리팩토링이란?
리팩토링과 디자인 패턴의 관계

Jia2-Chap09B 보충강의

자바 디자인 패턴과 리팩토링 (2003)

1부 객체지향이란 무엇인가?

- 1장 객체지향의 밑바닥
- 2장 UML 입문

2부 자바의 객체지향 특성 100% 활용

- 3장 클래스와 상속, 그리고...
- 4장 확장성 있고 유연한 자바 프로그램 만들기

3부 객체지향 소프트웨어 설계의 기본 원리

- 5장 소프트웨어의 변화와 포용
- 6장 변화를 수용하는 객체지향 설계의 원리

4부 GoF의 디자인패턴

7장 디자인패턴 소개

8장 GoF 디자인패턴 리스트

01 Abstract Factory
02 Factory Method
03 State & Stratege
04 Templet Method
05 Decorator
06 Composite
07 Bridge & Adapter
08 Mediator
09 Observer
10 Command
11 그밖의 유용한 패턴

9장 JHotdraw 프레임워크의 디자인패턴

제5부 리팩토링

10장 리팩토링 소개

01. 개요

리팩토링이란 (p. 662-668)
리팩토링과 디자인패턴의 관계 (p. 668-675)

11장 리팩토링 카탈로그

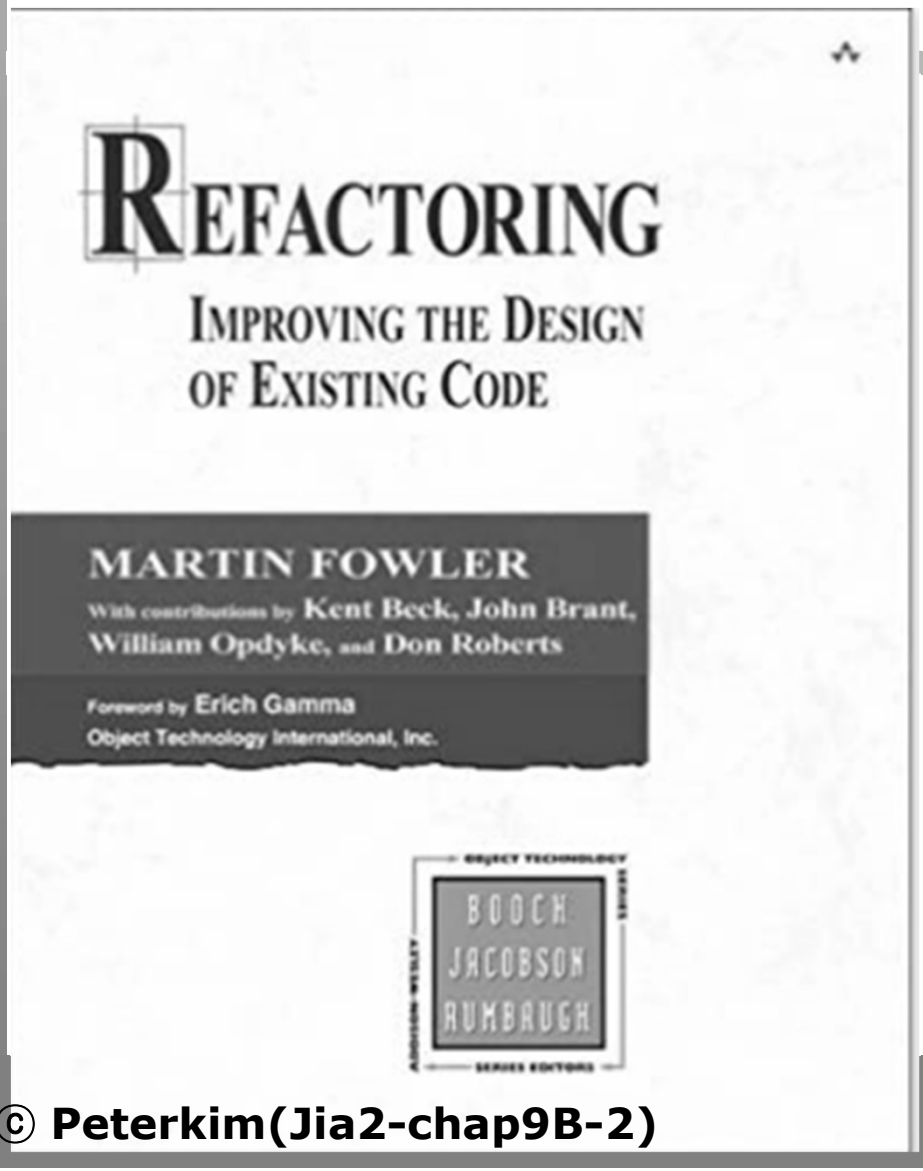
12장 종합 대여 관리 시스템의 리팩토링 실습

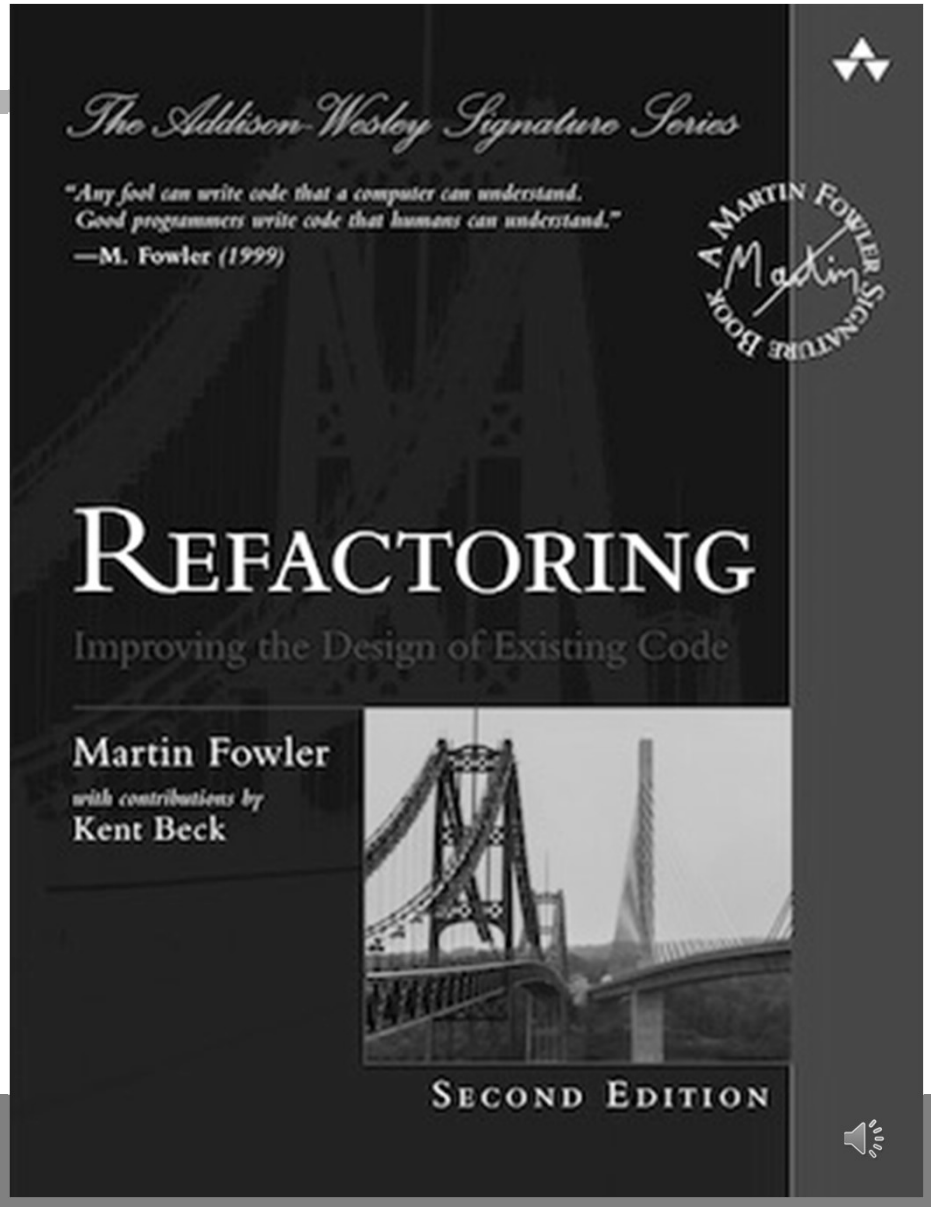
부록 : 짝 프로그래밍을 이용한 객체지향 설계 협력 학습 방법

리팩토링이란?

