# Clean Code

Chapter 01. 깨끗한 코드
Chapter 02. 의미 있는 이름

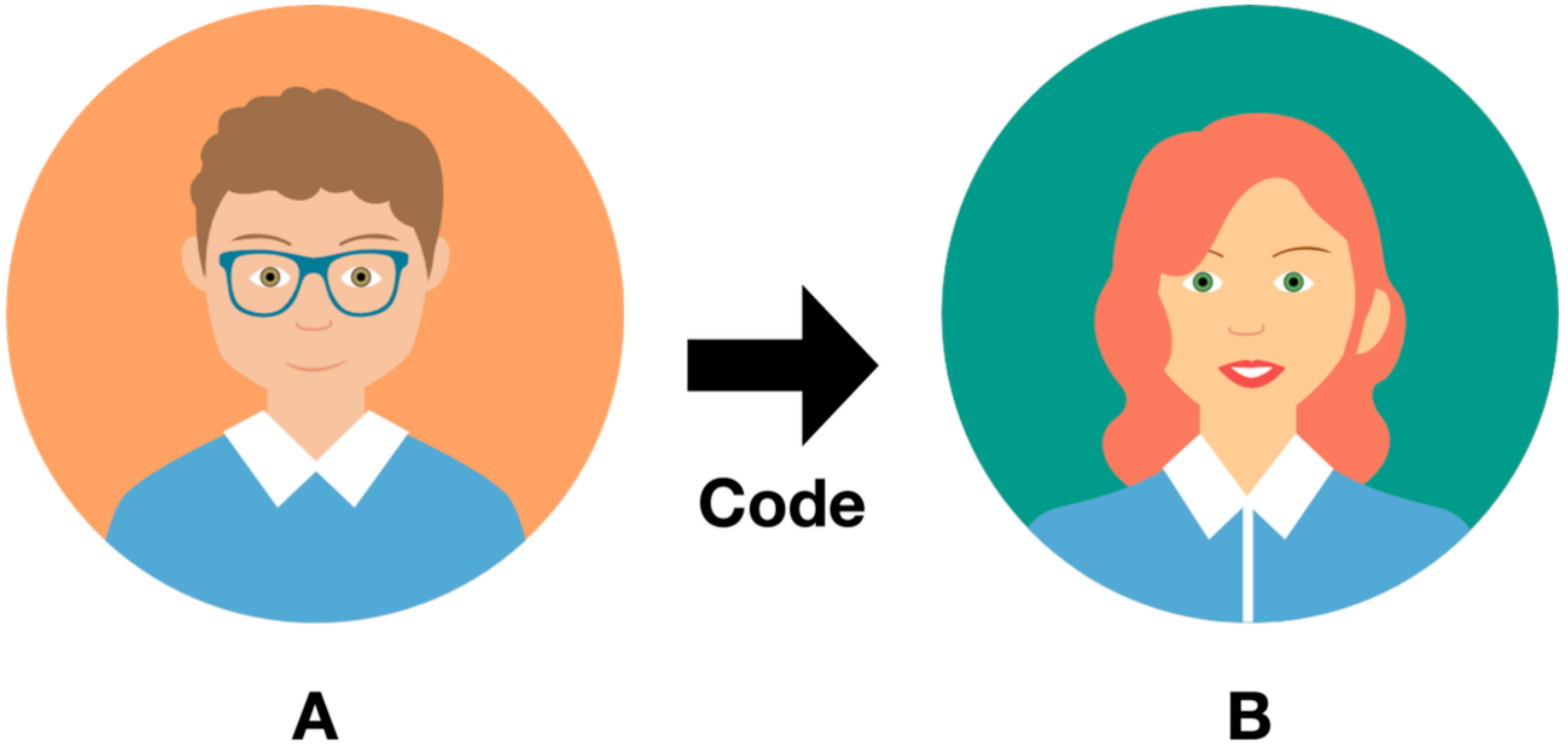
# 1

# 깨끗한 코드

# 깨끗한 코드

코드 = 커뮤니케이션

설명을 통하지 않고 코드로만 대화해도 이해할 수 있어야 한다.



