애플리케이션 리팩토링

리팩토링

- **긁어 부스럼 만들지 말자. 돌아가기만 한다면 코드에 손대지 말라!!!**
- **리팩토링**
  - 코드의 기능은 변경하지 않으면서 내부의 구조와 코드를 개선하는 작업
  - 코드를 작성하고 난 후 설계를 향상시키는 작업
  - 기존의 코드를 가독성, 재활용, 체계적 구조 측면에서 개선하는 작업
  - 프로그램을 보다 쉽게 이해할 수 있고 소비용으로 수정할 수 있도록 사용자에게 제공
  - 하는 기능 동작의 변화 없이 내부 구조를 변경하는 소프트웨어 품질 기법을 말한다.
- **리팩토링의 장점**
  - 소프트웨어의 설계 개선
  - 낯선 코드를 쉽게 이해
  - 오류의 원인을 찾는것이 쉬워지며 수정 또한 빠르게 처리

리팩토링

- **리팩토링 시점**
  - 리팩토링은 별도의 시간을 내는 것이 아니라 틈틈이 지속 해야 하는 것으로, 기능개선 및 개발을 할 때 소스코드의 구조를 재정리하는 리팩토링을 한다.
    - **같은 작업을 3번 이상 할 때**
    - **기능을 추가할 때**
    - **버그를 수정할 때**
    - **코드를 검수 할 때**
  - 주로 하는 시점은 코드 리뷰를 하는 경우이다.
    1. **기능 추가 시**
       새로운 기능을 추가하기 전에 수행할 코드에 대한 이해를 높이기 위해 리팩토링을 수행한다. 이를 통해 기능 추가하는 작업은 훨씬 더 빠르고 원활하게 진행될 수 있다.
    2. **버그 수정 시**
       버그를 수정할 때 리팩토링을 수행하면 소스코드에 대한 이해가 깊어지며 버그뿐만 아니라 잠재적인 결함을 제거할 수 있는 기회가 될 수 있다.
    3. **지속 수행**
       별도의 시간을 내서 리팩토링을 수행하는 것이 아니라 틈틈이 지속해야 소스코드의 잠재적인 위험을 제거하게 되며 소프트웨어의 생명력을 지속할 수 있다.

리팩토링

- **깨끗한 코드(Clean Code) 기준**

1. **가독성이 좋은 코드**

   단순하고 직접적으로 잘 쓰인 문장처럼 읽히고 설계자의 의도가 보이는 코드로 추상화와 단순한 제어문으로 구성된 코드를 말한다.

2. **변경이 쉬운 코드**

   작성자가 아닌 사람도 읽기 쉽고 고치기 쉬운 코드로 이를 위해서는 의미 있은 이름이 부여되고 특정한 목적을 달성하는 방법은 하나만 제공한다. 또 의존성은 최소화하며 각 의존성에 대한 정의가 명확하다.

3. **중복 없는 코드**

   같은 작업을 여러 차례 반복하지 않는 코드로 같은 작업을 여러 차례 반복한다면 코드가 문제 해결을 제대로 하지 못한다는 증거이다.

4. **주의 깊게 짜인 코드**

   고치려고 살펴봐도 딱히 손댈 곳이 없는, 작성자가 이미 모든 상황을 고려한 소스코드를 말한다.

리팩토링

- **코드의 악취**
  1. 중복코드(Duplicated Code)
  2. 너무 긴 메소드(Long Method)
  3. 거대한 클래스(Large Class)
  4. 너무 많은 인수(Long Parameter List)
  5. 변경의 발산(Divergent Change)
  6. 변경의 분산(Shotgun Surgery)
  7. 속성, 조작의 부적절한 관계(Feature Envy)
  8. 기본 데이터형의 집착(Primitive Obsession)
  9. 추측성 일반화(Speculative Generality)
  10. 일시적 속성(Temporay Field)
  11. 메시지 연쇄(Message Chains)
  12. 중개자(Middle Man)
  13. 클래스의 인터페이스 불일치

리팩토링

- 내 코드를 설명할 수 없다?!!!
- 내 코드를 테스트 하기 위해서는 프로그램 전체를 실행 해야 한다?!!!
- 코드의 규칙과 표준을 지키지 않은 코드?!!!