- 리팩토링이란?
  - 프로그램의 행위는 변경하지 않으면서 프로그램을 쉽게 이해하고 변경할수 있도록 프로그램의 내부 구조를 수정하는 것 [Fowler 99]
  - 프로그램을 이해하기 쉽게 , 그리고 변화를 포용할수 있도록 단계적으로 개선

Fowler





- 리팩토링하는 순서
  - 1) 먼저 소스코드를 관찰하여 이해하기 어렵거나 추후 쉽게 변경하기 어려운 코드 조각들을 꼼꼼히 찾음
  - 2) 문제가 있는 코드를 발견하면 그 문제를 해결할 리팩토링 방법을 선택하여 코드를 수정
  - 3) 리팩토링한 이후 코드가 제대로 동작하는지를 테스트하고 확인

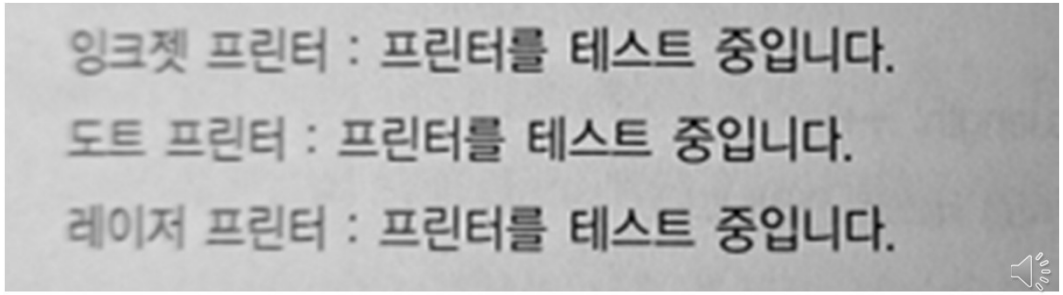
간단한 리팩토링 예제

- 프린터 출력 프로그램
  - 잉크젯, 레이저, 도트 프린터에게 메시지를 출력하도록 명령하는 프로그램
  - (고려사항) 프린터의 종류에 따라 출력하는 방식이 다름
  - (고려사항) 새로운 프린터를 추가할 필요성

  - 예제 10-1 고려사항을 생각하지 않고 일단 만든 코드

[예제 10-1]

```
/*
 * 이 코드는 "자바 디자인패턴과 리팩토링(박지훈 저, 한빛 미디어, 2003)"에 포함된 것입니다.
 * 비 상업적인 용도라면 마음대로 사용, 변경, 배포할 수 있습니다.
 * 단, 소스코드를 배포할 때 이 코멘트는 그대로 유지되어야 합니다.
 * http://www.pairprogrammer.com/book/DesignPatternAndRefactoring
 */

package refactoring.printer.before;

public class PrinterExample {

    public static void main(String[] args) {

        Printer[] printers = new Printer[3];

        printers[0] = new Printer(0); // 잉크젯 프린터
        printers[1] = new Printer(1); // 도트 프린터
        printers[2] = new Printer(2); // 레이저 프린터

        for (int i=0; i < printers.length; i++)
            printers[i].print("프린터를 테스트 중입니다.");
    }
}
```



```
class Printer {
        private int type; // 0 : 잉크젯, 1 : 도트, 2: 레이저

        public Printer(int type) {
                this.type = type;
        }

        public void print(String msg) {
                if ( type == 0 )
                        print0(msg);
                else if ( type == 1 )
                        print1(msg);
                else if ( type == 2 )
                        print2(msg);
    }

    private void print0(String msg) {
        System.out.println("잉크젯 프린터 : " + msg);
    }

    private void print1(String msg) {
        System.out.println("도트 프린터 : " + msg);
    }

    private void print2(String msg) {
        System.out.println("레이저 프린터 : " + msg);
    }
}
```

■ 예제 10-1

- 실행결과

```
잉크젯 프린터 : 프린터를 테스트 중입니다.
도트 프린터 : 프린터를 테스트 중입니다.
레이저 프린터 : 프린터를 테스트 중입니다.
```

- 평가
  - Type에 따라서 수행할 메소드가 달라짐
  - 해당 메소드가 서로 다르게 구현되어 있음
  - 개발자가 이해하기 어렵거나, 향후 코드를 변경하기 어려운 부분은??
  - 예) 0, 1,2 의 값의 의미를 미리 알고 있어야 함
  - 예) 새로운 프린터가 추가될때 if 문이 바뀌어야 함

