**CHAPTER1**

**리팩토링 첫 단계**

아무리 체계적이 리팩토링 공식을 이용해 버그가 생길 수 있는 대부분의 원인을 방지하더라도, 인간인 이상 실수할 수 있기 때문에 신뢰도 높은 테스트 작성은 필수다.

번거로운데 굳이 변수명을 변경할 필요성이 있나?

🡺당연히 있다. 좋은 코드는 그것이 무슨 기능을 하는지 분명히 드러나야 하는데, 코드의 기능을 분명히 드러내는 열쇠가 바로 직관적인 변수명이다.

**컴퓨터가 인식 가능한 코드는 바보라도 작성할 수 있지만, 인간이 이해할 수 있는 코드는 실력 있는 프로그래머만 작성할 수 있다.**

상태 패턴을 적용하려면 세 가지 리팩토링 기법을 사용해야 한다.

1) 분류 부호를 상태/전략 패턴으로 전환 기법을 실시해서 분류 부호의 기능을 상태 패턴안으로 옮겨야 한다.

2) 메서드 이동 기법을 실시해서 switch 문을 State(interface) 클래스 안으로 옮겨야 한다.

3) 조건문을 재정의로 전환 기법을 실시해서 switch 문을 없애야 한다.



**CHAPTER2 리팩토링 개론**

**리팩토링(명사) 겉으로 드러나는 기능은 그대로 둔 채, 알아보기 쉽고 수정하기 간편하게 소프트웨어 내부를 수정하는 작업**

**리팩토링(동사) 리팩토링 기법을 연달아 적용해서 겉으로 드러나는 기능은 그대로 둔 채 소프트웨어 구조를 변경한다.**

**모자 두개**

기능을 추가할 땐 코드를 수정하지 말고 기능만 추가해야 한다.

리팩토링할때는 코드를 추가하지 말고 코드 구조 개선만 해야 한다.

**리팩토링은 왜 하나**

-소프트웨어 설계가 개선되니까

-소프트웨어를 이해하기가 더 쉬워지니까

-버그를 찾기가 쉬워지니까

-프로그래밍 속도가 빨라지니까

**리팩토링은 어떨 때 필요한가**

-같은 작업의 삼진 아웃 때

-기능을 추가 할 때

-버그를 수정할 때

-코드를 검수할 때

**프로그램은 4가지 상황일 때 수정하기 힘들어진다.**

-코드를 알아보기 힘들 때

-중복된 로직이 들어 있을 때

-추가 기능을 넣어야 해서 실행 중인 코드를 변경해야 할 때

-조건문 구조가 복잡할 때

🡺그러므로 프로그램은 코드를 알아보기 쉽고, 모든 로직이 한 곳에 있으며, 기존 기능을 건드릴 필요 없이 조건문 구조가 최대한 간결하게끔 작성해야 한다.

**CHAPTER3 코드의 구린내**

-중복 코드

-장황한 메서드

-방대한 클래스

-과다한 매개변수

-수정의 산발

-기능의 산재 (수정의 산발 반대)

-잘못된 소속

-데이터 뭉치

-강박적 기본 타입 사용

-Switch 문

-평행 상속 계층

-직무유기 클래스

-막연한 범용 코드

-임시 필드

-메시지 체인

-과잉 중개 메서드

-지나친 관여

-인터페이스가 다른 대용 클래스

-미흡한 라이브러리 클래스

-데이터 클래스

-방치된 상속물

-불필요한 주석

책 완독 후, CHATPER3장 다시 보기

**CHAPTER4 테스트 작성**

리팩토링을 실시하기 위한 필수 전제조건은 반드시 견고한 테스트를 해야 한다는 것이다.

**테스트를 자주 실행하자, 적어도 하루에 한 번은 테스트를 실시하고, 컴파일할 때마다 테스트를 국소화하자.**

**단위 테스트와 기능 테스트(연동 테스트)**

단위 테스트는 매우 국소적이어서, 각 테스트 클래스는 하나의 패키지 안에서만 효력이 있다.

기능 테스트의 목적은 소프트웨어 전반이 제대로 돌아가는지 확인하는 것이다.

**CHAPTER6 메서드 정리**

-메서드 추출 : 코드 뭉치를 별도의 메서드로 빼낸다.

-메서드 삽입 : 제 역할을 못하는 메서드는 호출부에 삽입 / 방식을 변경할 때

-메서드 호출로 전환 : 임시변수를 호출로 전환해서 임시변수 제거

-임시변수 분리 : 임시변수가 여러부분 사용될때

-메서드 객체로 전환 : 임시변수가 너무 얽혀 있어서 메서드 호출로 전환할 수 없을 때

-값 대입 제거 : 매개변수로 대입하는 값이 있을 때

-알고리즘 전환 : 알고리즘을 더 명료하게 개선할 수 있음을 알게 될때

**1) 메서드 추출**

어떤 코드를 그룹으로 묶어도 되겠다고 판단될 땐, 그 코드를 빼내어 목적을 잘 나타내는 직관적 이름의 메서드로 만들자.