리팩토링

- **리팩토링 주의 사항**
  - 자칫 잘못하면 코드의 복잡도가 높아져 코드를 이해하기 어렵게 된다.
  - 중복 코드를 방지하는 과정에서 미약하나마 성능 저하가 발생할 수 있다.
  - 또 지나치게 중복 방지에 집착하다 보면 제한된 시간과 노력을 낭비할 가능성이 있다.
  - 중복된 코드가 제거된 코드는 문서화하기 용이하고 코드 가동성이 향상될 가능성이 높다.

리팩토링 기준 수립하기

- 대상 애플리케이션의 이슈와 문제를 식별하고 대상을 선정하여 리팩토링 목표를 정의한다. 정의된 리팩토링 목표와 대상을 정의하여 리팩토링 계획을 수립한다.

  1. **대상 애플리케이션의 문제를 식별하고 정의한다.**
     대상 애플리케이션 소스코드의 가독성, 변경 용이성, 잠재적 결함 발생 가능성 등 문제 상황이 있는지 확인하고 리팩토링 대상 모듈을 식별한다. 샘플 소스코드를 대상으로 육안으로 소스코드의 문제점을 식별한다.

리팩토링 기준 수립하기

(1) 중복된 소스코드 식별

(가) 유형 1: 공백과 주석만 다르고 코드는 동일한 경우

(나) 유형 2: 공백과 주석이 다르고 코드는 동일하고 변수의 이름이 다른 경우

(다) 유형 3: 공백과 주석이 다르고 코드는 동일하며 변수 이름이 다르고, 코드의 몇 라인이 변경되었거나 추가되었을 경우

(라) 유형 4: 의미적으로 동일한 코드이지만 코드 문법이 다소 다른 경우

(2) 긴 메소드(Long Method) 식별

메소드 추출 기법을 적용하여 메소드를 식별하고 주석을 확인하여 긴 메소드를 식별한다. 또 조건문과 루프로 역시 메소드를 추출한다.

(3) 긴 파라미터 리스트(Long Parameter List) 식별

(4) 방대한 클래스 식별

기능이 많은 클래스를 식별하기 위해 인스턴스 변수가 많은 코드를 식별한다.

리팩토링 기준 수립하기

(5) 임시 필드(Temporary Field) 식별

객체 안에 인스턴스 변수가 특정 상황에서만 할당되는 경우, 인스턴스가 해당 변수를 사용하는 클래스를 생성하는데 이러한 필드가 포함된 소스코드를 식별한다.

(6) 주석(Comments) 분석

소스코드가 무슨 작업을 하는지 설명하기 위해 주석이 필요한 경우, 메소드가 이미 추출되어 있는데도 여전히 소스코드의 기능에 대한 주석이 필요한 경우, 시스템의 필요한 조건에 대한 어떤 규칙, 기준 등 을 설명할 필요가 있는 경우를 식별하여 리팩토링대상으로 선정한다.

소스코드 분석 도구

- 분석 도구 선정

분석 도구	설명	적용 대상
conQAT	개발 과정에서 Copy&Paste를 한 코드 분석	Java, C#, C
FindBugs	Java에서 100여 개의 잠재적인 에러 유형을 찾아주며, 이 잠재적인 에러는 scariest, scary, troubling, concern으로 구분하여 점수(rank)화해 주는 도구	Java
PMD	사용하지 않는 변수, 비어 있는 소스코드 블록, 불필요한 객체생성과 같은 결함을 유발할 수 있는 소스코드의 검사 도구	Java
CheckStyle	개발된 코드가 얼마나 코딩 룰을 잘 따르고 있는지 분석하는 도구	Java
Cppcheck	예외 처리, 클래스별 코드, 메모리 누수, 사용되지 않는 함수 및 변수 등 분석 도구	C
OCLint	static analysis 항목들을 모두 지원하여 분석하는 도구	C

