공학
트렌드

소프트웨어공학센터 경영지원TF팀

클린코드 Part 2 : 적용과 개선
– 깨끗한 코드를 만들기 위한 방법 –

2015. 1. 13. [제119호]

Contents

정보통신산업진흥원 부설
nipa 소프트웨어공학센터

Ⅲ. 클린코드를 만드는 규칙들

지난 클린코드 Part 1 이해와 규칙에서는 클린코드를 위한 규칙들로 의미 있는 이름, 명확하고 간결하게 주석달기, 코드를 보기 좋게 배치하는 방법들을 소개하였다. 이번 파트에서는 착한 함수, 읽기 쉽게 흐름제어 만들기, 오류 처리 방법과 레거시 코드(legacy code)를 클린코드로 만들기 위한 프랙티스를 소개하고자 한다.

3.4 착한 함수 (Function)

함수를 만드는 원칙은 가급적 작게 만들어야 한다. if문이나 While문 안의 내용은 한 줄로 처리되도록 하는 것이 이상적이고 블록 안에서 다른 함수를 호출하도록 작성한다. 함수의 크기는 20줄 이내, 한라인당 150문자를 넘지 않도록 한다. 아래 함수는 웹기반 테스팅 프레임워크의 함수에서 if 블록문안에서 수행되는 여러 가지 일을 리팩토링하여 한 줄로 처리한 것을 볼 수 있다.

```
public String renderPageWithSetupsAndTeardowns(PageData pageData, boolean isSuite){

        If(isTestPage) {
              WikiPage testPage = pageData.getWikiPage();
              StringBuffer newPageContent = new StringBuffer();
              includeSetupPages(testPage, newPageContent, isSuite);
              newPageContent.append(pageData.getContent());
         ....
        }
        return pageData.getHTML();

}

public String renderPageWithSetupsAndTeardowns(PageData pageData, boolean isSuite){

        If(isTestPage) {
              includeSetupAndTeardownPages(pageData, isSuite);
        }
        return pageData.getHTML();

}
```

함수 하나당 하는 일은 하나만 하도록 한다. 여기서 하나의 일은 같은 추상화 수준을 의미한다. 예를 들어, getHtml( )은 높은 추상화 수준이라면 .append("₩n")는 낮은 추상화 수준이다. 이처럼 한 함수 안에 추상화 수준이 다른 것들이 혼용하여 사용하면 읽기가

어려울 뿐만 아니라 불필요한 혼동을 초래할 수 있다. 아래 함수를 보면 추상화 수준이 섞여있는 위보다 아래 함수가 훨씬 이해하기 쉽다는 것을 알 수 있다.

```
public void doTheDomesticThings() {
    takeOutTheTrash();
    walkTheDog();
    for (Dish dish : dirtyDishStack) {
       sink.washDish(dish);
       teaTowel.dryDish(dish);
    }
}
```

```
public void doTheDomesticThings() {
    takeOutTheTrash();
    walkTheDog();
    doTheDishes();
}
```

함수의 인수(parameter)는 적을수록 좋다. 가장 이상적인 함수의 인수 개수는 0개이다. 함수의 인수가 많으면 테스트 케이스를 작성하기도 어려워진다. 만약 함수에서 3개 이상의 인수가 필요하다면 객체 사용을 고려하도록 한다. 아래 그림에서는 3개의 인수를 갖는 함수를 새로운 객체로 합쳐서 사용하는 것을 보여준다.

```
Circle makeCircle(double x, double y, double radius);
```

```
Circle makeCircle(Point center, double radius);
```

함수를 만들 때 중복이 없도록 한다. 중복된 코드는 모든 소프트웨어 해악의 근본이라고 할 수 있다. 변경 시 중복된 여러 부분에 손을 대야하고 오류가 발생할 확률도 그만큼 높아지기 때문이다. 아래 그림4에서는 함수 내에서 반복되는 인자 값을 element 객체로 처리하도록 처리하여 코드 중복을 제거하였다.

```
public void bar () {
   foo ("A");
   foo ("B");
   foo ("C");
}
```

```
public void bar () {
  String [ ] elements = {"A", "B", "C"};

  for(String element : elements) {
    foo(element);
  }
}
```

3.5 읽기 쉽게 흐름제어 만들기 (Making control flow easy to read)

조건, 루프, 흐름을 통제하는 선언문이 코드에 있으면 코드를 읽기가 어려워진다. 분기문과 점프문이 코드를 복잡하게 만들기 때문이다. 논리가 명확해질 수 있는 코드흐름

을 읽기 쉽게 만드는 몇 가지 실천 방법을 살펴보자.