*Before*

private String \_name;  
  
public void printOwing(double amount) {  
 printBanner();  
 System.*out*.println("name : " + \_name);  
 System.*out*.println("amount : " + amount);  
}  
  
private void printBanner() {  
 System.*out*.println("banner");  
}

*after*

private String \_name;  
  
public void printOwing(double amount) {  
 printBanner();  
}  
  
private void printBanner() {  
 System.*out*.println("banner");  
}  
  
private void printDetails(double amount) {  
 System.*out*.println("name : " + \_name);  
 System.*out*.println("amount : " + amount);  
}

**2) 메서드 내용 직접 삽입**

메서드 기능이 너무 단순해서 메서드명만 봐도 너무 뻔할 땐

그 메서드의 기능을 호출하는 메서드에 넣어버리고 그 메서드는 삭제하자.

*before*

private int \_numberOfLateDeliveries = 10;  
  
private int getRating() {  
 return (moreThanFiveLateDeliveries()) ? 2 : 1;  
}  
  
private boolean moreThanFiveLateDeliveries() {  
 return \_numberOfLateDeliveries > 5;  
}

*after*

private int \_numberOfLateDeliveries = 10;  
  
private int getRating() {  
 return (\_numberOfLateDeliveries > 5) ? 2 : 1;  
}

**3) 임시변수 내용 직접 삽입**

간단한 수식을 대입받는 임시변수로 인해 **다른 리팩토링 기법 적용이 힘들 땐**

그 임시변수를 참조하는 부분을 전부 수식으로 치환하자.

*before*

private boolean moreThanPrice(int price) {  
 double basePrice = price;  
 return (basePrice > 1000);  
}

*after*

private boolean moreThanPrice(int price) {  
 return (price > 1000);  
}

**4) 임시변수를 메서드 호출로 전환**

수식의 결과를 저장하는 임시변수가 있을 땐

그 수식을 빼내어 메서드로 만든 후, 임시변수 참조 부분을 전부 수식으로 교체하자.

새로 만든 메서드는 다른 메서드에서도 호출 가능하다.

*before*

private double \_quantitiy = 5;  
private double \_itemPrice = 1000;  
  
private double getPrice() {  
 double basePrice = \_quantitiy \* \_itemPrice;  
 if (basePrice > 1000) {  
 return basePrice \* 0.95;  
 } else {  
 return basePrice \* 0.98;  
 }  
}

*after*

private double \_quantitiy = 5;  
private double \_itemPrice = 1000;  
  
private double getPrice() {  
 if (basePrice() > 1000) {  
 return basePrice() \* 0.95;  
 } else {  
 return basePrice() \* 0.98;  
 }  
}  
  
private double basePrice() {  
 return \_quantitiy \* \_itemPrice;  
}

**5) 직관적인 임시변수 사용**

사용된 수식이 복잡할 땐

수식의 결과나 수식의 일부분을 용도에 부합하는 직관적 이름의 임시변수에 대입하자.

*before*

private void isFilter() {  
 final String platform = "mac";  
 final String browser = "ie";  
 final int resize = 0;  
  
 if ((platform.toUpperCase().indexOf("MAC") > -1) &&  
 (browser.toUpperCase().indexOf("IE") > -1) &&  
 wasInitialized() && resize > 0) {  
 // 기능 코드  
 }  
  
}  
  
private boolean wasInitialized() {  
 return true;  
}

*after*

private void isFilter() {  
 final String platform = "mac";  
 final String browser = "ie";  
 final int resize = 1;  
  
 final boolean isMacOS = platform.toUpperCase().indexOf("MAC") > -1;  
 final boolean isIEBrowser = browser.toUpperCase().indexOf("IE") > -1;  
 final boolean wasResized = resize > 0;  
  
 if (isMacOS && isIEBrowser && wasInitialized() && wasResized) {  
 // 기능 코드  
 }  
  
}  
  
private boolean wasInitialized() {  
 return true;  
}

**6) 임시변수 분리**

루프 변수나 값 누적용 임시변수가 아닌 임시변수에 여러 번 값이 대입될 땐

각 대입마다 다른 임시변수를 사용하자.

*beafore*

private double \_height;  
private double \_width;  
  
private void tempMethod() {  
 double temp = 2 \* (\_height + \_width);  
 System.*out*.println(temp);  
 temp = \_height \* \_width;  
 System.*out*.println(temp);  
}

*after*

private double \_height;  
private double \_width;  
  
private void tempMethod() {  
 final double perimeter = 2 \* (\_height + \_width);  
 System.*out*.println(perimeter);  
 final double area = \_height \* \_width;  
 System.*out*.println(area);  
}

**7) 매개변수로의 값 대입 제거**

매개변수로 값을 대입하는 코드가 있을 땐 매개변수 대신 임시변수를 사용하게 수정하자.

*before*

private int discount(int inputVal, int quantitiy, int yearToDate) {  
 if (inputVal > 50) {  
 inputVal -= 2;  
 }  
  
 return inputVal;  
}

*after*

private int discount(int inputVal, int quantitiy, int yearToDate) {  
 if (inputVal > 50) {  
 inputVal -= 2;  
 }  
  
 return inputVal;  
}

**8) 메서드를 메서드 객체로 전환**

지역변수 때문에 메서드 추출을 적용할 수 없는 긴 메서드가 있을 때

그 메서드 자체를 객체로 전환해서 모든 지역변수를 객체의 필드로 만들자.

