|  |  |
| --- | --- |
| Refactoring Guide to reduce Cyclomatic complexity for Java | |
|  |  |

|  |
| --- |
| 목차 |

[1. Complexity metrics 5](#_Toc223879694)

[1.1 개념 5](#_Toc223879695)

[1.1.1 Cyclomatic complexity with Booleans 5](#_Toc223879696)

[1.1.2 CC2 계산 예제 6](#_Toc223879697)

[2. CC2 감소 방법 9](#_Toc223879698)

[2.1 개요 9](#_Toc223879699)

[2.2 Functions (Methods) 분리 및 중복 제거 10](#_Toc223879700)

[2.3 조건연산 단순화 15](#_Toc223879701)

[2.3.1 조건 연산 분리 15](#_Toc223879702)

[2.3.2 조건문의 통합 분리 16](#_Toc223879703)

[2.3.3 Control Flag의 제거 17](#_Toc223879704)

[2.3.4 다형성을 이용한 switch 사용의 최소화 18](#_Toc223879705)

[2.3.5 NULL Object 20](#_Toc223879706)

[2.4 클래스의 분리 및 중복 제거 23](#_Toc223879707)

[3. 요약 25](#_Toc223879708)

# Complexity metrics

## 개념

Cyclomatic complexity 또는 Conditional Complexity (CC)는 1976년 Thomas J. McCabe에 의해 고안 되었으며 소프트웨어 코드의 복잡도를 측정 하는 Metric 이다. CC는 측정 방법에 따라 Cyclomatic complexity with Booleans (Extended or strict cyclomatic complexity), Cyclomatic complexity without Cases (Modified cyclomatic complexity) 등이 존재 한다.

NHN에서는 Java, C/C++에 대한 Complexity 측정 도구로 N’SIQ Collector를 표준으로 사용 하는 것을 권장 하고 있으며 N’SIQ Collector의 경우 Cyclomatic complexity with Booleans를 측정 하고 있어 본 문서에서의 Cyclomatic Complexity는 Cyclomatic complexity with Booleans를 의미한다.

Cyclomatic Complexity의 값이 높으면 코드가 복잡하게 작성 된 것을 의미 한다. 코드의 복잡도가 높으면 개발자가 이해 하기 어려우며 개발 및 테스트에 더 많은 시간과 자원이 소요 된다. 그러므로 코드의 복잡도를 낮추는 것이 생산성 및 유지보수성을 높이고 오류를 피하기 위한 좋은 방법이며 본 문서에서는 Cyclomatic Complexity를 낮추기 위한 Refactoring 방법을 예제 중심으로 설명 하였다.

### Cyclomatic complexity with Booleans

Cyclomatic complexity는 소프트웨어 엔지니어링에서 가장 보편적으로 사용 되는 복잡도에 대한 Metric이다. 여기에 Boolean에 대한 복잡도를 같이 계산하는 것이 Cyclomatic complexity with Booleans이며 다음의 공식과 같다.

위의 식에서 “*Decision*”은 실행문 (*if, elseif, switch, while,등*)을 의미 하며 Boolean은 조건연산자 이다 . CC2의 최소값은 1이며 Decision의 수와 조건연산자의 수에 따라 얼마든지 큰 값을 가질 수 있다. 일반적인 연산 방법은 표 1‑1과 같다. (상세 측정 방법은 도구에 따라 달라 질 수 있다.)

|  |
| --- |
| Cyclomatic Complexity 요약 (*측정값의 크기: CC2 > CC > CC3)[[1]](#footnote-1)*  Cyclomatic complexity (CC): *CC = Decision 수의 합 + 1* (case문 하나당 Complexity가 1씩 증가)  Cyclomatic complexity with Booleans (CC2): *CC2 = Decision 수의 합 + Boolean 연산의 합 + 1* (case문 하나당 Complexity가 1씩 증가)  Cyclomatic complexity without Cases (CC3): *CC3 = Decision 수의 합 + 1* (전체 case문에 Complexity 1씩 증가) |