먼저 if/else 조건문에서 인수의 순서는 긍정적이고, 간단하고, 흥미로운 표현이 앞쪽에 위치하도록 하는 것이다. 가령 아래 표1 테이블에서 if/else 블록 안에 A와 같이 유동적이며 긍정적 질문을 받고, 흥미롭고 확실한 표현을 앞쪽에 두는 것이 B와 같이 고정적이며, 비교의 대상이 부정적으로 사용되고, 궁금한 것을 뒤에 두는 것보다 읽기가 쉽다.

표 1_if/else 블록의 순서

	if (length >= 10)		if (10 <= length)

	if (a == b) {
	// 첫 번째 경우
	} else {
	// 두 번째 경우
	}
			if (a != b) {
			// 첫 번째 경우
			} else {
			// 두 번째 경우
			}

A	if (url.HasQueryParameter("expand_all")) {
	for (int i = 0; i < items.size(); i++) {
	items[i].Expand();
	}
	....
	} else {
	response.Render(items);
	....
	}
		B	if (url.HasQueryParameter("expand_all")) {
			response.Render(items);
			....
			} else {
			for (int i = 0; i < items.size(); i++) {
			items[i].Expand();
			}
			....
			}

삼항연산자(? :)나 do/while, 혹은 goto 구문은 코드의 가독성을 떨어뜨리기 때문에 되도록 사용하지 않는 것이 바람직하다. 아래와 같은 삼항연산자의 사용을 통해 줄 수를 최소화하는 일보다 다른 사람이 코드를 읽고 이해하는데 걸리는 시간을 최소화하는 일이 중요하다. 이러한 코드는 if 문으로 작성하는 편이 자연스럽다. 삼항연산은 매우 간단한 경우에 한해 필요하다면 사용한다.

```
return exponent >= 0 ? mantissa * (1 << exponent) : mantissa / (1 << -exponent);
```

```
if (exponent >= 0) {
    return mantissa * (1 << exponent);
} else {
    return mantissa / (1 << -exponent);
}
```

일반적으로 if, while, for 문의 동작원리는 코드를 위에서 아래로 읽게 되지만 do/while 은 역순으로 읽어야하기 때문에 부자연스러운 흐름을 만들어낸다. do/while의

루프의 do는 적어도 한번은 실행되기 때문에 몇 번씩 코드를 다시 읽어야 이해할 수 있게 된다. do/while의 경우는 while문으로 변경하여 코드블록을 보기 전에 반복되는 조건을 미리 확인할 수 있도록 하는 것이 바람직하다.

```
public boolean ListHasNode(Node node, String name, int max_length) {
    do {
        if (node.name().equals(name))
            return true;
        node = node.next();
    } while (node != null && --max_length > 0);

    return false;
}
```

```
public boolean ListHasNode(Node node, String name, int max_length) {
    while (node != null && max_length-- > 0) {
        if (node.name().equals(name)) return true;
        node = node.next();
    }
    return false;
}
```

if/else 구문에서 중첩이 깊어지면 코드를 이해하기 어려워진다. 아래의 경우 코드를 읽는 사람은 머릿속에 user_result와 permission_result의 결과를 저장한 상태에서 코드를 계속 읽어나가며 각각의 if의 블록이 끝날 때마다 그에 상응하는 값을 따라서 변경해 나가야 한다. 이러한 경우에는 함수의 중간에서 변환하여 중첩을 제거함으로써 한 단계의 중첩만 가지도록 변경한다. 모든 if블록이 return과 함께 끝나기 때문에 이 코드를 읽는 사람은 머릿속에 있는 스택에서 어떤 값을 꺼낼 필요가 없게 되었다.

```
if (user_result == SUCCESS) {
    if (permission_result != SUCCESS) {
        reply.WriteErrors("error reading permissions");
        reply.Done();
        return;
    }
    reply.WriteErrors("");
} else {
    reply.WriteErrors(user_result);

}
reply.Done();
```

```
if (user_result != SUCCESS) {
    reply.WriteErrors(user_result);
    reply.Done();
    return;
}

if (permission_result != SUCCESS) {
    reply.WriteErrors(permission_result);
    reply.Done();
    return;
}

reply.WriteErrors("");
reply.Done();
```

코드에서 읽기 쉬운 좋은 흐름을 만들기 위해서는 프로그램 전체의 실행경로를 쉽게 따라갈 수 있도록 만드는 것이 최선의 방법이다. 그런데 실제로 뒤에서 실행하는 코드의 흐름을 완전히 따라가기 힘든 경우가 많다. 아래 표2의 프로그래밍 구조에서 상위수준의 프로그램 흐름이 혼란스러워지는 방식을 정리하였다. 여기서의 핵심은 이러한 구조가 차지하는 비율을 높이지 않는 것이다. 이런 구조를 과용하면 코드의 전체 흐름을 파악하는 것이 어려워지기 때문이다.