그런다음 그 메서드를 객체 안의 여러 메서드로 쪼개면 된다.

*before*

private double price() {  
 double primaryBasePrice = 1;  
 double secondaryBasePrice = 2;  
 return primaryBasePrice + secondaryBasePrice;  
}

*after*

public class Basic {  
 private double price() {  
 return new PriceCalculator(this, 1, 2).compute();  
 }  
 public double delta() {  
 return 1;  
 }  
}  
  
class PriceCalculator {  
 private double \_primaryBasePrice;  
 private double \_secondaryBasePrice;  
 private Basic \_basic;  
  
 public PriceCalculator(Basic basic, double primaryBasePrice, double secondaryBasePrice) {  
 // 여기서 처리할거 처리  
 this.\_primaryBasePrice = primaryBasePrice;  
 this.\_secondaryBasePrice = secondaryBasePrice;  
 this.\_basic = basic;  
 }  
 public double compute() {  
 return \_primaryBasePrice + \_secondaryBasePrice + \_tertiaryBasePrice + \_basic.delta();  
 }  
}

**9) 알고리즘 전환**

알고리즘을 더 분명한 것으로 교체해야 할 땐, 해당 메서드의 내용을 새 알고리즘으로 바꾸자.

*before*

private String foundPerson(String[] people) {  
 for (int i = 0; i < people.length; i++) {  
 if (people[i].equals("Don")) {  
 return "Don";  
 }  
  
 if (people[i].equals("John")) {  
 return "John";  
 }  
  
 if (people[i].equals("Kent")) {  
 return "Kent";  
 }  
 }  
 return "";  
}

*after*

private String foundPerson(String[] people) {  
 List<String> candidates = Arrays.*asList*(new String[] { "Done", "John", "Kent" });  
 for (int i = 0; i < candidates.size(); i++) {  
 if (candidates.contains(people[i])) {  
 return people[i];  
 }  
 }  
 return "";  
}

**CHAPTER7 객체 간의 기능 이동**

-메서드 이동 (필드 이동 후 진행)

-필드 이동

-클래스 추출 : 클래스가 방대해질 때

-클래스 직접 삽입 : 클래스 추출로 클래스에 기능이 너무 적어질 때

-대리 객체 은폐 : 다른 클래스(대리 클래스)가 사용중일 때, 그 대리 클래스가 사용 중이라는 사실을 감추는 것이 좋다.

-과잉 중개 메서드 제거 : 대리 클래스 은폐 시, 그 클래스를 소유한 클래스의 인터페이스가 계속 변경될 때도 있는데, 이럴 때

-외래 클래스에 메서드 추가(특수케이스) : 클래스의 원본 코드에 접근할 수 없는 상황에서 이 수정 불가능한 클래스로 기능을 이동해야 할 때 (메서드가 한두개 뿐일 때)

-국소적 상속확장 클래스 사용(특수케이스) : (위와 동일하지만 메서드가 세 개 이상일 때)

**1) 메서드 이동**

메서드가 자신이 속한 클래스보다 다른 클래스의 기능을 더 많이 이용할 땐

그 메서드가 제일 많이 이용하는 클래스 안에서 비슷한 내용의 새 메서드를 작성하자.

기존 메서드는 간단한 대리 메서드로 전환하든지 아예 삭제하자.

**2) 필드 이동**

어떤 필드가 자신이 속한 클래스보다 다른 클래스에서 더 많이 사용될 때는

대상 클래스 안에 새 필드를 선언하고 그 필드 참조 부분을 전부 새 필드 참조로 수정하자.

**3) 클래스 추출**

두 클래스가 처리해야 할 기능이 하나의 클래스에 들어 있을 땐

새 클래스를 만들고 기존 클래스의 관련 필드와 메서드를 새 클래스로 옮기자.

**4) 클래스 내용 직접 삽입**

클래스에 기능이 너무 적을 땐

그 클래스의 모든 기능을 다른 클래스로 합쳐 넣고 원래 클레스는 삭제하자.

**5) 대리 객체 은폐**

클라이언트가 객체의 대리 클래스를 호출할 땐

대리 클래스를 감추는 메서드를 서버에 작성하자.

**6) 과잉 중개 메서드 제거**

클래스에 자잘한 위임이 너무 많을 땐

대리 객체를 클라이언트가 직접 호출하게 하자.

**7) 외래 클래스에 메서드 추가**

사용 중인 서버 클래스에 메서드를 추가해야 하는데 그 클래스를 수정할 수 없을 땐

클라이언트 클래스 안에 서버 클래스의 인스턴스를 첫 번째 인자로 받는 메서드를 작성하자.

서버의 구현 로직을 클라이언트에서 만드는 것인데, 외래 메서드는 임시방편에 불가하다.

**8) 국소적 상속확장 클래스 사용**

사용 중인 서버 클래스에 여러 개의 메서드를 추가해야 하는데 클래스를 수정할 수 없을 땐

새 클래스를 작성하고 그 안에 필요한 여러 개의 메서드를 작성하자.