# 깨끗한 코드

나쁜 코드

- 일정에 맞추기 위해 나쁜 코드를 방치하고 '나중에 고쳐야지'
  - 나중은 절대 오지 않는다. – 르블랑의 법칙
- 나쁜 코드가 쌓일수록 팀 생산성은 떨어진다.(매번 코드를 해독해야 함)
- 결국 좋은 코드를 사수하는 일은 프로그래머의 책임이다.
- 기한을 맞추는 유일한 방법은 언제나 코드를 깨끗하게 유지하는 습관
- 깨끗한 코드와 나쁜 코드를 보고 구분하는 것은 대부분의 사람들이 할 수 있다.
  - 하지만 구분할 줄 안다고 깨끗한 코드를 작성할 줄 안다는 것은 아니다.
  - 나쁜 코드를 좋은 코드로 바꾸는 전략도 파악해야 한다.

# 깨끗한 코드

## 깨끗한 코드란

- Bjarne Stroustru : 논리가 간단, 의존성 최소한, 철저한 오류처리, 한 가지에 집중하는 코드
- Grady Booch : 코드는 추측이 아닌 설계자의 의도가 바로 드러나야 한다.
- Dave Thomas : 다른 사람이 고치기 쉬운 코드, 테스트 케이스가 있는 코드
- Michael Feathers : 주의를 기울인 코드
- Ron Jeffries : 중복이 없고, 한 기능만 수행하고, 표현력이 높고, 작게 추상화된 코드
- Ward Cunningham : 읽으면서 짐작한 대로 돌아가는 코드

# 2

# 의미 있는 이름

# 의미 있는 이름

## 의도를 분명히 밝혀라

### Original Bad Code

```
1   // Bad
2   List<int[]> theList;
3
4   public List<int[]> getThem() {
5       List<int[]> list1 = new ArrayList<int[]>();
6       for (int[] x : theList)
7           if (x[0] == 4)
8               list1.add(x);
9       return list1;
10  }
```

### Refactoring 1

```
1   List<int[]> gameBoard;
2   static final int STATUS_VALUE = 0;
3   static final int FLAGGED = 4;
4
5   public List<int[]> getFlaggedCells() {
6       List<int[]> flaggedCells = new ArrayList<>();
7       for (int[] cell : gameBoard)
8           if (cell[STATUS_VALUE] == FLAGGED)
9               flaggedCells.add(cell);
10      return flaggedCells;
11  }
```

# 의미 있는 이름

## 의도를 분명히 밝혀라

Refactoring 1

```
1   List<int[]> gameBoard;
2   static final int STATUS_VALUE = 0;
3   static final int FLAGGED = 4;
4
5   public List<int[]> getFlaggedCells() {
6       List<int[]> flaggedCells = new ArrayList<>();
7       for (int[] cell : gameBoard)
8           if (cell[STATUS_VALUE] == FLAGGED)
9               flaggedCells.add(cell);
10      return flaggedCells;
11  }
```

Refactoring 2

```
1   public class Cell {
2       private static final int STATUS_VALUE = 0;
3       private static final int FLAGGED = 4;
4       private int[] value;
5
6       public boolean isFlagged() {
7           return value[STATUS_VALUE] == FLAGGED;
8       }
9   }
10
11
12  List<Cell> gameBoard;
13
14  public List<Cell> getFlaggedCells() {
15      List<Cell> flaggedCells = new ArrayList<>();
16      for (Cell cell : gameBoard)
17          if (cell.isFlagged())
18              flaggedCells.add(cell);
19      return flaggedCells;
20  }
```

# 의미 있는 이름

## 그릇된 정보를 피하라, 의미 있게 구분하라

- 컨테이너 유형을 이름에 넣지 마라
- 유사한 개념은 유사한 표기법을 사용하라(코드 자동 완성 기능을 사용하기 위해)
- 비슷해 보이는 문자를 주의하라
  - I VS 1 (소문자 L과 숫자 1)
  - O vs 0 (대문자 O와 숫자 0)

