[우리 학과 취업스쿨 Day10]

소프트웨어 디자인 패턴과 리팩토링

리팩토링(Refactoring) 실습

-프린터 관리 예제1-

프린터 관리 예제

❖ 학습목표

- 프린터 관리 프로그램을 통해 **클래스를 추출하기**, **상위 클래스를 추출하기**, **인터페이스 추출하기**, **템플릿 메소드 만들기**, **메소드 옮기기**, **메소드 합치기**, **생성자를 팩토리 메소드로 바꾸기** 등의 리팩토링을 적용해 보자.

프린터 관리 예제

- 잉크젯, 도트, 레이저 프린터와 PDF 파일 생성기를 테스트 하는 데모 프로그램을 만들어보자. 잉크젯, 레이저 프린터는 매번 프린트할 때마다 각각 잉크와 토너를 소비하며 각각의 소비율은 서로 다르다. 사용하려는 프린터의 잉크나 토너의 용량이 다 떨어진 상태에서 프린트를 시도하면 경고 메시지를 보여준다. 반면 도트 프린터는 반영구적으로 사용할 수 있다. PDF 파일 생성기는 지정된 이름의 PDF 파일을 생성하고 그 안에 문서 내용을 출력한다고 가정한다.

- 먼저 잉크젯(InkjetPrinter), 도트(DotPrinter), 레이저(LaserPrinter) 프린터 클래스와 PDF 생성기(PDFWriter)클래스를 만듬

refactoring.examples.printer.one.PrinterExample.java

```
package refactoring.examples.printer.one;

/* Extract SuperClass */
public class PrinterExample {

	public static void main(String[] args) {

		InkjetPrinter iPrinter = new InkjetPrinter("101");
		DotPrinter dPrinter = new DotPrinter("102");
		LaserPrinter lPrinter = new LaserPrinter("103");
		PDFWriter pWriter = new PDFWriter("Test.pdf");

		iPrinter.print("환영합니다. 프린터를 테스트중입니다.");
		dPrinter.print("환영합니다. 프린터를 테스트중입니다.");
		lPrinter.print("환영합니다. 프린터를 테스트중입니다.");
		pWriter.print("환영합니다. 프린터를 테스트중입니다.");
	}
}
```

실행결과

```
*잉크젯 방식으로 프린트를 시작합니다.*
환영합니다. 프린터를 테스트중입니다.
*잉크젯 방식으로 프린트를 종료합니다.*
*도트 방식으로 프린트를 시작합니다.*
환영합니다. 프린터를 테스트중입니다.
*도트 방식으로 프린트를 종료합니다.*
*레이저 방식으로 프린트를 시작합니다.*
환영합니다. 프린터를 테스트중입니다.
*레이저 방식으로 프린트를 종료합니다.*
*문서내용을 Test.pdf 파일에 PDF 포맷으로 출력하기 시작합니다.*
환영합니다. 프린터를 테스트중입니다.
*문서내용을 PDF 포맷으로 출력을 완료했습니다.
```

refactoring.examples.printer.one.PrinterExample.java

```
class InkjetPrinter {
	private double inkCapacity; // 잉크의 용량
	private double inkReductionRate; // 한번 프린트할 때마다 줄어드는 잉크의 비율

	private String ID;

	public InkjetPrinter(String ID) {
		this.ID = ID;
		inkCapacity = 100;
		inkReductionRate = 0.5;
	}

	public String getID() {
		return ID;
	}

	public void print(Object msg) {
		if ( isPrintable() ) {
			System.out.println("*잉크젯 방식으로 프린트를 시작합니다.*");
			System.out.println(msg.toString());
			System.out.println("*잉크젯 방식으로 프린트를 종료합니다.*");

			inkCapacity -= inkReductionRate;
		} else
			alert();
	}

	public void alert() {
		System.out.println("잉크가 부족합니다. 빨간 램프를 깜박깜박~ ");
	}

	public boolean isPrintable() { // 한장을 찍을 분량이 남아있으면,
		return (inkCapacity - inkReductionRate) >= 0;
	}

	public void testPrinting() {
		print("아아~ 프린트 테스트. 프린트 테스트");

	}
}
```

refactoring.examples.printer.one.PrinterExample.java

```
class DotPrinter {
	private String ID;

	public DotPrinter(String ID) {
		this.ID = ID;
	}

	public String getID() {
		return ID;
	}

	public void print(Object msg) {
		if ( isPrintable() ) {
			System.out.println("*도트 방식으로 프린트를 시작합니다.*");
			System.out.println(msg.toString());
			System.out.println("*도트 방식으로 프린트를 종료합니다.*");
		}
	}

	public boolean isPrintable() {
		return true;
	}

	public void testPrinting() {
		print("아아~ 프린트 테스트. 프린트 테스트");

	}
}
```

