**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.