표 2_실행흐름을 따라가기 어렵게 만드는 프로그래밍구조

프로그래밍 구조	상위수준의 프로그램 흐름이 혼란스러워지는 방식
쓰레딩	어느 코드가 언제 실행되는지 불문명하다.
시그널/인터럽트 핸들러	어떤 코드가 어떤 시점에 실행될지 모른다.
예외처리	예외처리가 여러 함수 호출을 거치면서 실행될 수 있다.
함수포인터 & 익명함수	실행할 함수가 런타임에 결정되기 때문에 컴파일 과정에서는 어떤 코드가 실행될지 알기 어렵다.
가상메서드	object.virtualMethod()는 알려지지 않은 하위클래스의 코드를 호출할지도 모른다.

3.6 오류처리 (Error handling)

코드를 작성하다보면 반드시 에러처리를 다루어야한다. 그런데 흩어져 있는 에러체크의 if 구문들로 실제 코드가 하는 일을 이해하기 어려워지는 경우가 있다. 경험이 풍부한 개발자라면 이러한 방법이 좋지 않다는 것을 알 것이다. 오류코드를 반환하는 함수보다 예외(exception)을 적극적으로 활용하는 것이 바람직하다. 아래 코드 예제와 같이 오류처리를 위하여 중첩된 if문을 남발하여 가독성을 떨어뜨리기보다 try-catch 구문을 통하여 비즈니스 로직과 분리하고 예외처리만 별도의 함수로 구성하였다. 오류처리도 하나의 작업이므로 아래와 같이 오류를 처리하는 함수는 오류만 처리하도록 구성하는 것이 좋다.

```
if (deletePage(page) == E_OK) {
  if (registry.deleteReference(page.name) == E_OK) {
    if (configKeys.deleteKey(page.name.makeKey()) == E_OK){
      logger.log("page deleted");
    } else {
      logger.log("configKey not deleted");
    }
  } else {
    logger.log("deleteReference from registry failed");
  }
} else {
  logger.log("delete failed");
  return E_ERROR;
}
```

```
public void delete(Page page) {
    try {
        deletePageAndAllReferences(page);
    } catch (Exception e) {
        logError(e);
    }
}

private void deletePageAndAllReferences(Page page) throws Exception {
    deletePage(page);
    registry.deleteReference(page.name);
    configKeys.deleteKey(page.name.makeKey());
}

private void logError(Exception e) {
    logger.log(e.getMessage());
}
```

아울러 메서드에서 null을 전달하거나 반환하지 않도록 한다. null은 두 가지를 의미하는데 하나는 정말 없는 것인지 아니면 없다고 표현한 것인지 모호해질 수밖에 없다. 정상적인 인수로 null을 기대하는 API가 아니라면 메서드로 null을 전달하는 코드는 최대한 피해야 한다. 대다수 프로그래밍 언어는 호출자가 실수로 넘기는 null을 적절히 처리하는 방법이 없다. 부득이하게 null을 반환하는 경우라면 아래코드 예제처럼 null체크 코드가 들어가도록 한다. Collections.emptyList()를 통해 미리 정의된 읽기 전용 리스트를 반환하면 코드가 깔끔해질 뿐만 아니라 NullPointerException이 발생할 가능성도 줄어든다.

```
List<Employee> employees = getEmployees();
    for(Employee e : employees) {
        totalPay += e.getPay();
}


public List<Employee> getEmployees() {
    if( .. there are no employees .. )
        return Collections.emptyList();
}
```

Ⅳ. 레거시 코드를 다루기 위한 프랙티스

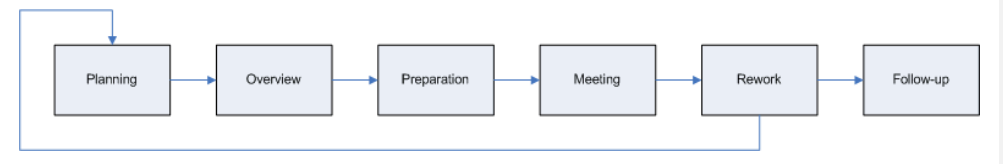
신규 코드는 클린코드의 원칙들에 따라 읽고 이해하기 쉬운 코드를 만들어간다고 하지만 오래된 기존의 레거시 코드들(Legacy code)은 손을 댄다는 것이 쉬운 일이 아니다. 자바 클래스가 코드 라인 수만 몇 천 라인을 넘어가고, 기존 코드들과 타이트하게 얽혀 복잡도가 높아져 있다면 더욱 그럴 것이다.