refactoring.examples.printer.one.PrinterExample.java

```
class LaserPrinter {
        private double tonerCapacity; // 토너의 용량
        private double tonerReductionRate; // 한번 프린트할 때마다 줄어드는 토너의 비율

        private String ID;

        public LaserPrinter(String ID) {
                this.ID = ID;
                tonerCapacity = 100;
                tonerReductionRate = 0.2;
        }

        public String getID() {
                return ID;
        }

        public void print(Object msg) {
                if ( isPrintable() ) {
                        System.out.println("*레이저 방식으로 프린트를 시작합니다.*");
                        System.out.println(msg.toString());
                        System.out.println("*레이저 방식으로 프린트를 종료합니다.*");

                        tonerCapacity -= tonerReductionRate;
                } else
                        alert();
        }

        public void alert() {
                System.out.println("토너가 부족합니다. 노란 램프를 깜박깜박~ ");
        }

        public void alert() {
                System.out.println("토너가 부족합니다. 노란 램프를 깜박깜박~ ");
        }

        public boolean isPrintable() { // 한장을 찍을 분량이 남아있으면,
                return (tonerCapacity - tonerReductionRate) >= 0;
        }

        public void testPrinting() {
                print("아아~ 프린트 테스트. 프린트 테스트");

        }
```

리팩토링1. 상위 클래스 추출하기

❖ 중복(메소드)

- 클래스들끼리 같은 종류의 속성과 메소드가 존재하는 경우
  - **상위 클래스를 추출하기(Extract Superclass) 리팩토링**

❖ Printer라는 상위 클래스를 정의하고, 그 안에 서로 중복되는 속성과 메소드를 올리자.

리팩토링1. 상위 클래스 추출하기 리팩토링 적용 소스

리팩토링2. 템플릿 메소드 만들기

❖ 중복(메소드)

- 여전히 나쁜 코드 조각 발견됨
- 하위 클래스들의 메소드들 중 **코드 뼈대의 구조가 중복되는 경우(템플릿 메소드 패턴 적용)**
  - **템플릿 메소드를 만들자.(Form Template Method)**

❖ Printer 의 하위 클래스에 속한 print() 메소드를 템플릿 메소드로 만들기

```
public void print(Object msg) {
    if ( isPrintable() ) {
        System.out.println("*레이저 방식으로 프린트를 시작합니다.*");
        System.out.println(msg.toString());
        System.out.println("*레이저 방식으로 프린트를 종료합니다.*");

        tonerCapacity -= tonerReductionRate;
    } else
        alert();
}
```

```
public void print(Object msg) {
    if ( isPrintable() ) {
        System.out.println("*잉크젯 방식으로 프린트를 시작합니다.*");
        System.out.println(msg.toString());
        System.out.println("*잉크젯 방식으로 프린트를 종료합니다.*");

        inkCapacity -= inkReductionRate;
    } else
        alert();
}
```

```
public void print(Object msg) {
    if ( isPrintable() ) {
        System.out.println("*도트 방식으로 프린트를 시작합니다.*");
        System.out.println(msg.toString());
        System.out.println("*도트 방식으로 프린트를 종료합니다.*");
    }
}
```

리팩토링2. 템플릿 메소드 만들기 리팩토링 적용 소스

리팩토링3. 클래스를 추출하기

❖ 중복

- 여러 클래스에서 공통적으로 속성들의 그룹이 나타나는 경우(속성)
  - **클래스를 추출하기(Extract Class)**
- 관계없는 다른 클래스와 공통된 메소드가 존재하는 경우(메소드)
  - **인터페이스를 추출하기(Extract Interface)**

❖ InkjetPrinter, LaserPrinter 클래스는 비슷한 속성의 그룹을 갖고 있다.

- 프린팅 원료의 전체 용량(xxxCapacity)
- 프린팅 때 소비되는 용량(xxxReductionRate)에 대한 변수
- **변수명은 다르지만 역할 즉 프린터 카트리지에 대한 정보이다.**
- 속성의 그룹이 별도의 의미와 역할을 가질 때, **이를 클래스로 추출하기 리팩토링을 적용한다.**

리팩토링3. 클래스를 추출하기 적용 소스

리팩토링4. 인터페이스를 추출하기

- Printer 클래스
  - 프린터를 상징한다.
- PDFWriter클래스
  - 프린터 클래스와 유사한 기능이 있다.
  - print(): PDF문서를 출력한다.

