Test-Driven Development

Chapter 30. 디자인패턴
Chapter 31. 리팩토링
Chapter 32. TDD 마스터하기

30

디자인패턴

커맨드

계산 작업에 대한 호출을 메시지가 아닌 객체로 표현한다.

복잡한 형태의 계산 작업에 대한 호출이 필요할 땐,
계산 작업에 대한 객체를 생성하여 이를 호출하라

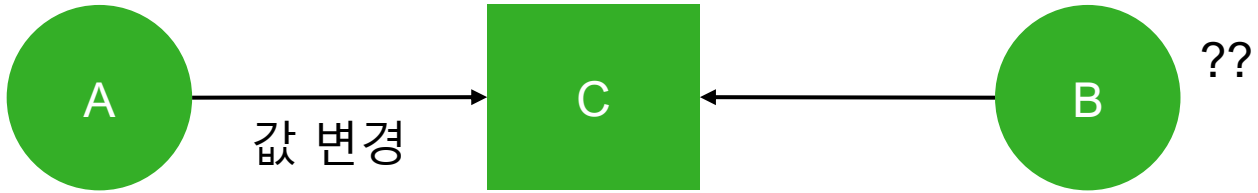
값 객체

객체가 생성된 이후 그 값이 절대로 변하지 않게 하여 별칭 문제가 발생하지 않게 한다.

널리 공유해야 하지만 동일성(identity)은 중요하지 않을 때는
객체가 생성될 때 상태를 설정한 후에 이 상태가 절대 변할 수 없도록 한다.

+ 이 객체에 대한 수행되는 연산은 언제나 새로운 객체를 반환하게 한다.

별칭문제



해결 방법

1. 현재 의존하는 객체에 대한 참조를 결코 외부에 알리지 않는 방법
   - 객체에 대한 복사본을 제공
   - 수행 시간, 메모리 공간 측면에서 비싼 해결책, 공유 객체 사용 불가능
2. 옵저버 패턴 사용
   - 의존하는 객체에 자기를 등록해 놓고, 객체의 상태가 변하면 통지를 하는 방법
   - 제어 흐름을 이해하기 어렵게 만들 수 있고, 의존성을 위한 로직이 지저분함
3. **객체를 덜 객체답게 취급하기**

별칭문제

객체를 변화가 불가능하게 만들자.

연산 결과로는 새로운 객체를 반환하게 만든다.

```
class Money {
        int value;

        public Money(int value) {
            this.value = value;
        }

        public Money add(Money money) {
            return new Money(this.value + money.value);
        }
}
```

값 객체 예시

A = 3

B = A

B = 5

A = ???

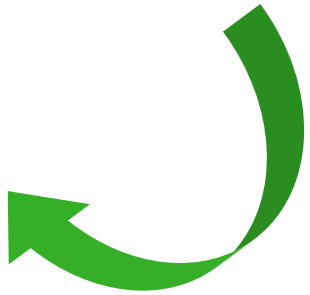
모든 값 객체는 동등성을 구현해야 한다.

동일성(identity)와 동등성(equality)은 서로 다르다.

널 객체

객체의 특별한 상황을 표현하는 새로운 객체를 만든다.

```
public boolean setReadOnly() {
        SecurityManager guard = System.getSecurityManager();
        guard.canWrite(path);
        return fileSystem.setReadOnly(this);
}

// LaxSecurity
public void canWrite(String path){}

// SecurityManager
public static SecurityManager getSecurityManager() {
        return security == null ? new LaxSecurity() : security;
}
```