이 상속확장 클래스를 원본 클래스의 하위클래스나 래퍼 클래스로 만들자.

**CHAPTER8 데이터 체계화**

-자체 캡슐화 : 읽기/쓰기 메서드를 사용해야 할 때

-데이터 값을 객체로 전환 : 불명확한 데이터를 뚜렷한 객체로 바꿀 때

-참조로 전환 : 이런 객체가 프로그램의 여러 부분에 사용될 인스턴스일 때

-배열을 객체로 전환 : 배열이 데이터 구조 역할을 할 때

-마법 숫자를 기호 상수로 전환 : 마법 숫자를 제거할 때

-클래스의 단방향 연결을 양방향으로 전환

-양방향 연결을 단방향으로 전환 : 양방향 연결이 필요 없어졌을 때, 불필요한 복잡함을 해소

-관측 데이터 복제 : GUI 관련 클래스가 해서는 안 될 비즈니스 로직을 수행할 때

-필드 캡슐화 : public 데이터가 질서 없이 여기저기 노출되어 있을 때

-컬렉션 캡슐화 : public 데이터가 컬렉션일 때

-레코드를 데이터 클래스로 전환 : 하나의 레코드가 통째로 공개되어 있을 때

-분류 부호를 클래스로 전환 : 분류 부호가 정보만 제공하고 클래스의 기능을 변경하진 않을 때

-분류 부호를 하위클래스로 전환 : 클래스의 기능이 분류 부호에 영향을 미칠 때

-분류 부호를 상태/전략 패턴으로 전환

**1) 필드 자체 캡슐화**

필드에 직접 접근하던 중 그 필드로의 결합에 문제가 생길 땐

그 필드용 읽기/쓰기 메서드를 작성해서 두 메서드를 통해서만 필드에 접근하게 만들자

*before*

private int \_low, \_high;  
  
boolean includes(int arg) {  
 return arg >= \_low && arg <= \_high;  
}

*after*

private int \_low, \_high;  
boolean includes(int arg) {  
 return arg >= getLow() && arg <= getHigh();  
}  
  
int getLow() {  
 return \_low;  
}  
  
int getHigh() {  
 return \_high;  
}

**2) 데이터 값을 객체로 전환**

데이터 항목에 데이터나 기능을 더 추가해야 할 때는 데이터 항목을 객체로 만들자.

**3) 값을 참조로 전환**

클래스에 같은 인스턴스가 많이 들어 있어서 이것들을 하나의 객체로 바꿔야 할 땐

그 객체를 참조 객체로 전환하자.

**4) 참조를 값으로 전환**

참조 객체가 작고 수정할 수 없고 관리하기 힘들 땐 그 참조 객체를 값 객체로 만들자

(equals와 hashCode 재정의)

**5) 배열을 객체로 전환**

배열을 구성하는 특정 원소가 별의별 의미를 지닐 땐 (의미가 다를 때)

그 배열을 각 원소마다 필드가 하나씩 든 객체로 전환하자.

*before*

private void testMethod() {  
 String[] row = new String[3];  
 row[0] = "Liverpool";  
 row[1] = "15";  
}

*after*

private void testMethod() {  
 Performance row = new Performance();  
 row.setName("Liverpool");  
 row.setWins("15");  
}

**6) 관측 데이터 복제**

도메인 데이터는 GUI 컨트롤 안에서만 사용 가능한데, 도메인 메서드가 그 데이터에 접근해야 할 땐, 그 데이터를 도메인 객체로 복사하고, 양측의 데이터를 동기화하는 관측 인터페이스 observer를 작성하자.

**7) 클래스의 단방향 연결을 양방향으로 전환**

두 클래스가 서로의 기능을 사용해야 하는데 한 방향으로만 연결되어 있을 땐

역 포인터를 추가하고 두 클래스를 모두 업데이트할 수 있게 접근 한정자를 수정하자.

**8) 클래스의 양방향 연결을 단방향으로 전환**

두 클래스가 양방향으로 연결되어 있는데 한 클래스가 다른 클래스의 기능을 더 이상 사용하지 않게 됐을 땐 불필요한 방향의 연결을 끊자

**9) 마법 숫자를 기호 상수로 전환**

특수 의미를 지닌 리터럴 숫자가 있을 땐

의미를 살린 이름의 상수를 작성한 후 리터럴 숫자를 그 상수로 교체하자.

*before*

double potentialEnergy(double mass, double height) {  
 return mass \* 9.81 \* height;  
}

*after*

static final double *GRAVITATIONAL\_CONSTANT* = 9.81;  
  
double potentialEnergy(double mass, double height) {  
 return mass \* *GRAVITATIONAL\_CONSTANT* \* height;  
}

**10) 필드 캡슐화**

public 필드가 있을 땐

그 필드를 private로 만들고 필드용 읽기 메서드와 쓰기 메서드를 작성하자.

*before*

public class basic {  
 public String \_name;  
}

*after*

private String \_name;  
  
public void setName(String name) {  
 this.\_name = name;  
}  
  
public String getName() {  
 return \_name;  
}

**11) 컬렉션 캡슐화**

메서드가 컬렉션을 반환할 땐

그 메서드가 읽기전용 뷰를 반환하게 수정하고 추가 메서드와 삭제 메서드를 작성하자.