- 파울러의 리팩토링 방식 제안
  - “타입코드를 하우 클래스로 바꾸기“
  - (Replace type code with subclasses)
  - 프린터의 타입을 결정하는 속성인 int type을 없앰
  - 타입의 값에 해당하는 하우 클래스를 정의

- [예제 10-2]
  - 예제 10-1을 리팩토링한 결과

[예제 10-2]

```
/*
 * 이 코드는 "자바 디자인패턴과 리팩토링(박지훈 저, 한빛 미디어, 2003)"에 포함된 것입니다.
 * 비 상업적인 용도라면 마음대로 사용, 변경, 배포할 수 있습니다.
 * 단, 소스코드를 배포할 때 이 코멘트는 그대로 유지되어야 합니다.
 * http://www.pairprogrammer.com/book/DesignPatternAndRefactoring
 */

package refactoring.printer.after;

public class PrinterExample {

    public static void main(String[] args) {

        Printer[] printers = new Printer[3];

        printers[0] = new InkjetPrinter();
        printers[1] = new DotPrinter();
        printers[2] = new LaserPrinter();

        for (int i=0; i < printers.length; i++)
            printers[i].print("프린터를 테스트 중입니다.");
    }
}
```

```
abstract class Printer {
    abstract public void print(String msg);
}

class InkjetPrinter extends Printer {
        public void print(String msg) {
                System.out.println("잉크젯 프린터 : " + msg);
        }
}

class DotPrinter extends Printer {
        public void print(String msg) {
                System.out.println("도트 프린터 : " + msg);
        }
}

class LaserPrinter extends Printer {
        public void print(String msg) {
                System.out.println("레이저 프린터 : " + msg);
        }
```

- [10-2]
  - 쉽게 이해
  - 유지보수하기 쉬움
  - 예제 10-1에서는 프린터의 종류를 명시하기 위해 주석이 필요
  - 예제 10-2에서는 하위클래스의 이름이 프린터이므로 주석이 불필요
  - 새로운 프린터 추가 용이

- 리팩토링의 효과와 안정성
  - 1) 소프트웨어 설계의 품질이 개선
  - 2) 소스를 이해하기 쉬움
  - 3) 버그를 발견하기 쉬움
  - 4) 프로그램을 더 빨리 개발

디자인패턴 & 리팩토링

- 디자인패턴&리팩토링 (공통적인) 지향점
  - 변화를 포용할 수 있는 유지보수성이 뛰어난 소프트웨어 개발

- 개발 접근#1
  - 코딩을 시작하기 전에
  - 많은 시간을 들여서
  - 미리 소프트웨어의 구조를 UML 모델링 언어를 사용하여
  - 어떤 디자인패턴 등을 적용할지 그래서 가능한한 이상적인 SW의 구조를 설계
  - 그 이후 코딩

- 개발 접근 #2
  - 일단 코딩
  - (이미 만들어져있는 코드에) 리팩토링을 적용
  - 이해하기 쉽고 변경을 포용할수 있는 구조로 변경
  - 소프트웨어의 설계 시간 최소화
  - XP 개발방법론

■ 디자인 패턴과 리팩토링의 비교

[표 10-1] 디자인패턴과 리팩토링의 비교

비교 대상	디자인패턴	리팩토링
시스템 생성력	작다	크다
적용 단위	일반적으로 크다	일반적으로 작다
개발자의 지속적인 집중력	상대적으로 낮다	상대적으로 높다
잘 맞는 방법론	RUP 등 설계 중심 방법론	XP 등 코드 진화 중심 방법론

- 마틴 파울러
  - “디자인 패턴은 리팩토링이 겨냥한 과녁 중 하나이다”
  - 코딩
  - 리팩토링 적용
  - 일부 디자인패턴 적용

프린터 예제 리팩토링 & 디자인 패턴 적용

- 프린터 예제
  - 복합 프린터를 테스트하는 프로그램
  - 잉크젯, 도트, 레이저 프린터
  - 고려할 점 – 새로운 종류의 프린터 속성이 계속 추가될 가능성
- 예제코드 10-3