이런 냄새나는 기존 코드들을 어떻게 읽기 쉬운 깨끗한 코드로 만들 수 있을까? 여러 가지 개선방법 중에서 가장 많이 사용되는 것이 코드리뷰와 리팩토링이다. 먼저 코드리뷰라는 행위 자체가 코드의 가독성을 향상시킨다. 다른 사람에게 코드를 보여주기 위해서 공통된 스타일가이드의 준수하려고 노력하거나 주석 등에 신경을 많이 쓰게 마련이다. 코드리뷰를 위한 몇 가지 방법들 중 인스펙션(inspection)은 정형화된 방법으로 코드 뷰를 위하여 아래 표3과 같이 역할을 정하고 그림 1에 따라서 6가지 단계로 진행된다.

표 3_코드 인스펙션 역할 정의

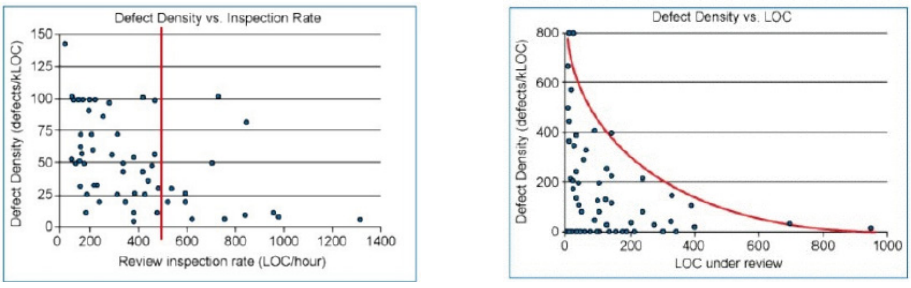
역할	설명
조정자 (Moderrator)	인스펙션을 계획하고 프로세스와 검토할 산출물과 담당자를 지정한다. 검토회의를 주관하고 문제점과 이슈를 관리한다.
발표자 (Reader)	발표할 산출물에 대한 저자 혹은 도메인 전문가로 리뷰 할 산출물을 읽어주고 예상되는 문제점이나 이슈를 제공한다.
검토자 (Reviewer)	코드를 검토하고 의견을 제시하여 수정방안을 함께 토론한다.
기록자 (Recoder)	검토회의에서 발견된 수정사항을 기록하고 결과를 공유하고 이슈를 추적한다.

그림 1_코드 인스펙션 진행 절차



1. Planning : 계획수립
2. Overview : 교육과 역할 정의
3. Preparation : 코드 인스펙션을 위한 인터뷰, 산출물, 도구 준비
4. Meeting : 인스펙션 검토 회의로 각자 역할에 따라 임무 수행
5. Rework : 발견된 결함을 수정하고 재검토가 필요한지 여부를 결정
6. Fellow-up : 보고된 결함 및 이슈 수정되었는지 확인하고 시정조치 이행

그림 2_코드 인스펙션 효과성



출처: http://www.mfagan.com/

일반적으로 코드 리뷰는 코드의 량이 적을수록 리뷰하기가 쉬워진다. 길어도 2시간이내 200정도에서 수행하는 것이 효과적인 것으로 알려져 있다.

일반적으로 코드리뷰는 직관에 의존하는 경우가 많다. 따라서 경험이 적은 개발자들은 코드리뷰를 어렵게 생각 할 수도 있다. 사전에 코드리뷰를 위한 체크리스트를 만들어 활용하면 많은 도움이 된다. 코드리뷰에서 클린코드를 위하여 사용할 체크리스트를 아래와 같이 정리하였다.