**12) 레코드를 데이터 클래스로 전환**

전통적인 프로그래밍 환경에서 레코드 구조를 이용한 인터페이스를 제공해야 할 땐

레코드 구조를 저장할 덤 데이터 객체를 작성하자.

-덤 데이터 객체 : 데이터가 거의 들어 있지 않은 객체로, 기능 추가 없이 데이터에 public 속성이나 읽기/쓰기 메서드로 접근할 수 있다.

**13) 분류 부호를 클래스로 전환**

기능에 영향을 미치는 숫자형 분류 부호가 든 클래스가 있을 땐

그 숫자를 새 클래스로 바꾸자.

**14) 분류 부호를 하위클래스로 전환 (is–a)**

클래스 기능에 영향을 주는 변경불가 분류 부호가 있을 땐

분류 부호를 하위 클래스로 만들자

**-적용할 수 없는 경우**

1) 분류 부호의 값이 객체 생성 후 변할 때

2) 다른 이유로 분류 부호를 이미 하위클래스로 만들었을 때

3) 분류 부호를 상태/전략 패턴으로 전환을 실시해야 할 때

**15) 분류 부호를 상태/전략 패턴으로 전환 (has–a)**

분류 부호가 클래스의 기능에 영향을 주지만 하위클래스로 전환할 수 없을 땐

그 분류 부호를 상태 객체로 만들자.

**16) 하위클래스를 필드로 전환**

여러 하위클래스가 상수 데이터를 반환하는 메서드만 다를 땐

각 하위클래스의 메서드를 상위클래스 필드로 전환하고 하위클래스는 전부 삭제하자.

**CHAPTER9 조건문 간결화**

-조건문 쪼개기 : 조건문을 여러 개로 나눌 때

-조건식 통합 : 여러 조건 검사가 있는데 결과가 모두 같을 때

-조건문의 공통 실행 코드 빼내기 : 조건문 안의 중복 코드를 제거할 때

-여러 겹의 조건문을 감시 절로 전환 : 특수한 case 조건문을 명확히 할 때

-제어 플래그 제거 : 복잡한 제어 플래그를 제거

-switch-case 문에도 전부 조건무을 재정의로 전환

-Null 검사를 널 객체에 위임

※ 객체지향 프로그램은 절차 프로그램에 비해 조건에 따른 동작이 대체로 적은데,

이것은 조건에 따른 동작 대부분이 재정의를 통해 처리되기 때문이다.

재정의 방식이 더 좋은 이유는 호출 코드가 조건문 원리를 알 필요가 없어서 **조건문을 확장하기가 더 간편하기 때문이다**.

**1) 조건문 쪼개기**

복잡한 조건문(if-then-else)이 있을 땐, If, then, else 부분을 각 메서드로 빼자.

(나는 보통 isNotEmpty, isEmpty 같은 유틸을 많이 사용함)

*before*

if (date.before(SUMMER\_START) || date.after(SUMMER\_END)) {  
 charge = quantity \* \_winterRate + \_winterServiceCharge;  
} else {  
 charge = quantity \* \_summerRate;  
}

*after*

if (notSummer(date)) {  
 charge = winterCharge(quantity);  
} else {  
 charge = summerCharge(quantity);  
}

**2) 중복 조건식 통합**

여러 조건 검사의 결과가 같을 땐, 하나의 조건문으로 합친 후 메서드로 빼내자.

*before*

double disabilityAmount() {  
 if (\_seniority < 2) {  
 return 0;  
 }  
 if (\_monthsDisabled > 12) {  
 return 0;  
 }  
 if (\_isPartTime) {  
 return 0;  
 }  
 return -1;  
}

*after*

double disabilityAmount() {  
 if (isNotElgibleForDisability()) {  
 return 0;  
 }  
  
 return -1;  
}

**3) 조건문의 공통 실행 코드 빼내기**

조건문의 모든 절에 같은 실행 코드가 있을 땐, 같은 부분을 조건문 밖으로 빼자

*before*

if (*isSpecialDeal*()) {  
 total = price \* 0.95;  
 *send*();  
} else {  
 total = price \* 0.98;  
 *send*();  
}

*after*

if (*isSpecialDeal*()) {  
 total = price \* 0.95;  
} else {  
 total = price \* 0.98;  
}  
*send*();

**4) 제어 플래그 제거**

논리 연산식의 제어 플래그 역할을 하는 변수가 있을 땐 그 변수를 break문이나 return 문으로 바꾸자

**5) 여러 겹의 조건문을 감시절로 전환**

메서드에 조건문이 있어서 정상적인 실행 경로를 파악하기 힘들 땐

모든 특수한 경우에 감시 절을 사용하자.