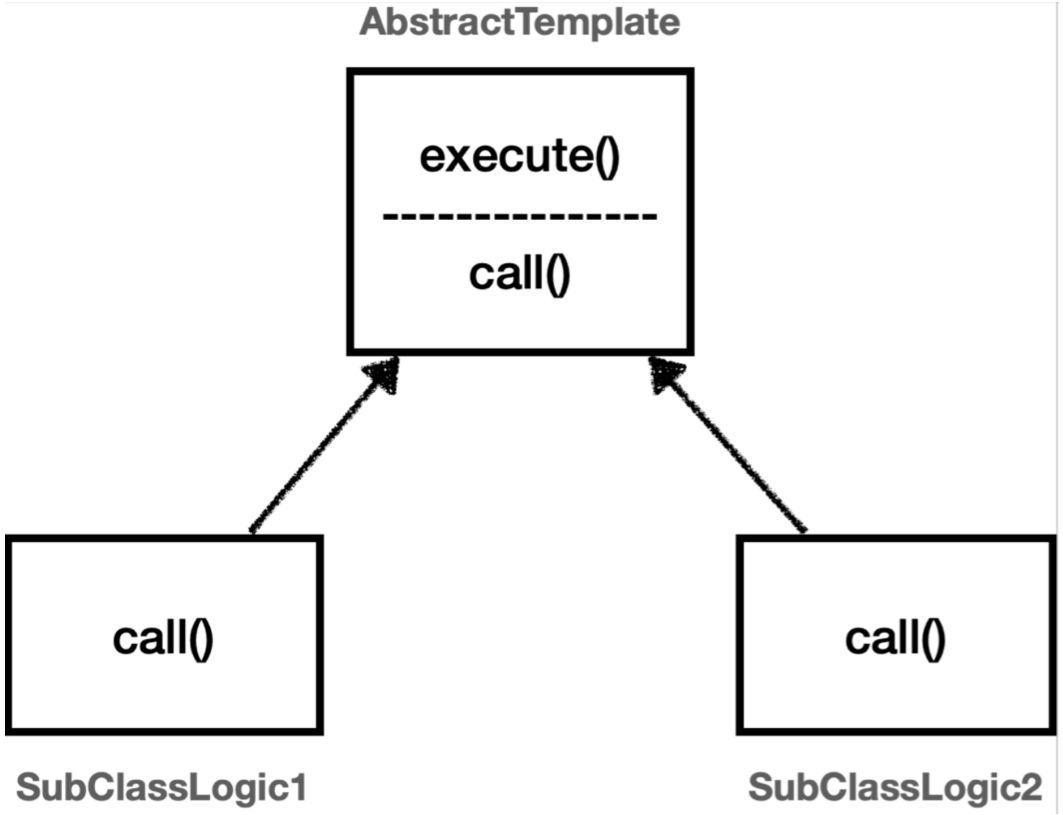


```
public boolean setReadOnly() {
        SecurityManager guard = System.getSecurityManager();
        if (guard != null) {
                guard.canWrite(path);
        }
        return fileSystem.setReadOnly(this);
}
```

Template Method Pattern

변하지 않는 것은 추상클래스의 메서드로 선언

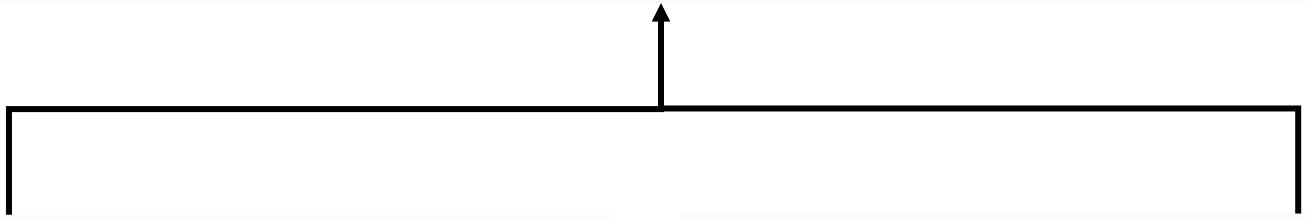
변하는 부분은 추상 메서드로 선언하여 자식 클래스가 오버라이딩



https://redcarrot1.github.io/posts/design_pattern_template_mthod_pattern/

Template Method Pattern

```java
public abstract class AbstractTemplate {

    public void execute() {
        System.out.println("템플릿 시작");
        //변해야 하는 로직 시작
        logic();
        //변해야 하는 로직 종료
        System.out.println("템플릿 종료");
    }

    protected abstract void logic(); //변경 가능성이 있는 부분은 추상 메소드로 선언한다.
}
```



```java
public class SubClassLogic1 extends AbstractTemplate {
    @Override
    protected void logic() {
        System.out.println("변해야 하는 메서드는 이렇게 오버라이딩으로 사용1.");
    }
}
```

```java
public class SubClassLogic2 extends AbstractTemplate {
    @Override
    protected void logic() {
        System.out.println("변해야 하는 메서드는 이렇게 오버라이딩으로 사용2.");
    }
}
```

Template Method Pattern

```java
public class templateMethod1 extends AbstractTemplate {
    public static void main(String[] args) {
        AbstractTemplate template1 = new SubClassLogic1();
        template1.execute();

        System.out.println();

        AbstractTemplate template2 = new SubClassLogic2();
        template2.execute();
    }
}
```

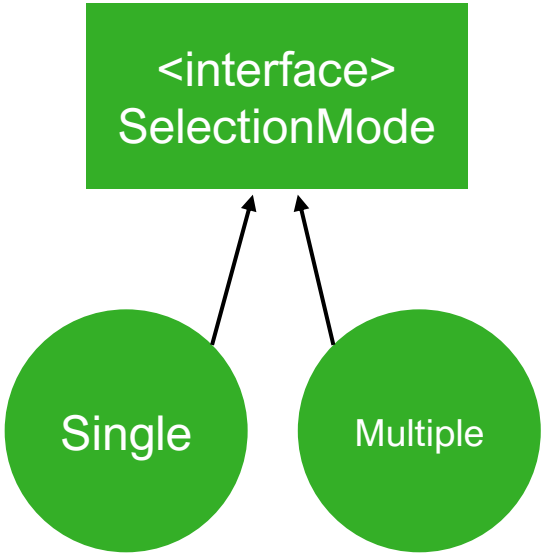
```
//출력
템플릿 시작
변해야 하는 메서드는 이렇게 오버라이딩으로 사용1.
템플릿 종료

템플릿 시작
변해야 하는 메서드는 이렇게 오버라이딩으로 사용2.
템플릿 종료
```