소스코드 분석 도구

- 분석 도구 설치

소스코드 분석 도구

- **분석 도구 사용**

  (1) Violations outline 확인

  분석 결과에 대한 에러 메시지와

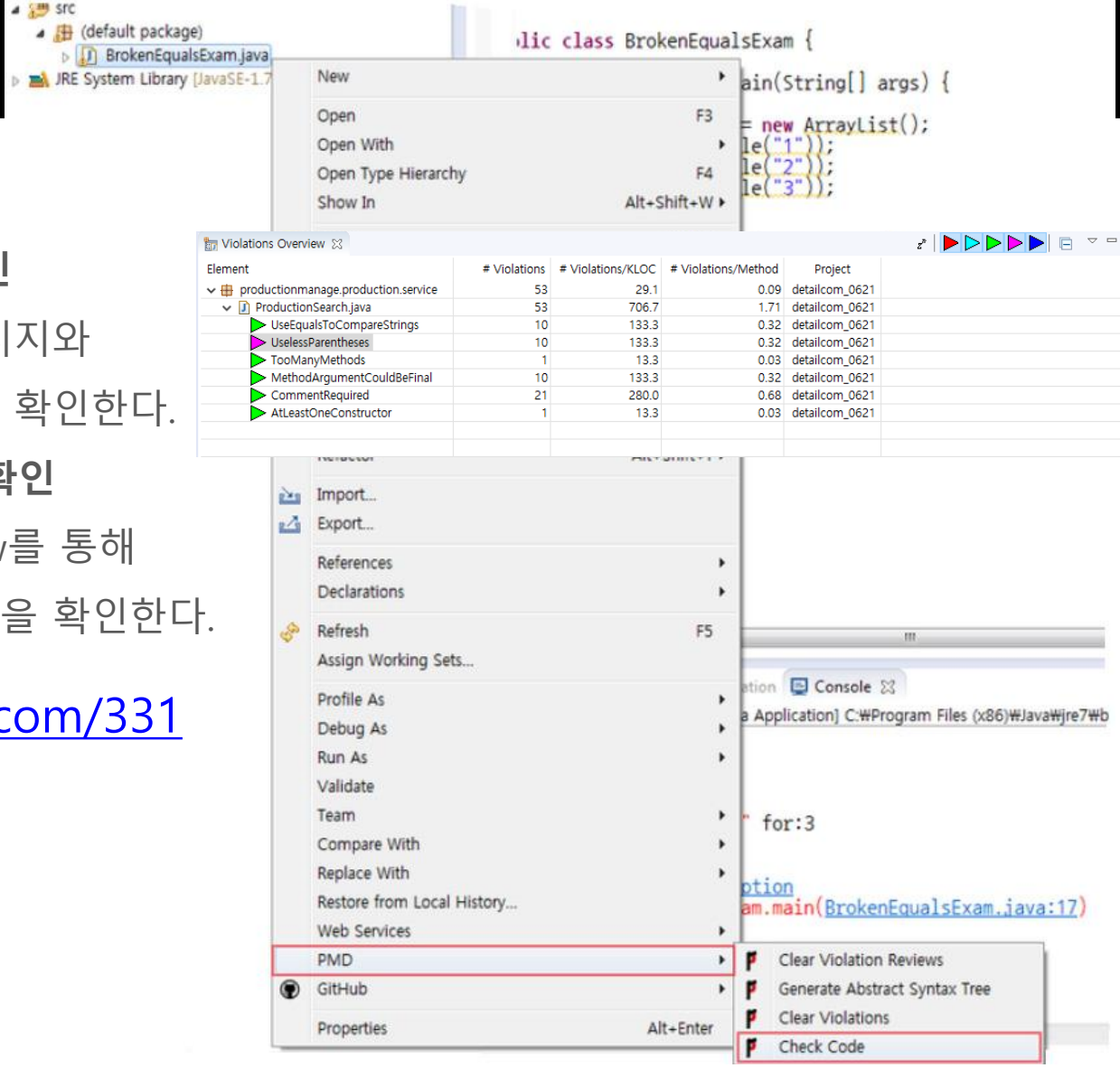
  에러가 발생한 해당 라인을 확인한다.