- 불용어(noise word)를 추가한 이름은 아무 추가 정보도 제공하지 못한다.
  - info, data, a, an, the, 변수타입
  - Product, ProductInfo, ProductData
  - NameString, Name,
  - moneyAmount, money
  - theMessage, message

# 의미 있는 이름

그릇된 정보를 피하라, 의미 있게 구분하라

```java
// Bad
public static void copyChars(char[] a1, char[] a2) {
    for (int i = 0; i < a1.length; i++) {
        a2[i] = a1[i];
    }
}
```

```java
// Good
public static void copyChars(char[] source, char[] destination) {
    for (int i = 0; i < source.length; i++) {
        destination[i] = source[i];
    }
}
```

# 의미 있는 이름

## 발음하기 쉬운 이름을 사용하라

```
// Bad
class DtaRcrd102 {
    private Date genymdhms;
    private Date modymdhms;
    private final String pszqint = "102";
}
```

generate year, month, day, hour, minute, second

```
// Good
class Customer {
    private Date generatonTimestamp;
    private Date modificationTimestamp;
    private final String recordId = "102";
}
```

# 의미 있는 이름

## 검색하기 쉬운 이름을 사용하라

```
// Bad
int s = 0;
for (int j = 0; j < 34; j++) {
  s += (t[j] * 4) / 5;
}
```

```
// Good
int realDaysPerIdealDay = 4;
final int WORK_DAYS_PER_WEEK = 5;
final int NUMBER_OF_TASKS = 34;
int sum = 0;
for (int j = 0; j < NUMBER_OF_TASKS; j++) {
  int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
  int realTaskWeeks = (realTaskDays / WORK_DAYS_PER_WEEK);
  sum += realTaskWeeks;
}
```

# 의미 있는 이름

## 인코딩을 피하라

- 헝가리식 표기법을 사용하지 마라
- 인터페이스 클래스와 구현 클래스(다음 슬라이드)

| 접두어 | 데이터 타입 |
|---|---|
| b | byte, boolean |
| n | int, short |
| i | int, short (주로 인덱스로 사용) |
| c | int, short (주로 크기로 사용) |
| l | long |
| f | float |

| 접두어 | 데이터 타입 |
|---|---|
| g_ | 네임스페이스의 글로별 변수 |
| m_ | 클래스의 멤버 변수 |
| s_ | 클래스의 static 변수 |
| c_ | 함수의 static 변수 |

# 의미 있는 이름

인코딩을 피하라 – 인터페이스와 구현 클래스(심화학습)

## SOLID 원칙

1. **단일 책임 원칙 (Single Responsiblity Principle; SRP)**
   모든 클래스는 하나의 책임만 가져야 한다. 클래스는 그 책임을 캡슐화해야 한다.

2. **개방-폐쇄 원칙 (Open Closed Principle; OCP)**
   기존의 코드를 변경하지 않으면서, 기능을 추가할 수 있도록 설계되어야 한다.

3. **리스코프 치환 원칙 (Liskov Substitution Principle; LSP)**
   자식 클래스는 언제나 부모 클래스를 대체할 수 있어야 한다.

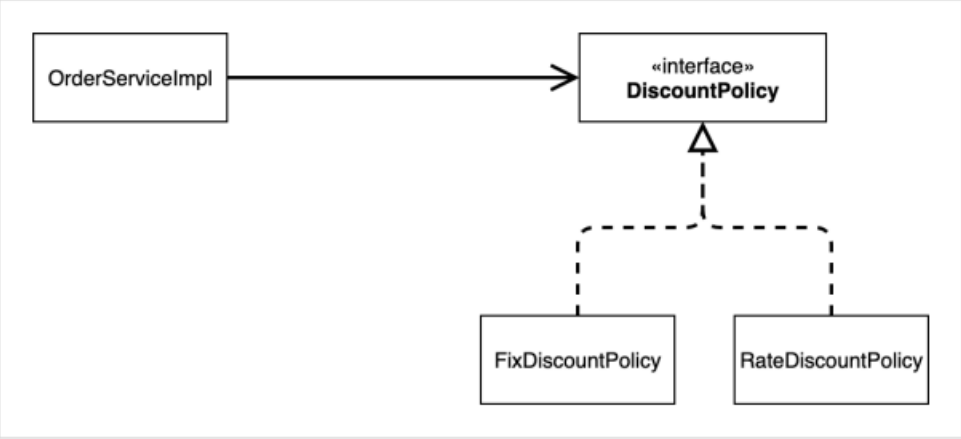
4. **인터페이스 분리 원칙 (Interface Segregation Principle; ISP)**
   구현 클래스는 자신이 사용하지 않을 인터페이스는 구현하지 않아야 한다.