플러거블 객체

둘 이상의 구현을 객체가 호출함으로써 다향성을 표현한다.

```
Figure selected;
public void mouseDown() {
    selected = FindFigure();
    if (selected != null)
        select(selected);
}

public void mouseMove() {
    if (selected != null)
        move(selected);
    else
        moveSelectionRectangle();
}

public void mouseUp() {
    if (selected == null)
        selectAll();
}
```

```
SelectionMode mode;
public void mouseDown() {
    selected = FindFigure();
    if (selected != null)
        mode = SingleSelection(selected);
    else
        mode = MultipleSelection();
}

public void mouseMove() {
    mode.mouseMove();
}

public void mouseUp() {
        mode.mouseUp();
}
```



플러거블 셀렉터

객체별로 서로 다른 메서드가 동적으로 호출되게 함으로써
필요 없는 하위 클래스 생성을 피한다.

1. 상속 이용하기 : 상속은 작은 변이를 다루기에는 무거운 기법이다.

```
abstract class Report {
    abstract void print();
}
class HTMLReport extends Report {
    void print() {...}
}
class XMLReport extends Report {
    void print() {...}
}
```

플러거블 셀렉터

2. switch문 이용하기 : OCP 위반

```
abstract class Report {
    String printMessage;
    Report(String printMessage) {
            this.printMessage = printMessage;
    }
    void print() {
            switch (printMessage) {
            case "printHTML" :
                printHTML();
                break;
            case "printXML" :
                printXML();
                break;
          }
    }
    void printHTML(){...}
    void printXML(){...}
}
```

3. 플러거블 셀럭터 해법

```
void print() {
    Method runMethod = getClass()
                        .getMethod(printMessage, null);
    runMethod.invoke(this, new Class[0]);
}
```

팩토리 메서드

생성자 대신 메서드를 호출함으로써 객체를 생성한다.

장점: 다른 클래스의 인스턴스를 반환할 수 있는 유연함

```
// Money.class
static Dollar dollar(int amount) {
    return new Dollar(amount);
}
```

사칭 사기꾼

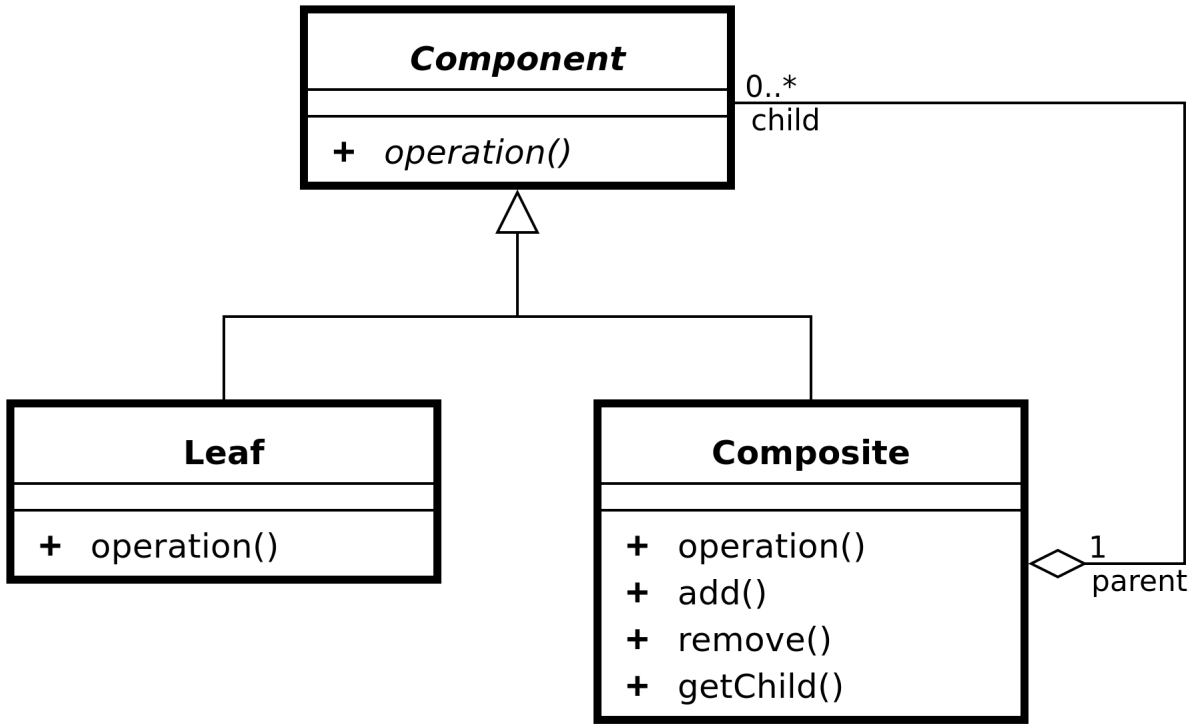
현존하는 프로토콜을 갖는 다른 구현을 추가하여 시스템에 변이를 도입한다.(다형성)

```
testRectangle() {
    Drawing d = new Drawing();
    d.addFigure(new RectangleFigure(0, 10, 50, 100));
    RecordingMedium brush = new RecordingMedium();
    d.display(brush);
    assertEquals("rectangle 0 10 50 100\n", brush.log());
}
```

```
testOval() {
    Drawing d = new Drawing();
    d.addFigure(new OvalFigure(0, 10, 50, 100));
    RecordingMedium brush = new RecordingMedium();
    d.display(brush);
    assertEquals("oval 0 10 50 100\n", brush.log());
}
```

