Chapter 2

목차

- 변화하는 요구사항 복사
- 행동 파라미터화
- 익명 클래스
- 람다 표현
- `Comparator`, `Runnable` , `GUI`

`동작 매개변수화` 는 빈번한 요구 사항을 처리하기에 적합한 소프트웨어 패턴이다. 실행하지 않고 코드 블럭을 만들 수 있는 기능이다. 메소드의 동작이 파라미터로 들어가는 방법이다.

Example

1. `filtering`

```
enum Color {RED, GREEN}

public static List<Apple> filterGreenApples(List<Apple> inventory) {
        List<Apple> result = new ArrayList<>();
        for(Apple apple: inventory){
         if( GREEN.equals(apple.getColor() ) {
                 result.add(apple);
         }
        }
        return result;
}
```

`GREEN.equals(apple.getColor()` 는 선택 조건을 보여준다. 이 값은 `enum` 안의 값이 해당된다. 만약 선택 조건이 변경된다면 메소드 이름을 변경해야하고, 만일 다중 조건이 필요하다면 좋은 선택이 아니다. 조건을 추상화해야 하는 것이다.

2. `parameterizing`

`조건을 파라미터화` 해서 변화에 유연하게 대응할 수 있다.

```
public static List<Apple> filterApplesByColor(List<Apple> inventory,
Color color) {
 List<Apple> result = new ArrayList<>();
 for (Apple apple: inventory) {
         if ( apple.getColor().equals(color) ) {
                 result.add(apple);
         }
 }
 return result;
}
```

```
// Use Example
List<Apple> greenApples = filterApplesByColor(inventory, GREEN);
List<Apple> redApples = filterApplesByColor(inventory, RED);
```

만약 조건이 여러개라면 어떨까?

```
public static List<Apple> filterApplesByWeight(List<Apple> inventory,
int weight) {
 List<Apple> result = new ArrayList<>();
 for (Apple apple: inventory){
         if ( apple.getWeight() > weight ) {
                 result.add(apple);
         }
 }
 return result;
}
```

이 경우에는 각 조건에 대해 비교 작업이 이루어져야 한다. 또, 각 작업에 대해서 파라미터 교체 작업을 모두 해주어야 한다. 여러 조건을 메소드 한개로 모으고, `filter` 라는 작업을 해주면 된다.

3. `filtering`

```
public static List<Apple> filterApples(List<Apple> inventory, Color color,
 int weight, boolean flag) {
         List<Apple> result = new ArrayList<>();
         for (Apple apple: inventory) {
         if ( (flag && apple.getColor().equals(color)) ||
                                 (!flag && apple.getWeight() > weight) ){
                 result.add(apple);
         }
 }
 return result;
}
```

이 코드는 바람직하지 못하다. 먼저, 읽기에 적합하지 않다. true와 false가 뜻하는 바를 알 수 없다. 그리고, 명시된 조건 외에 다른 조건이 요구된다면 변경 작업이 생긴다. 따라서 여러 복제가 가능한 `filter` 메소드를 사용해야 한다.

동작 파라미터화

```
// interface
public interface ApplePredicate{
 boolean test (Apple apple);
}

// method
public class AppleHeavyWeightPredicate implements ApplePredicate {
 public boolean test(Apple apple) {
         return apple.getWeight() > 150;
```

```
 }
}
public class AppleGreenColorPredicate implements ApplePredicate {
 public boolean test(Apple apple) {
         return GREEN.equals(apple.getColor());
 }
}
```

이 방법은 필터 메소드와 다른 동작을 보여준다. 디자인 패턴 중 전략 패턴 과 유사하다. 알고리즘의 집합을 정의하고, 각 알고리즘을 캡슐화해서 선택하게 한다.

이렇게 하면 여러 동작을 파라미터화 해서 내부에서는 다른 동작을 할 수 있다. 로직과 컴포넌트를 분리시켜 각 요소에 다른 동작을 할 수 있다.

추상화

```
public static List<Apple> filterApples(List<Apple> inventory,
 ApplePredicate p) {
        List<Apple> result = new ArrayList<>();
        for(Apple apple: inventory) {
                if(p.test(apple)) {
                        result.add(apple);
                }
        }
        return result;
}
```

이 코드는 처음보다 훨씬 유연하고, 동시에 읽기도 쉽다. 이제 `ApplePredicate` 객체를 각각 다르게 만들고 `filterApples` 에 전달할 수 있다.

```
public class AppleRedAndHeavyPredicate implements ApplePredicate {
        public boolean test(Apple apple){
                return RED.equals(apple.getColor())&& apple.getWeight() > 150;
        }
}
List<Apple> redAndHeavyApples =
                                                        filterApples(inventory,
new AppleRedAndHeavyPredicate());
```

동작 파라미터화 로직과 콜렉션을 분리하여 각 콜렉션 요소마다 적용할 수 있는 장점이 있다. 결과적으로, 메소드를 재사용하고 다른 동작까지 적용할 수 있다.

추상 클래스

추상 클래스 는 로컬 클래스 (블록으로 선언된 클래스)와 비슷하다. 추상 클래스는 일시적인 클래스를 동시에 선언하게 해준다. 즉, 임시 클래스를 만들어주는 것이다.

```
List<Apple> redApples = filterApples(inventory, new ApplePredicate() {
        public boolean test(Apple apple){ // 메소드를 추상 클래스로 파라미터화해서 전달
                return RED.equals(apple.getColor());
        }
});
```

추상클래스는 주로 GUI 애플리케이션에서 이벤트-핸들러 객체에 사용된다.

복잡성 은 좋지 않다. 언어의 능력을 떨어트릴 뿐만 아니라 코드를 작성하거나 유지하는데도 시간이 오래걸린다. 또한 읽기에도 좋지 않다.

좋은 코드 란 이해하기 쉬워야 한다. 추상 클래스가 아무리 인터페이스 중복 선언에 대해 복잡성을 낮춘다 해도, 여전히 만족스럽지 않다. 간단한 코드라 해도 여전히 객체를 만들고 새로운 동작을 하는 메소드를 선언하고 있다. 이상적으로 우리는 동작의 매개변수화 를 쓰도록 하고있다. 이는 코드를 더 적응되고 유연한 대처를 하게 한다.

람다식 표현

앞에서 본 코드는 람다식으로 다음과 같이 표현할 수 있다.

```
List<Apple> result =
 filterApples(inventory, (Apple apple) -> RED.equals(apple.getColor()));
```

이전 코드보다 훨씬 간결해졌다. 이런 표현 방식은 문제 상태에 맞게 표현한다.

List 타입의 추상화

앞의 코드는 한 가지 타입의 객체에만 유효하다. `List` 를 추상화 하면 다음과 같은 표현도 가능하다.

```
public interface Predicate<T> {
        boolean test(T t);
}

public static <T> List<T> filter(List<T> list, Predicate<T> p) {
        List<T> result = new ArrayList<>();
        for(T e: list) {
                if(p.test(e)) {
                        result.add(e);
                }
        }
        return result;
}
```

그리고 `lambda`로는 다음과 같이 표현할 수 있다.

```
List<Apple> redApples = filter(inventory, (Apple apple) -> RED.equals(apple.getColor()));
List<Integer> evenNumbers = filter(numbers, (Integer i) -> i % 2 == 0);
```