*before*

double getPayAmount() {  
 double result;  
 boolean \_isDead = false;  
 boolean \_isSeparated = true;  
 boolean \_isRetired = true;  
  
 if (\_isDead) {  
 result = deadAmount();  
 } else {  
  
 if (\_isSeparated) {  
 result = separatedAmount();  
 } else {  
 if (\_isRetired) {  
 result = retiredAmount();  
 } else {  
 result = normalPayAmount();  
 }  
 }  
 }  
 return result;  
}

*after*

double getPayAmount() {  
 double result;  
 boolean \_isDead = false;  
 boolean \_isSeparated = true;  
 boolean \_isRetired = true;  
  
 if (\_isDead) {  
 return deadAmount();  
 }  
  
 if (\_isSeparated) {  
 return separatedAmount();  
 }  
  
 if (\_isRetired) {  
 return retiredAmount();  
 }  
  
 return normalPayAmount();  
}

**6) 조건문을 재정의로 전환**

객체 타입에 따른 다른 기능을 실행하는 조건문이 잇을 땐

조건문의 각 절을 하위클래스의 재정의 메서드 안으로 옮기고, 원본 메서드는 abstract 타입으로 수정하자.

**7) Null 검사를 널 객체에 위임 (Java8 이상부터는 Optional 처리)**

Null 값을 검사하는 코드가 계속 나올 땐 null 값을 널 객체로 만들자.

if(customer ==null) {  
 plan = BillingPlan.basic();  
} else {  
 plan = customer.getPlan();  
}

※ 좋긴 한데 NullClass를 별도로 만들어야 되는 번거로움이 있다.

**8) 어설션 넣기**

일부 코드가 프로그램의 어떤 상태를 전제할 땐, 어설션을 넣어서 그 전제를 확실하게 코드로 작성하자.

*before*

double getExpenseLimit() {  
 Memeber \_primaryProject = new Memeber();  
 final double NULL\_EXPENSE = 0;  
 double \_expenseLimit = 10;  
  
 return (\_expenseLimit != NULL\_EXPENSE) ? \_expenseLimit : \_primaryProject.getMemberExpenseLimit();  
}

*after*

double getExpenseLimit() {  
 Memeber \_primaryProject = new Memeber();  
 final double NULL\_EXPENSE = 0;  
 double \_expenseLimit = 10;  
  
 Assert.*assertTrue*(\_expenseLimit != NULL\_EXPENSE || \_primaryProject != null);  
 return (\_expenseLimit != NULL\_EXPENSE) ? \_expenseLimit : \_primaryProject.getMemberExpenseLimit();  
}

**CHAPTER10 메서드 호출 단순화**

-메서드명 변경 : 남들도 쉽게 이해할 수 있게 전달하고자 할 때

-매개변수 추가 / 매개변수 제거

-객체를 통째로 전달 : 객체에서 여러 값을 전달할 때

-객체로 전환 : 위의 하나의 객체가 존재하지 않을때, 객체를 생성

-메서드로 추출 : 메서드가 이미 접근한 객체로부터 데이터를 가져올 수 있을 때

-매개변수를 메서드로 전환 : 조건별 동작 판단에 사용되는 매개변수들이 있을 때

-상태 변경 메서드와 값 반환 메서드를 분리 : 둘이 합쳐져 있을 때 (병렬성과 연관)

-메서드 은폐 / 쓰기 메서드 제거 : 리팩토링을 실시할 때 뭔가를 잠시만 보이게 한 후 다시 감춰야 할 때

-생성자를 팩토리 메서드로 전환 : 생성해야 할 객체의 클래스를 알 필요가 없을 때

-하향 타입 변환을 캡슐화 : 클래스 사용 부분에서 하향 타입 변환이 수행되지 않게 할 때

-예외 통지로 교체 : 새 예외 처리 기능을 사용하려 할 때

-예외 처리를 테스트로 교체 : 예외 처리가 정답이 아닐 경우, 우선 테스트부터 실시

**1) 메서드명 변경**

메서드명을 봐도 기능을 알 수 없을 땐 메서드명을 직관적인 이름으로 바꾸자

**※ 이 기술을 발전시키는 것이야말로 진성으로 노련한 프로그래머가 되는 열쇠다**

**2) 매개변수 추가**

메서드가 자신을 호출한 부분의 정보를 더 많이 알아야 할 땐 객체에 그 정보를 전달할 수 있는 매개변수를 추가하자.

**3) 매개변수 제거**

메서드가 어떤 매개변수를 더 이상 사용하지 않을 땐 그 매개변수를 삭제하자.

**4) 상태 변경 메서드와 값 반환 메서드를 분리**

값 반환 기능과 객체 상태 변경 기능이 한 메서드에 들어 있을 땐 질의 메서드와 변경 메서드로 분리하자.

**5) 메서드를 매개변수로 전환**

여러 메서드가 기능은 비슷하고 안에 든 값만 다를 땐 서로 다른 값을 하나의 매개변수로 전달받는 메서드를 하나 작성하자.

**6) 매개변수를 메서드로 전환**

매개변수로 전달된 값에 따라 메서드가 다른 코드를 실행할 땐

그 매개변수로 전달될 수 있는 모든 값에 대응하는 메서드를 각각 작성하자

*before*

void setValue(String name, int value) {  
 if (name.equals("height")) {  
 \_height = value;  
 return;  
 }  
  
 if (name.equals("width")) {  
 \_width = value;  
 return;  
 }  
}

*after*

void setHeight(int arg) {  
 \_height = arg;  
}  
  
void setWidth(int arg) {  
 \_width = arg;  
}

**7) 객체를 통째로 전달**

객체에서 가져온 여러 값을 메서드 호출에서 매개변수로 전달할 객체를 통째로 전달하게 수정하자.