컴포지트

하나의 객체로 여러 객체의 행위 조합을 표현한다.



Component

```
public interface Shape {
    public void draw(String paintColor);
}
```

컴포지트

Leaf

```
public class Triangle implements Shape {
    public void draw(String paintColor) {
        System.out.println("삼각형 색칠 : " + paintColor);
    }
}
```

```
public class Square implements Shape {
    public void draw(String paintColor) {
        System.out.println("사각형 색칠 : " + paintColor);
    }
}
```

```
public class Circle implements Shape {
    public void draw(String paintColor) {
        System.out.println("원 색칠 : " + paintColor);
    }
}
```

컴포지트

Composite

```
public class Drawing implements Shape {
    private List<Shape> shapes = new ArrayList<>();

    @Override
    public void draw(String paintColor) {
        for(Shape shape : shapes) {
            shape.draw(paintColor);
        }
    }

    public void add(Shape s) {
        this.shapes.add(s);
    ]

    public void remove(Shape s) {
        this.shapes.remove(s);
    }

    public void clear() {
        System.out.println("모든 도형을 제거합니다.");
        this.shapes.clear();
    }
}
```

컴포지트

```
public class CompositePattern {
    public static void main(String args[]) {
        Shape triangle = new Triangle();
        Shape circle = new Circle();
        Shape square = new Square();

        Drawing drawing = new Drawing();
        drawing.add(triangle);
        drawing.add(circle);
        drawing.add(square);

        drawing.draw("빨간색");

        List<Shape> shapes = new ArrayList<>();
        shapes.add(drawing);
        shapes.add(new Triangle());
        shapes.add(new Circle());

        for(Shape shape : shapes) {
            shape.draw("초록색");
        }
    }
}
```

```
삼각형 색칠 : 빨간색
동그라미 색칠 : 빨간색
사각형 색칠 : 빨간색

삼각형 색칠 : 초록색
동그라미 색칠 : 초록색
사각형 색칠 : 초록색
삼각형 색칠 : 초록색
동그라미 색칠 : 초록색
```

수집 매개 변수

여러 다른 객체에서 계산한 결과를 모으기 위해 매개 변수를 여러 곳으로 전달한다.

```
class Drink {
    String name;
    int alcohol;
    int stock;

    @Override
    public String toString() {
        return "Drink{" +
                "name='" + name + '\'' +
                ", alcohol=" + alcohol +
                ", stock=" + stock +
                '}';
    }
}
```

```
class Factory {
    void calculateAlcohol(Drink drink) {
        drink.alcohol = 3;
    }
}
```

```
class Company {
    void namingDrink(Drink drink, String name) {
        drink.name = name;
    }
}
```

```
class Shop {
    void buy(Drink drink, int number) {
        drink.stock = number;
    }
}
```

수집 매개 변수

```
public static void main(String args[]) {
    Drink drink = new Drink();
    Factory factory = new Factory();
    Company company = new Company();
    Shop shop = new Shop();

    factory.calculateAlcohol(drink);
    company.namingDrink(drink, "일품 진로");
    shop.buy(drink, 4);
    System.out.println(drink);
}
```

```
Drink{name='일품 진로', alcohol=3, stock=4}
```

싱글톤

전역 변수를 제공하지 않는 언어에서는 전역 변수를 사용하지 마라

31

리팩토링

차이점 일치시키기

비슷해 보이는 두 코드 조각을 합치려면

1. 두 코드가 단계적으로 닮아가게끔 수정한다.
2. 이 둘이 완전히 동일해지면 둘을 합친다.

대표적인 예시

반복문 구조가 비슷하다

조건문에 의해 나눠지는 두 분기의 코드가 비슷하다

두 클래스가 비슷하다.

변화 격리하기

객체나 메서드의 일부만 바꾸려면, 일단 바꿔야 할 부분을 격리한다.