[예제 10-3]

```
/*
 * 이 코드는 "자바 디자인패턴과 리팩토링(박지훈 저, 한빛 미디어, 2003)"에 포함된 것입니다.
 * 비 상업적인 용도라면 마음대로 사용, 변경, 배포할 수 있습니다.
 * 단, 소스코드를 배포할 때 이 코멘트는 그대로 유지되어야 합니다.
 * http://www.pairprogrammer.com/book/DesignPatternAndRefactoring
 */

package refactoring.printer2.before;

public class PrinterExample {

    public static void main(String[] args) {

        Printer printer = new Printer();

        printer.setMode(0); // 잉크젯 프린팅 모드
        printer.print("프린터를 테스트 중입니다.");

        printer.setMode(1); // 도트 프린팅 모드
        printer.print("프린터를 테스트 중입니다.");

        printer.setMode(2); // 레이저 프린팅 모드
        printer.print("프린터를 테스트 중입니다.");
    }
}
```

```
class Printer {
        private int type; // 0 : 잉크젯, 1 : 도트, 2: 레이저

        public Printer() {
                type = 0; // 기본 잉크젯
        }

        public void setMode(int type) {
                this.type = type;
        }

        public void print(String msg) {
                if ( type == 0 )
                        print0(msg);
                else if ( type == 1 )
                        print1(msg);
                else if ( type == 2 )
                        print2(msg);
    }
    private void print0(String msg) {
        System.out.println("잉크젯 프린터 : " + msg);
    }

    private void print1(String msg) {
        System.out.println("도트 프린터 : " + msg);
    }

    private void print2(String msg) {
        System.out.println("레이저 프린터 : " + msg);
    }
}
```

- 예제 10-3 의 문제점
  - Printer 객체는 시간에 따라 그 값이 바뀜
  - ➔ 타입코드를 State/Strategy 패턴으로 바꾸기 라는 리팩토링을 적용
    - Replace Type Code with State/Strategy 적용
    - 타입코드를 GoF의 State 객체로 교체하도록 코드를 변경
    - ➔ 예제 10-4

[예제 10-4]

```
/*
 * 이 코드는 "자바 디자인패턴과 리팩토링(박지훈 저, 한빛 미디어, 2003)"에 포함된 것입니다.
 * 비 상업적인 용도라면 마음대로 사용, 변경, 배포할 수 있습니다.
 * 단, 소스코드를 배포할 때 이 코멘트는 그대로 유지되어야 합니다.
 * http://www.pairprogrammer.com/book/DesignPatternAndRefactoring
 */

package refactoring.printer2.before;

public class PrinterExample {

    public static void main(String[] args) {

        Printer printer = new Printer();

       printer.setMode(0); // 잉크젯 프린팅 모드
        printer.print("프린터를 테스트 중입니다.");

        printer.setMode(1); // 도트 프린팅 모드
        printer.print("프린터를 테스트 중입니다.");

        printer.setMode(2); // 레이저 프린팅 모드
        printer.print("프린터를 테스트 중입니다.");
    }
}
```

```
class Printer {
        private PrinterImpl pImpl; // 프린팅 모드

        public void setMode(PrinterImpl pImpl) {
                this.pImpl = pImpl;
        }

        public void print(String msg) {
                pImpl.print(msg);
        }
}

abstract class PrinterImpl {
   abstract public void print(String msg);
}

class InkjetPrinterImpl extends PrinterImpl {
        public void print(String msg) {
                System.out.println("잉크젯 프린터 : " + msg);
        }
}

class DotPrinterImpl extends PrinterImpl {
        public void print(String msg) {
                System.out.println("도트 프린터 : " + msg);
        }
}

class LaserPrinterImpl extends PrinterImpl {
        public void print(String msg) {
```

- 예제 10-4
  - 예제 10-3에 state 패턴을 적용
  - 리팩토링 과정을 거쳐 적용
  - 만약 리팩토링을 거치지 않고 처음부터 디자인패턴을 적용하려는 시도???
  - 디자인 패턴을 무리하게 적용하게 됨
  - 디자인 패턴을 적용할 필요가 없는 곳에 끼어맞추기
    - 망치로 모기를 잡는 격

- 경험이 적은 개발자
  - ➔ 빨리 먼저 프로그램을 완성하고 나서
    - 소스코드에서 문제점을 발견하고
    - 리팩토링의 과정을 거쳐 코드를 변경함을 권유

- 경험이 많은 개발자
  - 특정 디자인 패턴을 구현하는 것이 더 효과적이라고 사전에 판단
  - 설계 단계에서 미리 결정
  - 리팩토링 단계를 미리 생략

- 리팩토링을 사용하기를 추천하는 경우 [Fowler99]
  - 새로운 기능을 추가할때
  - 코드를 리뷰할때
  - 버그를 잡거나 사전에 방지하고자 할때
  - XP 개발 방법론에 따라 프로그램을 개발할때