- 두 클래스 '무언가를 출력한다'라는 동일한 기능을 수행하므로 그 기능을 인터페이스 타입으로 추상화 할 수 있다 . 이렇게 서로 다른 클래스에서 동일한 기능의 메소드가 발견되면, **"인터페이스를 추출하기" 리팩토링을 적용할 수 있다.**

리팩토링4. 인터페이스를 추출하기 적용 소스

- interface Printable 정의하기
  - print() 선언하기

리팩토링5. 메소드를 이동하기

- 메소드가 주로 *다른 객체의 메소드 반환값을 이용하여 일을 하는 경우*
  - **메소드를 이동하기(Move Method)**

```
public void printing(Object msg) {
        System.out.println("*잉크젯 방식으로 프린트를 시작합니다.*");
        System.out.println(msg.toString());
        System.out.println("*잉크젯 방식으로 프린트를 종료합니다.*");

        cartridge.setCapacity(cartridge.getCapacity() - cartridge.getReductionRate());

}
```

```
public boolean isPrintable() { // 한장을 찍을 분량이 남아있으면,
        return (cartridge.getCapacity() - cartridge.getReductionRate()) >= 0;
}
```

리팩토링5. 메소드를 이동하기

❖ 메소드가 주로 *다른 객체의 메소드 반환값을 이용하여 일을 하는 경우*

- **메소드를 이동하기(Move Method)**

```
public void printing(Object msg) {
        System.out.println("*레이저 방식으로 프린트를 시작합니다.*");
        System.out.println(msg.toString());
        System.out.println("*레이저 방식으로 프린트를 종료합니다.*");

        cartridge.setCapacity(cartridge.getCapacity() - cartridge.getReductionRate());

}
```

```
public boolean isPrintable() { // 한장을 찍을 분량이 남아있으면,
        return (cartridge.getCapacity() - cartridge.getReductionRate()) >= 0;
}
```

리팩토링5. 메소드를 이동하기

- isPrintable(),printing()메소드에서 PrintCatridge객체를 사용하는 코드들은 단지 잉크나 토너의 현재용량과 소비 비율을 알아내서 프린트가 가능한지를 확인 한 후, 만약 그렇다면 출력 후 용량을 줄이는 기능을 한다.
- PrintCatridge 클래스는 단지 프린트 카트리지의 데이터만 저장하는 클래스이다. 하지만, 프린터 카트리지의 현재용량이 프린트가 가능한 상태인지 확인하고 프린트한 후 용량을 소비시키는 **코드는 그것을 수행할 때 필요한 데이터(capacity, reductionRate)가 있는 곳인 PrintCatridge클래스에 정의 되는 것이 좋다.** 어떤 기능은 그 기능과 함께 사용하는 데이터와 같은 클래스에 구현해야 하기 때문이다.
- 특정 메소드의 내부 코드가 주로 다른 클래스의 속성값을 읽고 처리한다면 “메소드를 이동하기” 리팩토링을 적용해야 한다.

리팩토링5. 메소드를 이동하기 적용 소스

- PrintCatridge 클래스
  - isAvailable(), consume() 메소드 정의하기
  - 각 메소드로 코드 이동하기

리팩토링5. 메소드를 이동하기

❖ 메소드가 주로 *다른 객체의 메소드 반환값을 이용하여 일을 하는 경우*

- **메소드를 이동하기(Move Method)**

```
class InkjetPrinter extends Printer {
        private PrintCartridge cartridge;

        public InkjetPrinter(String ID) {
                super(ID);
                cartridge = new PrintCartridge(100, 0.5);
        }

        public void printing(Object msg) {
                System.out.println("*잉크젯 방식으로 프린트를 시작합니다.*");
                System.out.println(msg.toString());
                System.out.println("*잉크젯 방식으로 프린트를 종료합니다.*");

                cartridge.consume();

        }

        public void alert() {
                System.out.println("잉크가 부족합니다. 빨간 램프를 깜박깜박~ ");
        }

        public boolean isPrintable() { // 한장을 찍을 분량이 남아있으면,
                return cartridge.isAvailable();
        }
```

리팩토링5. 메소드를 이동하기

❖ 메소드가 주로 *다른 객체의 메소드 반환값을 이용하여 일을 하는 경우*

- **메소드를 이동하기(Move Method)**

```
class LaserPrinter extends Printer {
        private PrintCartridge cartridge;

        public LaserPrinter(String ID) {
                super(ID);
                cartridge = new PrintCartridge(100, 0.2);
        }

        public void printing(Object msg) {
                System.out.println("*레이저 방식으로 프린트를 시작합니다.*");
                System.out.println(msg.toString());
                System.out.println("*레이저 방식으로 프린트를 종료합니다.*");

                cartridge.consume();
        }

        public void alert() {
                System.out.println("토너가 부족합니다. 노란 램프를 깜박깜박~ ");
        }

        public boolean isPrintable() { // 한장을 찍을 분량이 남아있으면,
                return cartridge.isAvailable();
        }
}
```