변화를 격리하기 위해 사용할 수 있는 방법

1. 메서드 추출하기(가장 일반적)
2. 객체 추출하기
3. 메서드 객체(Method Object)

메서드 객체(Method Object)

```
int complexCalculation(int one, int two, int three, Obejct _mos) {
        int result1 = one + two + three + _mos.otherMethod();
        int result2 = one * two * three * _mos.otherMethod();
        return result1 + result2;
    }
```

```
class ComplexCalculator {
    private int one;
    private int two;
    private int three;
    private int result1;
    private int result2;
    private Obejct _mos;

    public ComplexCalculator(int one, int two, int three, Obejct _mos) {
        this.one = one;
        this.two = two;
        this.three = three;
        this._mos = _mos;
    }

    int calculate() {
        result1 = one + two + three + _mos.otherMethod();
        result2 = (one * two * three) + _mos.otherMethod();
        return result1 + result2;
    }
}
```

데이터 이주시키기

표현 양식을 변경하려면 일시적으로 데이터를 중복시킨다.

새로운 포맷의 인스턴스 변수를 추가한다.
기존 포맷의 인스턴스 변수를 세팅하는 모든 부분에서 새로운 인스턴스 변수도 세팅하게 만든다.
기존 변수를 사용하는 모든 곳에서 새 변수를 사용하게 만든다.
기존 포맷을 제거한다.
새 포맷에 맞게 외부 인터페이스를 변경한다.

때로는 외부 API를 먼저 변화시키기를 원할 때도 있다.

새 포맷으로 인자를 하나 추가한다.
새 포맷 인자에서 이전 포맷의 내부적 표현양식으로 번역한다.
이전 포맷 인자를 삭제한다.
이전 포맷을 사용하는 것들을 새 포맷으로 바꾼다.
이전 포맷을 지운다.

메서드 추출하기

길고 복잡한 메서드를 읽기 쉽게 만들기 위해 일부분을 별도의 메서드로 분리해내고 이를 호출하게 한다.

1. 기존의 메서드에서 **별도의 메서드로 분리할 수 있을 만한 부분을 찾아 낸다**. 반복문 내부의 코드나 반복문 전체, 혹은 조건문의 가지들이 일반적인 후보다.
2. 추출할 영역의 **외부에서 선언된 임시 변수**에 대해 할당하는 문장이 없는지 확인한다.
3. 추출할 코드를 **복사**해서 새 코드에 **붙인다**.
4. 원래 메서드에 있던 각각의 임시 변수와 매개 변수 중 새 메서드에서도 쓰이는 게 있으면, 이들을 **새 메서드의 매개 변수로 추가**한다.
5. 기존의 메서드에서 **새 메서드를 호출**한다.

메서드를 작은 조각으로 나누는 것은 때때로 그 정도가 지나칠 수도 있다.

더 나아갈 방법이 보이지 않을 때엔, 새로운 방식으로 메서드를 추출하기 위해 일단 모든 코드를 한 자리에 모아놓고 메서드 인라인 리팩토링을 한다.

메서드 인라인

너무 꼬여있거나 산재한 제어 흐름을 단순화시키기 위해 메서드를 호출하는 부분을 호출될 메서드의 본문으로 교체한다.

1. 메서드를 복사한다.
2. 메서드 호출하는 부분을 지우고 복사한 코드를 붙인다.
3. 모든 형식(formal) 매개 변수를 실제(actual) 매개 변수로 변경한다. 예를 들어 만약 reader.getNext()같은 매개 변수를 전달했다면, 이를 지역 변수에 할당해주어야 할 것이다.

인터페이스 추출하기

자바 오퍼레이션에 대한 두 번째 구현을 추가하려면 공통되는 오퍼레이션을 담은 인터페이스를 만들면 된다.

1. 인터페이스를 선언한다. 때론 새로 추가될 인터페이스의 이름으로 기존 클래스의 이름을 사용해야 하는 경우가 있는데, 그런 경우라면 인터페이스를 추가하기 전에 기존 클래스의 이름을 변경해주어야 한다.
2. 기존 클래스가 인터페이스를 구현하도록 만든다.
3. 필요한 메서드를 인터페이스에 추가한다. 필요하다면 클래스에 존재하는 메서드들의 가시성을 높여준다.
4. 가능한 모든 곳의 타입 선언부에서 클래스 이름 대신 인터페이스 이름을 사용하게 바꾼다.

이때 인터페이스 이름을 찾는 것은 어느 경우에 딱 맞는 메타포를 찾기 위해 고생하기도 한다.

메서드 옮기기

메서드를 원래 있어야 할 장소로 옮기려면 어울리는 클래스에 메서드를 추가하고, 호출하게 하라

1. 메서드를 복사한다.
2. 원하는 클래스에 붙이고 이름을 적절히 지어준 다음 컴파일한다.
3. 원래 객체가 메서드 내부에서 참조된다면, 원래 객체를 새 메서드의 매개 변수로 추가한다. 원래 객체의 필드들이 참조되고 있다면 그것들도 매개 변수로 추가한다. 만약 원래 객체의 필드들이 갱신된다면 포기해야 한다.
4. 원래 메서드의 본체를 지우고, 그곳에 새 메서드를 호출하는 코드를 넣는다.