  (2) Violations Overview 확인

  분석 결과에 대한 Overview를 통해

  에러가 발생한 전체 통계량을 확인한다.

https://syaku.tistory.com/331



대상 애플리케이션의 리팩토링 세부 기법을 선정

- 소스코드 분석 결과를 파악하여 대상 애플리케이션의 잠재적 위험도와 개선을 위한 적절한 방법을 선정한다.

방법	설명	적용 대상
메소드 정리	그룹핑 할 수 있는 소스코드, 수식을 메소드로 변경	✓
객체 간의 기능 이동	메소드에 따른 위치 변경, 클래스 기능의 명확한 구분	✓
데이터 구성	객체 지향 캡슐화 기법을 적용하여 데이터 접근을 관리	데이터 접근 영역 부재
조건문 단순화	조건 논리를 단순하고 명확하게 작성	✓
메소드 호출 단순화	메소드 이름이 목적에 맞지 않는 경우 변경	✓
클래스 및 메소드 일반화	동일 메소드가 여러 클래스에 있으면 슈퍼 클래스로 이동	✓

소스코드분석

- **개발 표준**

  시스템 개발을 위한 환경 구성과 표준을 설명한 문서로 개발자 및 관리자에게 일관성 있는 개발이 가능하도록 정보를 제공하는 설명을 말한다.

목 차

소스코드분석

- 소스코드 개발 원칙

4.1. Coding 원칙

주석은 모든 코드에 상세히 기술하는 것을 원칙으로 한다.

소스 코드는 불가피한 내용을 제외하고 원칙적으로 중복을 금지한다.

소스 코드에서 사용하는 모든 단어는 자료사전의 내용을 기준으로 한다.

개발 및 테스트를 수행함에 있어 기능과 성능은 물론 보안에도 각별한 주의를 기울인다.

소스코드분석

- 코딩 스타일

4.2. Coding Style

4.2.1. Indent

Indent는 tab을 사용하지 않고, space만 사용한다.

Indent의 크기는 4로 한다.

4.2.2. space

한 줄에 하나의 statement만 기술한다.

Semicolon, comma, 예약어 뒤에는 space를 둔다.

unary operation은 space를 두지 않는다. (ex. i++;)

binary operation은 양쪽에 space를 둔다. (ex. i = i + 1;)

괄호 안에 괄호가 있는 경우에는 괄호 사이에 space를 두지 않는다.

4.2.3. brace

'{' 와 '}' 는 새로운 라인에 기술하며, 다른 내용과 함께 기술하지 않는다. (주석 제외)

'{' 는 기존의 '{' 와 비교해서 indent(4-space)를 준다.