*before*

int low = daysTempRange().getLow();  
int high = daysTempRangE().getHight();  
withinPlan = plan.withinRangE(low, high);

*after*

withinPlan = plan.withinRange(daysTempRangE());

**8) 매개변수 세트를 메서드로 전환**

객체가 A메서드를 호출해서 그 결과를 B 메서드에 매개변수로 전달하는데,

결과를 매개변수로 받는 B메서드도 직접 A 메서드를 호출할 수 있을 땐

매개변수를 없애고 A메서드를 B메서드가 호출하게 하자.

*before*

int basePrice = \_quantity \* \_itemPrice;  
discountLevel = getDiscountLevel();  
double finalPrice = discountedPrice(basePrice, discountLevel);

*after*

int basePrice = \_quantity \* \_itemPrice;  
double finalPrice = discountedPrice(basePrice);

**9) 매개변수 세트를 객체로 전환**

여러 개의 매개변수가 항상 붙어 다닐 땐, 그 매개변수들을 객체로 바꾸자

※ 나 같은 경우 빌더 패턴을 사용한다.

**10) 쓰기 메서드 제거**

생성할 때 지정한 필드 값이 절대로 변경되지 말아야 할 땐 (final)

그 필드를 설정하는 모든 쓰기 메서드를 삭제하자.

**11) 메서드 은폐**

메서드가 다른 클래스에 사용되지 않을 땐 그 메서드의 반환 타입을 private로 만들자

**12) 생성자를 팩토리 메서드로 전환**

객체를 생성할 땐 단순한 생성만 수행하게 해야할 땐 생성자 팩토리 메서드로 교체하자.

*before*

public class Employee {  
 private int \_type;  
  
 Employee(int type) {  
 \_type = type;  
 }  
}

*after*

public class Employee {  
 private int \_type;  
  
 Employee(int type) {  
 \_type = type;  
 }  
  
 static Employee create(int type) {  
 return new Employee(type);  
 }  
}

**13) 하향 타입 변환을 캡슐화**

메서드가 반환하는 객체를 호출 부분에서 하향 타입 변환해야 할 땐

하향 타입 변환 기능을 메서드 안으로 옮기자 .

**14) 에러 부호를 예외 통지로 교체**

메서드가 에러를 나타내는 특수한 부호를 반환할 땐

그 부호 반환 코드를 예외 통지 코드로 바꾸자

*before*

public int withdraw(int amount) {  
 if (amount > \_balance) {  
 return -1;  
 } else {  
 \_balance -= amount;  
 return 0;  
 }  
}

*after*

public void withdraw(int amount) throws Exception {  
 if (amount > \_balance) {  
 throw new Exception();  
 } else {  
 \_balance -= amount;  
 }  
}

**15) 예외 처리를 테스트로 교체**

호출 부분에 사전 검사 코드를 넣으면 될 상황인데 예외 통지를 사용했을 땐

호출 부분이 사전 검사를 실시하게 수정하자.

*before*

double getValueForPeriod(int peroidNumber) {  
 try {  
 return \_values[peroidNumber];  
 } catch (ArrayIndexOutOfBoundsException e) {  
 return 0;  
 }  
}

*after*

double getValueForPeriod(int peroidNumber) {  
 if (peroidNumber >= \_values.length) {  
 return 0;  
 } else {  
 return \_values[peroidNumber];  
 }  
}

**CHAPTER11 일반화 처리**

-필드 상향, 메서드 상향 : 상위클래스로 옮길 때

-메서드 하향, 필드 하향 : 하위클래스로 옮길 때

-생성자 내용 상향 : 상위 클래스로 생성자를 옮기고 싶을 때

-팩토리 메서드로 전환 : 하위 클래스로 생성자를 옮기고 싶을 때

-템플릿 메서드 형성 : 여러 메서드가 서로 틀만 비슷하고 세부 구현이 다를 때

-하위클래스 추출, 상위클래스 추출, 인터페이스 추출 : 여러 지점의 기능을 새로운 요소로 만들어 내고 싶을 때

-인터페이스 추출 : 타입 시스템에서 눈에 띄도록 기능의 작은 부분에 표시해야 할 때

-계층 병합 : 계층구조에 불필요한 클래스가 있을 때 계층 병합을 적용해서 없앤다.

-상속을 위임으로 전환

-위임을 상속으로 전환

**1) 필드 상향**

두 하위클래스에 같은 필드가 들어 있을 땐 필드를 상위클래스로 옮기자.

**※일반화 : 상위클래스로 옮기는 작업**

**2) 메서드 상향**

기능이 같은 메서드가 여러 하위클래스에 들어 있을 땐 그 메서드를 상위클래스로 옮기자.

**3) 생성자 내용 상향**

하위클래스마다 거의 비슷한 내용의 생성자가 있을 땐

상위 클래스에 생성자를 작성하고, 그 생성자를 하위클래스의 메서드에서 호출하자.