메서드 옮기기

메서드 옮기기의 세 가지 속성

1. 코드에 대한 깊은 이해가 없더라도 언제 이 리팩토링이 필요한지 쉽게 알아낼 수 있다. 다른 객체에 대한 **두 개 이상의 메시지**를 보내는 코드를 볼 때마다 메서드 옮기기를 해주면 된다.
2. 리팩토링 절차가 빠르고 안전하다.
3. 리팩토링 결과가 종종 새로운 사실을 알려준다.

   "이렇게 되면 Rectangle이 아무 계산도 하지 않게 되잖아? 아하, 알았다. Rectangle에서 area 메서드를 추가하는 방향이 더 좋겠군"

메서드 옮기기

rectangle 객체에 4개의 메시지가 보내지고 있다.

```
// Shape.class
int width = rectangle.right() - rectangle.left();
int height = rectangle.bottom() - rectangle.top();
int area = witdh * height;
```

```
// Rectangle.class
public int area {
    int width = this.right() - this.left();
    int height = this.bottom() - this.top();
    return width * height;
}

// Shape.class
...
int area = rectangle.area()
...
```

메서드 객체(Method Object)

```
int complexCalculation(int one, int two, int three, Obejct _mos) {
        int result1 = one + two + three + _mos.otherMethod();
        int result2 = one * two * three * _mos.otherMethod();
        return result1 + result2;
    }
```

```
class ComplexCalculator {
    private int one;
    private int two;
    private int three;
    private int result1;
    private int result2;
    private Obejct _mos;

    public ComplexCalculator(int one, int two, int three, Obejct _mos) {
        this.one = one;
        this.two = two;
        this.three = three;
        this._mos = _mos;
    }

    int calculate() {
        result1 = one + two + three + _mos.otherMethod();
        result2 = (one * two * three) + _mos.otherMethod();
        return result1 + result2;
    }
}
```

매개변수 추가

1. 메서드가 인터페이스에 선언되어 있다면 일단 **인터페이스에 매개 변수를 추가**한다.
2. **매개 변수를 추가**한다.
3. **컴파일 에러**가 여러분에게 어딜 고쳐야 하는지 알려줄 것이다. 이것을 이용하라.

메서드 매개변수를 생성자 매개변수로 바꾸기

1. 생성자에 매개 변수를 추가한다.
2. 매개 변수와 같은 이름을 갖는 인스턴스 변수를 추가한다.
3. 생성자에서 인스턴스 변수의 값을 설정한다.
4. 'parameter'를 'this.parameter'로 하나씩 찾아 바꾼다.
5. 매개 변수에 대한 참조가 더 이상 존재하지 않으면 해당 매개 변수를 메서드와 모든 호출자에서 제거한다.
6. 이제 필요 없어진 this.를 제거한다.
7. 변수명을 적절히 변경한다.

32

TDD 마스터하기

단계가 얼마나 커야 하는가?

1. 각 테스트가 다뤄야 할 범위는 얼마나 넓은가?
2. 리팩토링을 하면서 얼마나 많은 중간 단계를 거쳐야 하는가?

한 줄 로직 추가+리팩토링, 수백 줄 로직 추가+리팩토링
둘 다 할 수 있어야 한다!

시간이 지남에 따라 테스트 주도 개발자의 경향은 단계가 점점 작아지는 방향을 나타남

리팩토링 초기에는 아주 작은 단계부터 시작
매우 작은 단계로 수작업 리팩토링을 20번 하고 나면, 몇 단계 건너 뛰는 실험을 해보라

테스트할 필요가 없는 것은?

애플리케이션 개발에 있어서 **두려움이 지루함으로 변할때까지 테스트를 만드는 것**이다.

그러나 결국 스스로 대답을 찾아야만 한다. 다음 것들을 테스트해야 한다.
조건문, 반복문, 연산자, 다형성

당신이 작성하는 것들에 대해서만 테스트하라.
불신할 이유가 없다면 **다른 사람이 만든 코드를 테스트하지 마라.**

좋은 테스트를 갖췄는지의 여부를 어떻게 알 수 있는가?

다음은 설계 문제가 있음을 알려주는 테스트의 속성이다.

긴 셋업 코드 : 객체가 너무 크다는 뜻이므로 나뉠 필요가 있다.

셋업 중복 : 공통의 셋업 코드를 넣어 둘 공통의 장소를 찾기 힘들다면, 서로 밀접하게 엉킨 객체들이 너무 많다는 뜻이다.

실행 시간이 오래 걸리는 테스트 : 실행하는 데 오래 걸리면 테스트를 자주 실행하지 않게 되고, 한동안 실행이 안 된 채로 남게 되는 경우가 종종 있고, 이렇게 되면 테스트가 아예 동작하지 않을 수도 있다. 더 나쁜 점은, 애플리케이션의 작은 부분만 따로 테스트하기가 힘들다는 것을 의미한다.

깨지기 쉬운 테스트 : 예상치 못하게 실패하는 테스트가 있다면 이는 애플리케이션의 특정 부분이 다른 부분에 이상한 방법으로 영향을 끼친다는 뜻이다. 연결을 끊거나 두 부분을 합하는 것을 통해 멀리 떨어진 것의 영향력이 없어지도록 설계해야 한다.