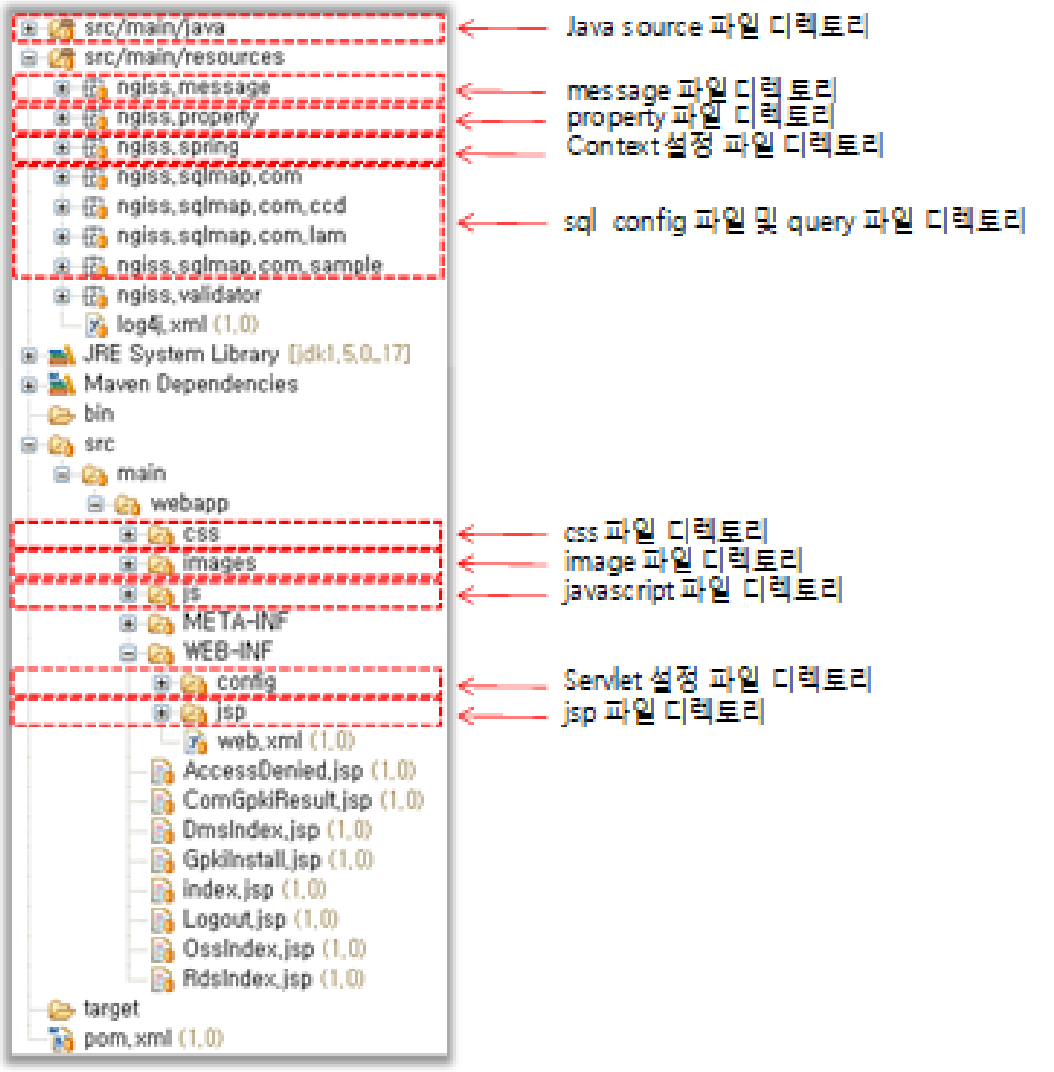
'}' 는 짝이 되는 '{' 과 동일하게 indent(4-space)를 준다.

brace에 주석을 기입하는 경우 '//' 주석을 사용한다.

소스코드분석

- 소스코드 디렉터리 구조

.5. 디렉토리 구조



리팩토링 적용

- **메소드 이동**

  메소드 이동은 기능이 정의된 객체보다 다른 객체의 메소드에서 더 많이 사용하고 있다면, 이 기능을 가장 많이 사용하고 있는 클래스에서 비슷한 형태를 가진 새로운 기능을 만들고 이 기능은 간단한 위임으로 바꾸거나 완전 제거하는 것을 의미한다.

```
class Account {
  double overdraftCharge() {
    if (_type.isPremium()) {
      double result = 10;
      if (_daysOverdrawn > 7 ) result += (_daysOverdrawn - 7) * 0.85;
      return result;
    }
    else return _daysOverdrawn * 1.75;
  }
  double bankCharge() {
    double result = 4.5;
    if (_daysOverdrawn > 0) result += overdraftCharge();
    return result;
  }
  private AccountType _type;
  private int _daysOverdrawn;
}
```

```
class AccountType {
  double overdraftCharge(int daysOverdrawn) {
    if (isPremium()) {
      double result = 10;
      if (daysOverdrawn > 7) result += (daysOverdrawn - 7) * 0.85;
      return result;
    }
    else return daysOverdrawn * 1.75;
  }
}
```

```
class Account {
  double bankCharge() {
    double result = 4.5;
    if(_dayOverdrawn > 0) result += _type.overdraftCharge(_daysOverdrawn);
    return result;
  }
}
```