- ✓ 코딩 스타일가이드를 준수하고 있는가?
- ✓ 하나의 함수가 하나의 기능만을 수행하고 있는가?
- ✓ 각 함수의 정보는 코드를 설명하기에 충분한가?
- ✓ 주석은 적절하게 기술돼 있는가?
- ✓ 코드는 잘 구조화되어 있는가? (가독성과 기능적 측면)
- ✓ 헤더, 함수 정보를 도구로 추출해서 자동으로 문서화할 수 있는 구조인가?
- ✓ 함수와 변수의 이름은 일관되고 상세하게 기술되어 있는가?
- ✓ 숫자를 직접 서술하지 않고 상수를 사용하고 있는가?
- ✓ 주석처리 된 코드는 삭제되었는가?
- ✓ 설명을 보거나 작성자에게 물어봐야만 이해할 수 있는 코드가 존재하는가?
- ✓ 함수의 길이는 적절한가? (20줄 이내, 한 라인 당 150문자를 넘지 않도록 한다.)
- ✓ 코드의 재사용이 가능한가?
- ✓ 전역변수를 최소화했는가?
- ✓ 변수의 범위는 적절하게 선언되었는가?
- ✓ 클래스와 함수가 관련된 기능끼리 그룹으로 묶여있는가?
- ✓ 중복된 함수나 클래스 혹은 코드가 보이지 않는가?
- ✓ 데이터에 맞게 타입이 구체적으로 선언되었는가?
- ✓ if/else 구문에 2단계 이상 중첩이 있는가? 중첩되었다면 함수로 더 구분했는가?
- ✓ switch/case문이 중첩되었는가? 이를 더 구분할 수 없는가?
- ✓ 리소스에 lock이 있다면 unlock은 이루어졌는가?
- ✓ 스택변수는 반환하고 있는가?
- ✓ 입력 파라미터의 유효범위는 체크하고 있는가?
- ✓ 오류코드를 반환하는 함수보다 예외(exception)을 적극적으로 활용하고 있는가?
- ✓ try/catch 에러 핸들링 사용방법은 적절하게 구현되었는가?
- ✓ 메서드에서 null을 전달하거나 반환하지 않고 있는 않는가?

- ✓ switch문에 default가 존재하며, 예외처리하고 있는가?
- ✓ 배열을 사용할 때, index 범위를 체크하는가?
- ✓ 포인트 사용 시에 유효한 범위를 체크하는가?
- ✓ 가비지 컬레션(garbage collection)은 제대로 수행되고 있는가?
- ✓ 에러 조건이 체크되고 에러 메시지와 에러 코드가 의미를 잘 전달하는가?

코드리뷰를 통하여 냄새나는 코드들이 발견되면 이를 리팩토링을 통하여 제거하도록 한다. 리팩토링은 냄새가 나는 코드에 대하여 점진적으로 반복 수행되는 과정을 통해 코드를 조금씩 개선해 나가게 된다. 리팩토링의 대상 코드는 마틴파울러가 집필한 서적을 참조하여 다음과 같은 몇 개의 카테고리로 구분해 볼 수 있다.

표 4_리팩토링 대상 카테고리

역할	설명
메서드 정리	그룹으로 묶을 수 있는 코드, 수식을 메서드로 변경
객체 간의 기능 이동	메서드 기능에 따른 위치 변경, 클래스 기능의 명확한 구분
데이터 구성	객체지향 캡슐화 기법을 적용해 데이터 접근을 관리
조건문 단순화	조건 논리를 단순하고 명확하게 작성
메서드 호출 단순화	메서드 이름, 목적이 맞지 않는 경우 변경
클래스 및 메서드 일반화	동일 기능 메서드가 여러 클래스에 있으면 수퍼클래스 이동

리팩토링은 아키텍처 관점에서 시작해, 디자인 패턴을 적용하며, 단계적으로 하위 기능에 대한 변경을 진행하는 것이 바람직하다. 또한 리팩토링 과정에서 의도하지 않은 기능의 변경이나 버그의 발생에 대비해 회귀테스트 진행도 필요하다.

리팩토링을 잘하기 위해서는 해당 언어에 대한 특성을 잘 파악하고 있어야 한다. 한 가지 예로 아래 자바 언어에서 문자열을 선언하는 코드를 살펴보자.

(1) String str = new String ("Clean Code")

(2) String str = "Clean Code"

2개 문자열 선언방식에서 (1)번은 자바 클래스를 new를 이용해 메모리에 생성해서 작동하게 만드는 전형적인 자바 문법형 코딩 스타일인 반면 (2)은 자바 언어의 문자열 사용의 일반적인 방식이다. 실제로 사용한다면 (2)번이 (1)번보다 좋은 방식이라고 할 수 있는데, (2)번은 실행할 때마다 새로운 인스턴스를 만들지 않고 하나의 String 객체에 문자열을 넣고 재사용할 수가 있기 때문이다. 소스코드에서는 작은 차이겠지만 문자열 생성 횟수에 따라 메모리 사용량에서 둘은 상당한 차이를 만들어내게 된다.