TDD로 프레임워크를 만들려면 어떻게 해야 하나?

테스트 주도 개발은 비록 발생하지 않은 변주 종류는 잘 표현하지 못할지라도, 발생하는 변주 종류는 그것들을 잘 표현하는 프레임워크를 만들게 해 준다.

그래서 3년 후에 일반적이지 않은 변화가 발생하면 어떻게 될까?
그 변화를 수용하기 위해 정확히 필요한 지점에서 설계가 급격한 진화를 거치게 된다.

이는 개방-폐쇄 원칙을 잠시 위배하게 되지만, 이에 따른 비용은 크지 않다.
왜냐하면 스스로 뭔가를 잘못하지 않았다는 확신을 줄 수 있는 많은 테스트들이 존재하기 때문이다.

피드백이 얼마나 필요한가?

테스트를 얼마나 작성할지 고려할 때 실패간 평균시간을 고려해야한다.

삼각형의 각 변의 길이를 나타내는 세 개의 정수를 받아서 다음 값을 반환하는 문제다.

정삼각형이면 1을 반환
이등변삼각형이면 2를 반환
부등변삼각형이면 3을 반환
삼각형이 될 수 없다면 예외를 던진다.

필자는 6개의 테스트 작성, 누군가는 65개의 테스트 작성

TDD의 테스트에 대한 관점은 **실용적**이다.
만약 어떤 구현에 대한 지식이 신뢰할 만 하다면 그에 대한 테스트는 작성하지 않을 것이다.

테스트를 지워야 할 때는 언제인가?

첫째 기준 : 자신감
테스트를 삭제할 경우 자신감이 줄어들 것 같으면 절대 테스트를 지우지 말아야 한다.

둘째 기준 : 커뮤니케이션
두 개의 테스트가 코드의 동일한 부분을 실행하더라도, 이 둘이 서로 다른 시나리오를 말한다면 그대로 남겨두어야 한다.

프로그래밍 언어와 환경이 TDD에 어떤 영향을 주는가?

TDD 주기(테스트/컴파일/실행/리팩토링)를 수행하기가 힘든 언어나 환경에서 작업하게 되면 **단계가 커지는 경향**이 있다.

- 각 테스트가 더 많은 부분을 포함하게 만든다.
- 중간 단계를 덜 거치고 리팩토링을 한다.

거대한 시스템을 개발할 때에도 TDD를 할 수 있는가?

시스템에 있는 기능의 양은 TDD의 효율에 영향을 미치지 않음.

중복을 제거함에 더 작은 객체들이 만들어지고,
애플리케이션 크기와 무관하게 독립적으로 테스트 될 수 있음.

애플리케이션 수준의 테스트로도 개발을 주도할 수 있는가?

애플리케이션 테스트 주도 개발(ATDD, Application Test-Driven Development)에는 사회적인 문제가 존재한다.

사용자에게 테스트를 작성시키는 책임을 부여하는 것이다.

또 다른 문제는 테스트와 피드백 사이의 길이다. 만약 고객이 테스트를 작성하고 통과하기까지 열흘이 걸린다면, 거의 열흘동안 빨간 막대만 보게 될 것이다.

필자는 앞으로도 프로그래머 수준의 TDD를 원할 것 같다.

즉시 초록 막대를 볼 수 있고

내부 설계를 단순화할 수 있길 원한다.

프로젝트 중반에 TDD를 도입하려면 어떻게 해야 할까?

테스트를 염두에 두지 않고 만든 코드는 테스트하기 쉽지 않다

"고치면 되지 않은가?"
리팩토링 과정에서 에러가 발생할 수도 있는데 아직 테스트가 없기 때문에 에러가 생겼다는 점을 알아내기가 힘들다. (테스트와 리팩토링 사이에 존재하는 **교착 상태**)

따라서 먼저 해야 할 일은 **변경의 범위를 제한하는 것**이다.
지금 당장 변할 필요가 없는 부분을 봤다면, 그냥 그대로 놔둘 것이다.

그 다음으로는 **테스트와 리팩토링 사이에 존재하는 교착 상태(deadlock)를 풀어주는 것**이다.
테스트가 아닌 다른 방법으로도 피드백을 얻을 수 있는데,

1. 아주 조심스럽게 작업을 하거나
2. 파트너와 함께 작업을 하는 방법 등이 그런 것이다.

TDD는 누구를 위한 것인가?

만약 어느 정도는 작동하는 코드를 한번에 몰아 입력해 넣는 것에 행복해 하고, 그 결과를 두 번 다시 쳐다보지 않는 것에 행복해 한다면, TDD는 당신을 위한 것이 아니다.

TDD는 **더 나은 코드를 작성한다면 좀더 성공할 것**이라는, 매력적일 정도로 나이브하며 해커적인(geekoid) 가정에 근거한다.

TDD는 코드에 **감정적 애착을 형성**하는 해커들에게 좋다.