리팩토링 적용

- **메소드 추출**

  메소드가 너무 길거나 코드에 주석을 달아야만 의도를 이해할 수 있을 때, 그 소스코드를 분리하여 별도의 메소드를 만든다.

```
void printOwing(double amount) {
printBanner();

//상세 정보 표시
System.out.println( "userID:" + _userID );
System.out.println( "amount:" + amount ); }
```

```
void printOwing(double amount)
{
  printBanner();
  printDetails(amount);
}
void printDetails(double amount)
{
  System.out.println( "userID:" + _userID );
  System.out.println( "amount:" + amount );
}
```

리팩토링 적용

- **임시 Inline 정리**

  수식의 결괏값을 가지는 임시 변수가 있고 그 임시 변수가 다른 리팩토링을 하는 데 방해가 된다면, 해당 임시 변수를 참조하는 코드를 모두 원래 수식으로 변경하는 기법이다.

```
double basePrice = anOrder.basePrice();
    return (basePrice > 1000);


    return (anOrder.basePrice() > 1000);
```

리팩토링 적용

- **임시 변수 참조 변경**

  결과값을 저장하기 위해 임시 변수를 호출하여 사용하고 있다면, 수식을 추출하여 메소드로 만들고, 임시 변수를 참조하는 곳을 찾아 모두 메소드 호출로 변경한다. 새로운 메소드는 다른 메소드에서 사용 가능하다.

```
double basePrice = _quantity * _itemPrice;
    if( basePrice > 1000 )
      return basePrice * 0.95;
    else
      return basePrice * 0.98;
```

```
if( basePrice() > 1000 )
      return basePrice * 0.95;
    else
      return basePrice * 0.98;
    ...

    double basePrice()
    {
      return _quantity * _itemPrice;
    }
```

리팩토링 적용

- **조건문 개선 - 중첩된 조건문 처리 하기**

  결과값을 저장하기 위해 임시 변수를 호출하여 사용하고 있다면, 수식을 추출하여 메소드로 만들고, 임시 변수를 참조하는 곳을 찾아 모두 메소드 호출로 변경한다. 새로운 메소드는 다른 메소드에서 사용 가능하다.

```
boolean con1 = true;
boolean con2 = true;
boolean con3 = true;
boolean con4 = true;

if (con1 == true) {
  if (con2 == true) {
    if (con3 == true) {
      if (con4 == true) {
        System.out.println("Complete");
      } else {
        System.out.println("con4
break");
      }
    } else {
      System.out.println("con3 break");
    }
  } else {
    System.out.println("con2 break");
  }
} else {
  System.out.println("con1 break");
}
```

```
boolean con1 = true;
boolean con2 = true;
boolean con3 = true;
boolean con4 = true;

String result = "Complete";

if (con1 == true) {
  result = "con1 break";
}
if (con2 == true) {
  result = "con2 break";
}
if (con3 == true) {
  result = "con3 break";
}
if (con4 == true) {
  result = "con4 break";
}

System.out.println(result);
```

리팩토링 적용

- 여러 형태의 처리를 하나의 서비스에서 처리할 때에는 커멘트 패턴을 이용

```
if(구글로그인){
    // 로그인 처리
}else if(인스타로그인){
    // 로그인 처리
}else if(페이스북로그인){
    // 로그인 처리
}else{
    // 로그인 처리
}
```