표 1‑1 Construct에 따른 CC2 계산 방법

|  |  |  |
| --- | --- | --- |
| Construct | CC2 값의 변경 | 설명 |
| *if* | +1 | if 문 하나에 1씩 증가 |
| *else if* | +1 | else if문 하나에 1씩 증가 |
| *else* | 0 | else문은 Complexity에서 제외 |
| *switch case* | case 하나 당 +1 | case문 하나에 1씩 증가 |
| *switch default* | 0 | default는 Complexity에서 제외 |
| *for* | +1 | for문 하나에 1씩 증가 |
| *while* | +1 | while문 하나에 1씩 증가 |
| *do while* | +1 | do while 하나에 1씩 증가 |
| *try catch* | +1 | catch 하나에 1씩 증가 |
| *try finally* | 0 | finally는 Complexity에서 제외 |
| *Booleans* | Boolean 연산 하나 당 +1 | 조건 연산자 (&&, ||, ?) 하나에 1씩 증가 |

### CC2 계산 예제

아래의 Java 코드에 대한 CC2의 측정 예제 이다. 각 예제 중에 ***이탤릭 굵은 글씨 밑줄***로 쓰여진 부분이 CC2 값이 증가 하는 부분이다. CC2는 1을 최소값으로 갖는다.

#### CC2: 1

public void cc1() {

System.*out*.println("CC is one");

}

#### CC2: 2

public void cc2(boolean bPrint) {

***if*** (bPrint) {

System.*out*.println("if");

}

else {

System.*out*.println("if else");

}

System.*out*.println("CC is two");

}

#### CC2: 3

public void cc3(boolean bFirst, boolean bSecond) {

***if*** (bFirst ***&&*** bSecond) {

System.*out*.println("if");

}

else{

System.*out*.println("if else");

}

System.*out*.println("CC is three");

}

#### CC2: 4

public void cc4(int select) {

switch (select) {

***case*** 1:

System.*out*.println("case 1");

break;

***case*** 2:

System.*out*.println("case 2");

break;

***case*** 3:

System.*out*.println("case 3");

break;

default:

System.*out*.println("default");

}

System.*out*.println("CC is four");

}

#### CC2: 4

public void ccForwithBoolean() {

boolean bContinue = true;

***for*** (int i=0 ; i < 10 ***&&*** bContinue ; ++i) {

***if*** (i == 5) {

bContinue = false;

}

System.*out*.println(i);

}

}

#### CC2: 3

public void ccWhile() {

int i = 0;

***while*** (true) {

***if*** (i >= 10) {

break;

}

else {

++i;

}

}

}

#### CC2: 4

public void ccTryCatch() {

int denominatorArray[] = {5,4,3,2,1,0};

int quotient;

try {

***for*** (int i=0 ; i < 10 ; ++i){

quotient = 10 / denominatorArray[i];

System.*out*.println(quotient);

}

} ***catch*** (ArithmeticException e) {

} ***catch*** (ArrayIndexOutOfBoundsException e) {

} finally {

}

}

#### CC2: 5

public void ccBooleans(boolean bCondition) {

boolean bCC = bCondition ***?*** true : false;

boolean bNotCC = true & false;

bCC = true ***&&*** false;

bCC = true ***||*** false ***&&*** true;

bNotCC = true == true;

bNotCC = true != true;

System.*out*.println("Booleans");

}

# CC2 감소 방법

## 개요

투자의 정석 중 하나는 위험의 분산이라고 이야기 한다. 이와 유사 하게 코드의 복잡도 감소를 위한 첩경도 모여 있는 Complexity를 여러 곳으로 분산 시키는 것이다. 아래 그림에서와 같이 하나의 서랍 안에 복잡하게 들어가 있는 서류를 여러 개의 서랍에 종류에 따라 분리해서 정리 하는 것을 생각 해 보자. 그림의 왼쪽과 같이 하나의 서랍에 여러 문서가 뒤 섞여 있는 경우가 복잡도가 높은 예이다. 물론 사용자는 그런 서류 더미 속에서도 자신이 원하는 자료를 찾을 수 있을 것이다. 다만 찾는 것이 쉽지 않은 과정이 될 것이며 재수가 없는 경우라면 원하는 서류가 다른 서류 사이에 있어 발견 하지 못하는 경우가 발생 할 수 있다. 이러한 서류 더미를 여러 개의 서랍에 색인 별로 나누어 분류 하면 원하는 서류를 쉽게 찾을 수 있게 된다. 그리고 서류 정리 작업도 한결 수월 하게 진행 할 수 있을 것이다. 이러한 개념이 코드의 복잡도를 낮추는 것과 동일 하다. 예를 들어 하나의 Method에 대한 복잡도의 측정 결과가 30인 경우 해당 Method는 굉장히 복잡도가 높아 유지 보수가 어렵고 오류가 발생할 확률이 높은 상태라 이야기 할 수 있다. 이런 복잡한 Method를 10 이하의 복잡도를 갖는 4개의 Method들로 분리 하면 이해 하기 쉬우며 유지보수가 용이해 진다.