5. **의존 역전 원칙 (Dependency Inversion Principle; DIP)**
   클라이언트는 구현 클래스가 아닌, 인터페이스에 의존해야 한다.

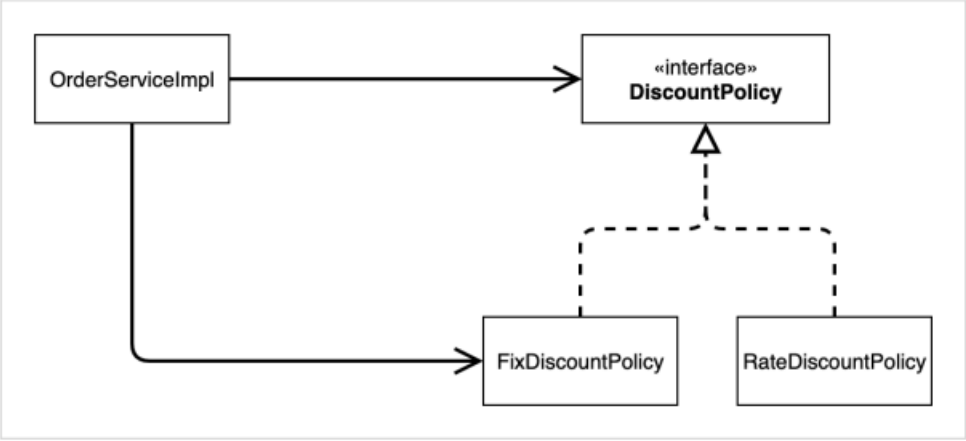
# 의미 있는 이름

## 인코딩을 피하라 – 인터페이스와 구현 클래스(심화학습)

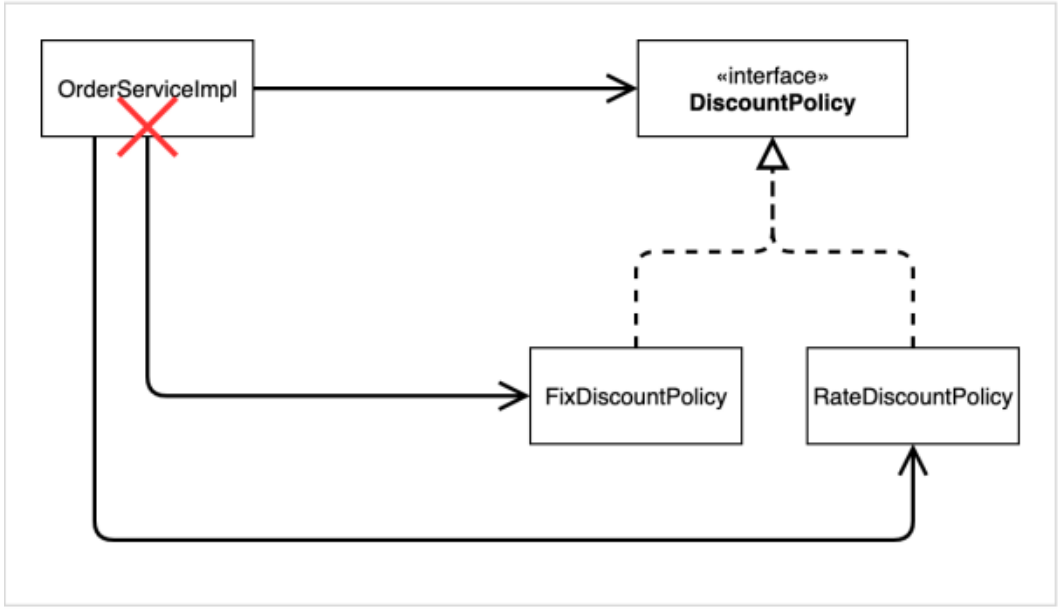
Expected




Actually : DIP 위반




정책 변경 : OCP 위반



# 의미 있는 이름

## 인코딩을 피하라 – 인터페이스와 구현 클래스(심화학습)

```
public class OrderService {

    private final DiscountPolicy discountPolicy = new RateDiscountPolicy();

    public void discountPrice(int price) {
        System.out.println("= 주문 금액 : " + price);
        int actualPrice = discountPolicy.discountPrice(price);
        System.out.println("= 실결제 금액 : " + actualPrice);
    }
}
```

```
public interface DiscountPolicy {
    int discountPrice(int price);
}
```

```
public class RateDiscountPolicy implements DiscountPolicy {
    @Override
    public int discountPrice(int price) {
        return (int) (price * 0.9);
    }
}
```

```
public class FixDiscountPolicy implements DiscountPolicy {
    @Override
    public int discountPrice(int price) {
        return price - 1000;
    }
}
```

# 의미 있는 이름

## 인코딩을 피하라 – 인터페이스와 구현 클래스(심화학습)

```
public class OrderService {

    private final DiscountPolicy discountPolicy;

    public OrderService(DiscountPolicy discountPolicy) {
        this.discountPolicy = discountPolicy;
    }

    public void discountPrice(int price) {