또한 자바는 클래스 설계 시 캡슐화에 신경을 써야 한다. 캡슐화를 통해 결합도를 낮추고 멤버의 접근성을 최소화할 필요가 있다. 이러한 캡슐화는 private, public, protected의 접근수정자와 패키지를 통해 구현할 수 있다. 클래스 내부 변수는 public 형태로 접근하는 것보다 getValue, setValue의 형태로 접근해 사용할 수 있도록 만들어야 한다.

그림 3_클래스 멤버의 접근성 최소화 예제

class Gem {
  public value;
}
	class Gem {
	  private value;
	  public getValue(){ return value; }
	  public setValue(){ this.value = value; }
	}

Gem obj = new Gem(); obj.value;	Gem obj = new Gem(); obj.getValue();

소프트웨어 개발을 완료하는 과정에서 많은 에러에 직면하게 된다. 이때 발생하는 문제점은 사전 컴파일러가 경고한 메시지와 관련이 있다. 간혹 동작만 하면 된다는 식으로 컴파일 경고 메시지를 무시하는 경우가 있는데, 관련 내용을 확인하고 검색하여 내용을 파악하는 습관을 갖도록 한다. 아울러 문제점을 파악하고 처리하기 위해 예외처리를 사용하도록 한다. 특정 사건이 발생할 때 사용자에게 메시지를 찍어주면, 해당 위치로부터 디버깅의 힌트를 얻어 문제를 해결해나가는데 도움이 된다.

지금까지 클린코드의 개념과 클린코드를 위한 규칙들 그리고 기존의 레거시 코드를 다루기 위한 프랙티스로 코드 리뷰와 리팩토링 방법을 살펴보았다. 클린코드를 만들고 익히는 과정은 개발자의 학습과 더불어 조직에서 코드리뷰와 리팩토링과 같은 프랙티스를 정착시키려는 노력이 필요하다. 이번에는 급하니까 하드코딩을하고 다음에 개선해야지 마음을 먹지만 다음이 오는 경우는 드물다. 유명한 “깨진 유리창의 법칙”과 같이 사소한 것들을 방치하면 결국에는 손을 대기조차 어려운 상황으로 악화되기 쉽다.

클린코드의 규칙을 지켜 새로운 코드를 만들고, 작성한 코드를 동료들과 검토한 뒤에 커밋을 하고 냄새나는 코드들을 리팩토링을 통해 지속적으로 개선하는 것만이 클린코드를 만들어 내는 방법이다.

실제로 수작업으로 리팩토링을 진행하는 것은 매우 어려운 작업이다. 메서드의 이름을 변경하는 경우 해당 메서드를 일일이 찾아 몇 백 개를 수작업으로 처리할 수는 없다. 자바 무료 개발도구인 이클립스 도구를 활용한다면 보다 손쉽게 이런 작업들을 수행할 수 있다. 클래스, 메서드, 변수, 파라미터 등의 이름 변경, 익명 클래스를 중첩 클래스로 변경처리, 상위 클래스의 값을 하위 클래스로 옮기기 등 다양한 기능들을 제공

하고 있다. 리팩토링의 다양한 기법과 이클립스의 리팩토링 기능에 대한 상세한 내용은 아래 참고자료에서 4번과 3번 자료를 각각 참조하기 바란다.

참고 자료

1. Code Simplicity, Max Kanal-Alexander, O'Reilly, 2012
2. Clean Code 클린코드: 애자일 소프트웨어 장인정신, 로버트마틴(저)/박재호.이해영(공역), 인사이트, 2013
3. Eclips refactoring tutorial, http://www.ibm.com/developerworks/library/os-ecref/
4. Java Refactoring, http://www.refactoring.com/catalog/index.html
5. The Art of Readable Code: Simple and Practical Techniques for Writing Better Code, Dustin Boswell, Trevor Foucher, O'Reilly, 2012
6. 리팩토링 : 코드 품질을 개선하는 객체지향 사고법, 마틴파울러(저)/김지원(역), 한빛미디어, 2012
7. 소프트웨어 프로세스 이야기, http://swprocess.egloos.com/