*before*

class Manager extends Employee {  
 String \_name;  
 String \_id;  
 int \_grade;  
  
 public Manager(String name, String id, int grade) {  
 \_name = name;  
 \_id = id;  
 \_grade = grade;  
 }  
}

*after*

public class Employee {  
 public Employee(String name, String id) {  
 \_name = name;  
 \_id = id;  
 }  
  
}  
  
class Manager extends Employee {  
 public Manager(String name, String id, int grade) {  
 super(name, id);  
 \_grade = grade;  
 }  
}

**4) 메서드 하향**

상위클래스에 있는 기능을 일부 하위클래스만 사용할 땐

그 기능을 관련된 하위클래스 안으로 옮기자.

**5) 필드 하향**

일부 하위클래스만이 사용하는 필드가 있을 땐 그 필드를 사용하는 하위클래스로 옮기자.

**6) 하위클래스 추출**

일부 인스턴스에만 사용되는 기능이 든 클래스가 있을 땐 그 기능 부분을 전담하는 하위클래스를 작성하자.

**7) 상위클래스 추출**

기능이 비슷한 두 클래스가 있을 땐 상위클래스를 작성하고 공통된 기능들을 그 상위클래스로 옮기자.

**8) 인터페이스 추출**

클래스 인터페이스의 같은 부분을 여러 클라이언트가 사용하거나, 두 클래스에 인터페이스의 일부분이 공통으로 들어 있을 땐 공통 부분을 인터페이스로 빼내자

**9) 계층 병합**

상위클래스와 하위클래스가 거의 다르지 않을 땐 둘을 합치자.

**10) 템플릿 메서드 형성**

하위클래스 안의 두 메서드가 거의 비슷한 단계들을 같은 순서로 수행할 땐

그 단계들을 시그너처가 같은 두 개의 메서드로 만들어서 두 원본 메서드를 같게 만든 후,

두 메서드를 상위클래스로 옮기자.

**11) 상속을 위임으로 전환**

하위클래스가 상위클래스 인터페이스의 일부만 사용할 때나 데이터를 상속받지 않게 해야 할 땐

상위클래스에 필드를 작성하고, 모든 메서드가 그 상위클래스에 위임하게 수정한 후 하위클래스를 없애자.

**12) 위임을 상속으로 전환**

위임을 이용 중인데 인터페이스 전반에 간단한 위임으로 도배하게 될 땐 위임 클래스를 대리 객체의 하위클래스로 만들자.

**CHAPTER12 복합 리팩토링**

-상속 구조 정리 : 온갖 변경사항을 뒤죽박죽 합쳐 놓은 것 같은 지저분한 상속 구조를 정리할 때

-절차 코드를 객체로 전환 : 절차 코드 기능의 고질적인 문제를 해결하는 리팩토링

-도메인 로직을 표현과 분리 : 비즈니스 로직을 인터페이스 코드에서 분리

-계층구조 추출 : 매우 복잡한 클래스를 여러 하위클래스로 전환해서 단순화

**1) 상속 구조 정리**

하나의 상속 계층이 두 작업을 동시에 수행할 땐 상속 계층을 하나 더 만들어서 위임을 통해 다른 계층을 호출하자.

**2) 절차 코드를 객체로 전환**

코드가 절차식으로 작성되어 있을 땐 데이터 레코드를 객체로 바꾸고, 기능을 쪼개서 각각의 객체로 옮기자.

**3) 도메인 로직을 표현과 분리**

도메인 로직이 들어 있는 GUI 클래스가 있을 땐 도메인 로직을 별도의 도메인 클래스로 떼어내자.

**4) 계층구조 추출**

한 클래스에 기능이 너무 많고 일부분에라도 조건문이 많을 땐 각 조건에 해당하는 하위클래스를 작성해서 계층구조를 만들자.

**CHAPTER13 리팩토링, 재사용, 현실성**

**개발자가 리팩토링을 꺼리는 이유**

-리팩토링 방법을 몰라서

-리팩토링의 장점은 오랜 시간이 흘러야 가시화될 테고, 그때가 되면 프로젝트 팀원도 아닐 텐데 공연히 리팩토링에 힘 뺄 필요 없어서

-코드를 리팩토링 하는 일은 추가적인 부담인데다, 월급은 새 기능을 추가하라고 주는거지 리팩토링하라고 주는건 아니라서

-리팩토링하다가 괜히 멀쩡한 프로그램을 망가뜨릴까 걱정되서

**안전하게 리팩토링하기**

-자신의 코딩 능력에 신념을 갖자

-자신이 놓친 에러는 컴파일러가 잡아서 처리하리란 믿음을 갖자

-컴파일러마저 놓친 에러는 테스트 스위트가 잡아 처리하리란 신뢰를 갖자

-테스트 스위트마저 놓친 에러는 코드 검수 단계에서 발견하리란 확신을 갖자

**CHAPTER15 조각 맞추기**

**리팩토링은 학습 가능한 기술이다.**

-습관적으로 목표를 정하자

-확신이 없으면 중단하자

-되돌아 가자

-작업은 둘이서 하자