        System.out.println("= 주문 금액 : " + price);
        int actualPrice = discountPolicy.discountPrice(price);
        System.out.println("= 실결제 금액 : " + actualPrice);
    }
}
```

Dependency Injection!


스프링 빈 저장소

| 빈 이름 | 빈 객체 |
|---|---|
| memberServiceImpl | MemberServiceImpl@x01 |
| orderServiceImpl | OrderServiceImpl@x02 |
| memoryMemberRepository | MemoryMemberRepository @x03 |
| rateDiscountPolicy | RateDiscountPolicy@x04 |

```
public class Main {
    public static void main(String[] args) {
        OrderService orderService = new OrderService(new RateDiscountPolicy());
        orderService.discountPrice(10000);
    }
}
```

# 의미 있는 이름

## 클래스 이름과 메서드 이름

### 클래스, 객체: 명사, 명사구

좋은 예) Customer, WikiPage, Account, AddressParser

나쁜 예) Manager, Processor, Data, Info

### 메서드 이름: 동사, 동사구

좋은 예) postPayment, deletePage, save

접근자, 변경자, 조건자에는 get, set, is을 변수명 앞에 붙인다.

# 의미 있는 이름

클래스 이름과 메서드 이름

생성자 오버로딩할 때는 정적 팩토리 메서드를 사용한다.
이때 메서드명은 인수를 설명하는 이름을 사용한다.

```
1   public class Cell {
2       double value;
3
4       private Cell(double value) {
5           this.value = value;
6       }
7
8       public static Cell FromRealNumber(double x) {
9           return new Cell(x);
10      }
11  }
12
13  Cell cell = Cell.FromRealNumber(23.0);
```

# 의미 있는 이름

## 기발한 이름은 피하라

기발하거나 재미난 이름보다는 명료한 이름을 선택하라

## 한 개념에 한 단어를 사용하라

이름이 다르면 독자는 클래스와 타입이 다르다고 생각한다.

