 (저자: 조영호) p. 28

- 프로시저와 데이터를 동일한 모듈 내부에 위치하도록 프로그래밍하는 방식
- Theater는 TicketSeller에만 의존한다. TicketSeller 는 Audience에 대한 의존성이 추가됐지만 적절한 trade-off의 결과다.
- 책임이 여러 객체로 나누어진다

객체지향
여러 객체가 협력하는 구조를 만드는 것
→ 객체 사이의 **결합도**를 낮추고, **변경하기 쉬운 코드**를 작성할 수 있도록 설계해야 한다

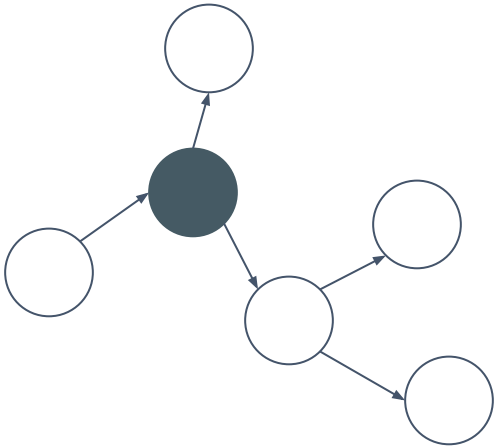
객체지향 설계

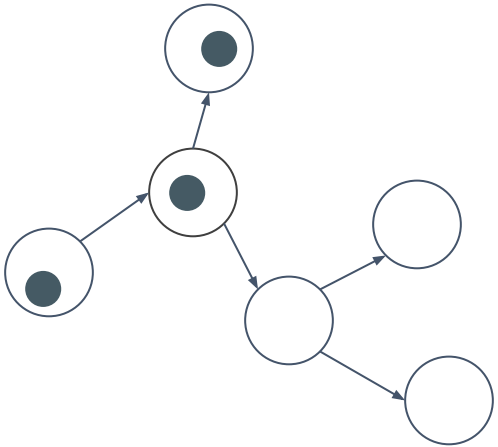
- 각 객체에 책임이 적절하게 분배된다.
- 불필요한 **의존성**을 제거해 객체 사이의 **결합도**를 낮춘다.
- 불필요한 세부사항을 객체 내부로 **캡슐화**한다. → 객체의 **자율성**을 높이고 **응집도** 높은 객체들의 공동체를 창조한다.

응집도

- 모듈에 포함된 내부 요소들이 연관돼 있는 정도
- 객체 또는 클래스에 얼마나 관련 높은 책임들을 할당했는지를 알 수 있는 지표
- 측정 방법 : 변경이 발생할 때 모듈 내부에서 발생하는 변경의 정도

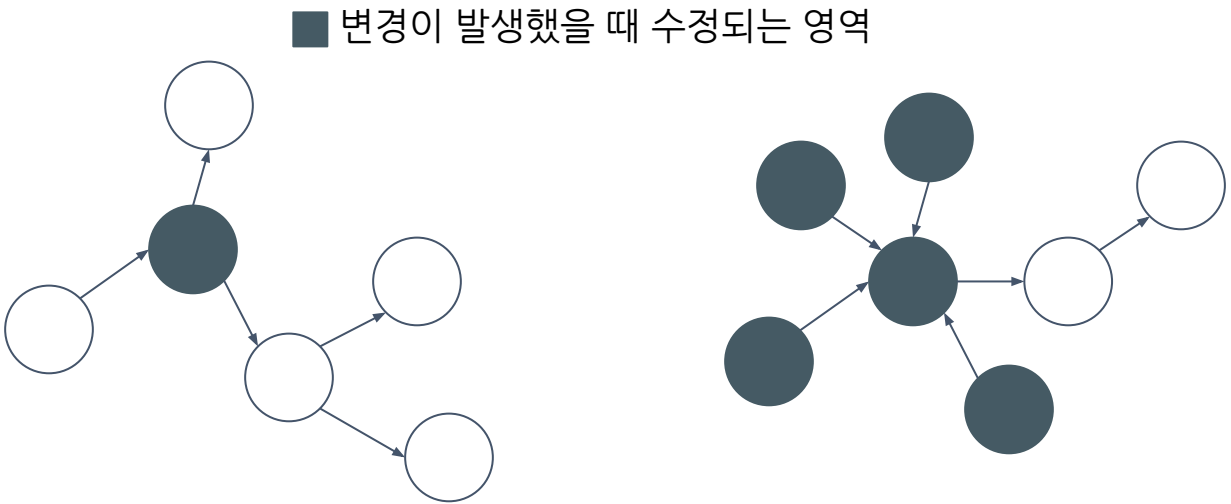
■ 변경이 발생했을 때 수정되는 영역





결합도

- 의존성의 정도. 다른 모듈에 대해 얼마나 많은 지식을 갖고 있는지를 나타낸다.
- 객체 또는 클래스가 협력에 필요한 적절한 수준의 관계만을 유지하고 있는지 알 수 있는 지표
- 측정 방법 : 한 모듈이 변경되기 위해서 다른 모듈의 변경을 요구하는 정도



추상화 과정

추상화(Abstraction)

불필요한 정보를 제거하고 현재의 문제 해결에 필요한 핵심만 남긴다.