사실 TDD는 오버액션이다.
TDD는 현재 업계에서 통용하는 수준보다 **훨씬 더 적은 수의 결함과 훨씬 더 깨끗한 설계**의 코드를 작성하게 해준다.

TDD는 누구를 위한 것인가?

안 좋은 사이클
1. 엄청난 흥미로 새 프로젝트 시작
2. 시간이 지남에 따라 서서히 코드가 썩음
3. 하루빨리 버려버리고 새로운 프로젝트가 시작되기를 기다림

그러나 TDD는 시간이 지남에 따라 코드에 대한 자신감을 점점 더 쌓아갈 수 있게 해준다.

테스트가 쌓여감에 따라(그리고 테스팅 기술이 늘어감에 따라) 시스템의 행위에 대한 자신감을 얻게 되고, 설계를 개선해 나감에 따라 점점 더 많은 설계 변경이 가능해진다.

TDD는 초기 조건에 민감한가?

테스트를 특정 순서로 구현하는 것이 다른 순서에 비해 훨씬 빠르고 쉽다는 것이 사실인가?

단지 내가 구현 기술이 부족해서는 아닐까?

혹 테스트를 특정 순서로 공략해야 한다는 것을 넌지시 알려주는 무언가가 테스트 속에 있는 건 아닐까?

TDD와 패턴의 관계는?

반복적 행동 → 규칙 → 기계적이며 단순 암기
처음에는 규칙을 찾아보거나 새로운 규칙을 기록하기 위해 속도가 훨씬 느리지만, 일주일 후에는 속도가 훨씬 빨라진다.

TDD와 패턴의 또 다른 관계는 패턴 주도 설계(Pattern-Driven Design)에 대한 구현 방법으로써 TDD다.

어떤 작업을 수행하기 위해 전략 패턴을 사용하기로 결정했다고 가정해보자.
그 다음 리팩토링 단계에서 자연스럽게 전략 패턴이 나타날 수 있도록 하기 위해 의식적으로 두 번째 테스트를 작성해본다.

문제는 설계가 **항상 의도하지 않은대로 흘러간다는 것**이다.
완벽하게 사리에 맞는 설계 아이디어가 결국은 틀린 것으로 판명난다.
그냥 시스템이 무슨 일을 할지 생각하고 나중에 설계가 알아서 정해지도록 하는 것이 더 낫다.

어째서 TDD가 잘 작동하는가?

TDD는 결함을 빨리 발견해 고칠 수 있게함으로써 비용을 낮출 수 있다.

그러면서 결함 감소에서 오는 이차적인 심리학적, 사회적인 효과가 많다.

- 결함은 빨리 발견해 고칠수록 비용이 낮아진다.
- 팀원과의 관계가 더 긍정적으로 변했다.
- 사람들은 내 소프트웨어로 작업하는 것을 신뢰할 수 있다.
- 새로 릴리즈한 시스템은 더 이상 새로운 버그의 근원지가 아니다.

TDD의 또다른 효과는 설계 결정에 대한 피드백 고리를 단축시킨다는 점이다.

설계에 대한 생각과 그 첫 번째 예제 사이의 간격이 피드백 고리이다.

이름을 테스트 주도 개발이라고 한 이유는?

개발
소프트웨어 개발을 어떤 단계(분석 부터 배포에 이르는)에 따라 나누는 과거의 사고방식은 약화됐다. 분석, 논리적 설계, 물리적 설계, 구현, 테스팅, 검토, 통합, 배포를 아우르는 것을 개발이라 한다.

주도
어떤 이름의 반대는 최소한 모호하게라도 불만족스러워야 한다는 명명 규칙에 따라
'테스트 우선'에서 '테스트 주도'로 바꿈 (‘우선’의 반대는 ‘마지막’이므로 명확함)

테스트
자동화되고 추제적이고 명확한 테스트를 말한다.
TDD는 테스트 기술이 아니라 분석 기술이며 설계 기술이다.

TDD와 XP의 실천법 사이에 어떤 관련이 있는가?

짝 프로그래밍
짝으로 일하는 것은 TDD를 강화시킴. 그 리듬 때문에 몰입할 수 있으며, 지칠 때 활기를 줌.
작성하게 되는 테스트는 짝 프로그래밍 과정에서 좋은 의사소통 수단이 됨.

활기차게 일하기
기운이 있을 때 일을 시작해, 지치면 그만할 것을 권유

지속적인 통합
테스트는 더 자주 통합할 수 있게 해주기 때문에 훌륭한 자원이 됨.
테스트 통과 및 중복을 제거 후에 체크인.

TDD와 XP의 실천법 사이에 어떤 관련이 있는가?

단순설계
테스트를 통과하기 위해 최소한의 필요한 코딩을 하고 중복을 제거한다면, 자동으로 요구사항에 맞는 설계를 얻음.

리팩토링
테스트가 있다면 큰 리팩토링을 수행하더라도 시스템의 행위가 변하지 않았다는 자신감을 얻을수 있음.

지속적인 전달
고객을 혼란시키지 않으면서도 더 자주 코드를 출시할 수 있음