예) controller, manager, driver을 동일 개념으로 사용

## 말 장난을 하지 마라

다른 개념에 같은 단어를 사용하지 마라

값 2개 더하기(또는 잇기) – add()　　집합에 값 하나 추가하기 – add()

## 해법 영역에서 가져온 이름을 사용하라

알고리즘 이름, 패턴 이름, 수학 용어 등은 사용해도 좋다.(ex: JobQueue)

적절한 프로그래머 용어가 없다면 문제 영역(domain)에서 이름을 가져와라

# 의미 있는 이름

## 의미 있는 맥락을 추가하라

```
public void enterAddress(String city, String street, String zipcode) {
    // do something
}
```

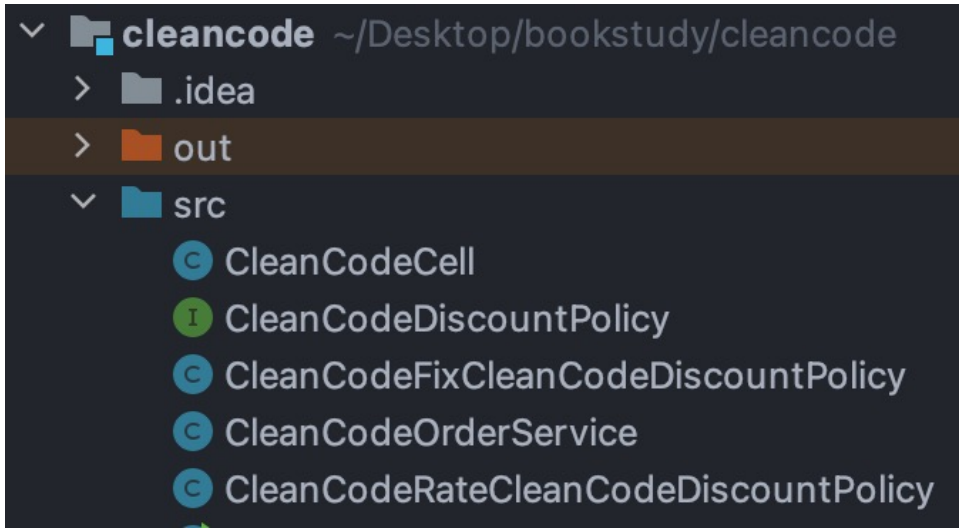
---

```
public class Address {
    private String city;
    private String street;
    private String zipcode;
}
```

```
public void enterAddress(Address address) {
    // do something
}
```

# 의미 있는 이름

## 불필요한 맥락을 없애라




```
public void function() {
    Address accountAddress = new Address();
    Address customerAddress = new Address();
}
```

# 적용

```
public List<PostManagerDto> findPostList() {
    List<Post> posts = postRepository.findAll(Sort.by(Sort.Direction.ASC, ...properties: "date"));
    List<PostManagerDto> ret = new ArrayList<>();
    List<String> dates = SundayDate.dates;

    int listPoint = 0;
    for (String date : dates) {
        if (posts.size() > listPoint) {
            Post post = posts.get(listPoint);
            if (post.getDate().equals(LocalDate.parse(date, DateTimeFormatter.ISO_DATE))) {
                ret.add(PostManagerDto.builder()
                        .id(post.getId())
                        .title(post.getTitle())
                        .date(date)
                        .isExist(true)
                        .build());
                listPoint++;
                continue;
            }
        }
        ret.add(PostManagerDto.builder()
                .date(date)
                .isExist(false)
                .build());
    }
    return ret;
}
```

# 적용

```
public List<PostSummaryDto> findPostSummaryDtoList() {
    final String sortProperty = "data";
    List<Post> posts = postRepository.findAll(Sort.by(Sort.Direction.ASC, ...properties: sortProperty));
    List<String> sundayDates = SundayDate.dates;

    List<PostSummaryDto> postSummaryDtoList = new ArrayList<>();
    int postsIndex = 0;
    for (String sundayDate : sundayDates) {
        if (postsIndex < posts.size()) {
            Post post = posts.get(postsIndex);
            if (post.getDate().equals(LocalDate.parse(sundayDate, DateTimeFormatter.ISO_DATE))) {
                postSummaryDtoList.add(PostSummaryDto.createExistPost(post.getId(), post.getTitle(), sundayDate));
                postsIndex++;
                continue;
            }
        }
        postSummaryDtoList.add(PostSummaryDto.createNotExistPost(sundayDate));
    }

    return postSummaryDtoList;
}
```