- 사물들 간의 공통점은 취하고, 차이점은 버리는 **일반화**를 통해 단순하게 만든다.
- 중요한 부분을 강조하기 위해 불필요한 세부 사항을 제거해 단순하게 만든다.

→ 객체지향 패러다임은 '(역할과 책임을 수행하는)객체'라는 추상화를 통해 복잡성을 극복한다.

출처: 오브젝트 p. 217, 객체지향의 사실과 오해 p. 77

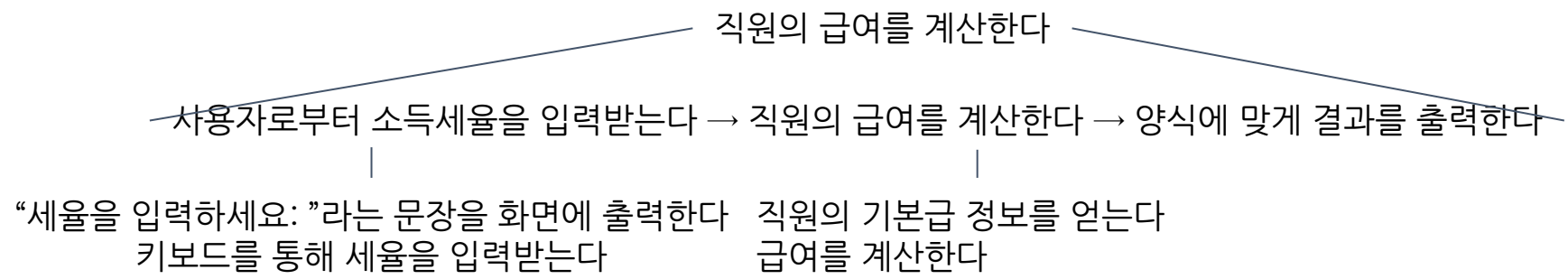
프로시저 추상화(procedure abstraction)

(1) 소프트웨어가 무엇을 해야하는지 추상화 한다

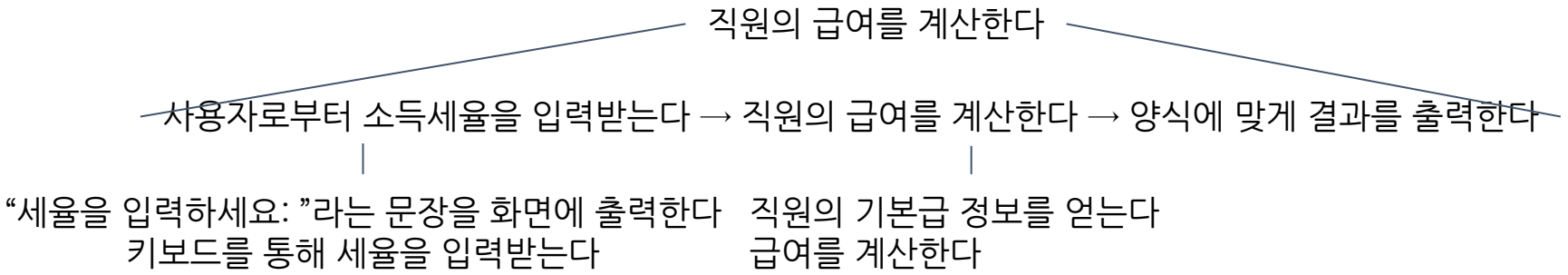
→ 기능 분해(functional decomposition), 알고리즘 분해

(2) 프로시저 중심의 기능 분해 관점에서, 시스템은 필요한 더 작은 작업으로 분해될 수 있는 하나의 커다란 메인 함수다

- 상위기능은 더 간단하고, 더 구체적이며 덜 추상적인 하위 기능의 집합으로 분해된다.