문제점

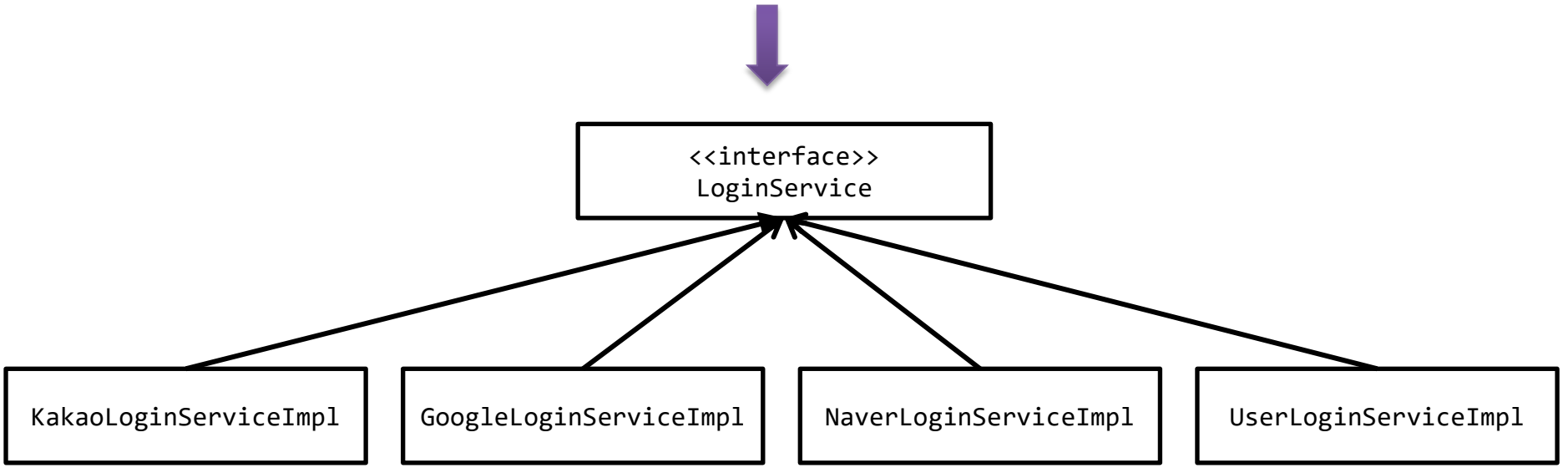
새로운 추가가 발생할 때 마다 소스를 추가/변경해야 한다.

코드의 양이 커진다.

복잡도 증가한다.

가독성이 떨어진다.

부분 테스트 불가능하다.



리팩토링 적용

- **Use try-with-resources or close this "BufferedInputStream" in a "finally" clause**

  DB Connection, InputStream, file, URLConnection 등 Closeable 인터페이스를 구현 또는 상속받은 클래스는 반드시 finally에서 해당 자원을 close 해줘야만 하지만 누락하는 경우가 있습니다.

  이때 try~catch~finally 를 이용해서 반드시 close 처리를 해주어야 합니다.

리팩토링 적용

- **타입선언은 인터페이스로 하자**

  List, Set, Map 사용 시 자주 실수하는 내용 중 하나입니다.

```
ArrayList<String> list = new
ArrayList<>();

HashMap<String, String> map = new
HashMap<>();

HashSet<String> set = new HashSet<>();



public void test(HashMap<String,
String>){

    ...

}
```

```
List<String> list = new ArrayList<>();

Map<String, String> map = new HashMap<>();

Set<String> set = new HashSet<>();



public void test(Map<String, String>){

    ...

}
```

리팩토링 적용

- Map 전체 내용을 읽을 때 Key, Value를 모두 사용한다면 "entrySet"을 사용하라

```
public void doSomethingWithMap(Map<String,Object> map) {
  for (String key : map.keySet()) {  // Noncompliant; for each key the value is retrieved
    Object value = map.get(key);
    // ...
  }
}
```

```
public void doSomethingWithMap(Map<String,Object> map) {
  for (Map.Entry<String,Object> entry : map.entrySet()) {
    String key = entry.getKey();
    Object value = entry.getValue();
    // ...
  }
}
```

리팩토링 적용

- 동일 처리를 하는 catch문 합치기

```
public boolean checkUser(UserParam userParam) {
            try {
                        if(getUserInfo(userParam) != null) return true;
            }catch(DataNotFoundException e) {
                        return false;
            }catch(UserNotFoundException e) {
                        return false;
            }
            return false;
}
```

```
public boolean checkUser(UserParam userParam) {
            try {
                        if(getUserInfo(userParam) != null) return true;
            }catch(DataNotFoundException | UserNotFoundException  e) {
                        return false;
            }
            return false;
}
```