프로시저 추상화(procedure abstraction)

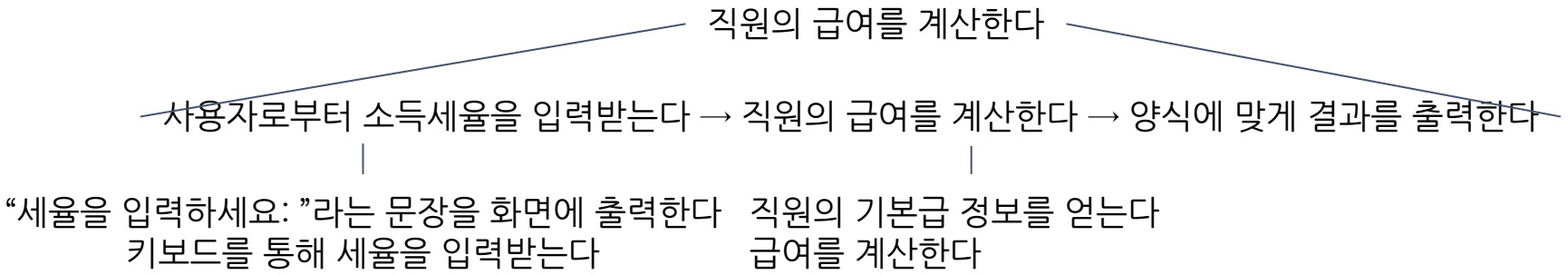


하향식 접근법(Top-Down Approach)

- 필요한 기능을 먼저 생각하고, 이 기능을 분해하는 과정에서 필요한 데이터의 종류 & 저장방식을 결정한다.

기능이 추가되거나, 요구사항이 변경된다면 어떻게 될까?

프로시저 추상화(procedure abstraction)



- 모든 직원들의 기본급의 총합을 구하는 기능 추가 → 어느 위치에 추가할 것인가?
- 입력 UI 변경 / 추가 → 비즈니스 로직과 UI 로직이 섞여 있는데, 가능할까?
- 처음부터 함수들의 실행 순서를 결정한다 → 기능이 추가/변경될 때마다 함수의 제어구조를 바꿔야한다

상위 함수가 강요하는 문맥에 절차를 구성하는 다른 함수들이 강하게 결합된다 → 강한 결합, 변경에 취약

데이터 추상화 (data abstraction)

(1) 소프트웨어가 무엇을 알아야 하는지를 추상화한다.

(2) 2가지 관점

- 데이터를 중심으로 타입을 추상화(type abstraction) → 추상 데이터 타입
- 데이터를 중심으로 프로시저를 추상화(procedure abstraction) → 객체지향

데이터 추상화 - 추상 데이터 타입(Abstract Data Type)

(1) type
- 변수에 저장할 수 있는 내용물의 종류와, 변수에 적용될 수 있는 연산의 가짓수

  ex. 정수 타입의 변수 : 정수값으로 간주하고, 수행할 수 있는 연산이 결정된다

(2) 추상 데이터 타입
- 상태를 저장할 데이터를 정의한다.
- 대표적인 타입이 다수의 세부적인 타입을 감춘다.
- **오퍼레이션을 기준**으로 타입을 묶는다

Employee Type

오퍼레이션	정규 직원	아르바이트 직원
calculatePay	basePay * (1 - taxRate)	timeBasedPay * (1 - taxRate)
calculateBonus	basePay * bonusRate	0

데이터 추상화 - 데이터 중심 프로시저 추상화(객체지향)

- **타입을 기준**으로 오퍼레이션을 묶는다
- 타입에 대한 클래스를 정의하고, 각 클래스가 오퍼레이션을 구현한다.
- 공통 로직을 제공하는 부모 클래스를 상속받는다면,
  - 동일한 메시지에 대해 서로 다르게 처리할 수 있다. → **다형성**

오퍼레이션	Employee Type 정규 직원	아르바이트 직원
calculatePay	basePay * (1 - taxRate)	timeBasedPay * (1 - taxRate)
calculateBonus	basePay * bonusRate	0

클래스

- 대부분의 객체지향 언어는 클래스라는 도구를 제공한다
- 객체지향에서의 클래스 : 데이터 추상화와 프로시저 추상화를 함께 포함한 클래스를 이용해 시스템을 분해한다

- 타입을 기준으로 프로시저를 추상화하지 않았다면, 객체지향 분해가 아니다.

객체지향 설계는 항상 좋은가?

- 설계는 코드 배치 방법이며, 설계가 필요한 이유는 변경에 대비하기 위함이다.
- 설계의 유용성은 변경의 방향성과 발생 빈도에 따라 결정된다.
    - 새로운 타입을 빈번하게 추가해야 하는 경우, 객체지향의 클래스 구조가 유용하다.