복잡도를 감소 시키는 다른 방법은 코드의 중복을 제거 하는 것이다. 중복된 코드는 복잡도를 증가 시킬 뿐만 아니라 유지 보수를 어렵게 만드는 주요 요인이므로 반드시 관리 되어야 한다. 이 외 조건 연산 부분을 관리 함으로써 코드의 복잡도를 줄일 수 있다.

Refactoring

본 장에서는 CC2를 감소 시키는 효과적인 방법과 고려 사항에 대하여 예제 중심으로 설명 하고 있다. 또한 실제 개발 환경에서처럼 N’SIQ Collector를 이용 하여 복잡도를 측정하고 Code Refactoring을 수행 하는 과정을 반복 하여 복잡도를 줄이는 방식을 취하였다. 해당 내용을 정독 하는 것으로도 효과를 볼 수 있지만 가장 효과적인 방법은 연습이므로 단순한 예라도 꼭 따라 해보길 추천 한다.

## Functions (Methods) 분리 및 중복 제거

함수 또는 Method의 Complexity가 높은 경우를 살펴보면 통상적으로 함수 내에서 여러 기능을 수행 하고 있거나 중복된 코드에 의한 경우가 많다. 함수는 해당 기능에 대한 정확한 이름을 정의 하고 단 하나의 동작만을 하게 만드는 것이 Code의 Readability를 높이고 Complexity를 낮출 수 있는 방법이다. 또한 함수 내부의 코드 중복을 제거 하면 유지 보수성을 높일 수 있으며 코드의 Complexity를 낮출 수 있다.

다음의 코드는 “Setup, Exercise, Verify, Teardown”와 테스트 종류에 맞추어 알맞은 문자열을 생성 하여 리턴 하는 예제이다. N’SIQ Collector로 측정 된 CC2는 17이며 여러 중복 코드와 조건문에 의한 높은 복잡도를 가지고 있다.

List 2‑1 CC2의 값이 17인 ccComplexMethod

|  |
| --- |
| **public** String ccComplexMethod(String strTest, **int** testType) {  String strDecoration = **new** String("");  //Setup  **if** (strTest.equals("SetUp")) {  strDecoration += "SetUp";  **if** (0 == testType) {  strDecoration += "Test Type is 0.";  } **else** **if** (1 == testType) {  strDecoration += "Test Type is 1.";  } **else** **if** ( 2 == testType) {  strDecoration += "Test Type is 2.";  } **else** {  strDecoration += "Not defined Test Type.";  }  }    //Exercise  **if** (strTest.equals("Exercise")) {  strDecoration += "Exercise";  **if** (0 == testType) {  strDecoration += "Test Type is 0.";  } **else** **if** (1 == testType) {  strDecoration += "Test Type is 1.";  } **else** **if** ( 2 == testType) {  strDecoration += "Test Type is 2.";  } **else** {  strDecoration += "Not defined Test Type.";  }  }  //Verify  **if** (strTest.equals("Verify")) {  strDecoration += "Verify";  **if** (0 == testType) {  strDecoration += "Test Type is 0.";  } **else** **if** (1 == testType) {  strDecoration += "Test Type is 1.";  } **else** **if** ( 2 == testType) {  strDecoration += "Test Type is 2.";  } **else** {  strDecoration += "Not defined Test Type.";  }  }  //Teardown  **if** (strTest.equals("Teardown")) {  strDecoration += "Teardown";  **if** (0 == testType) {  strDecoration += "Test Type is 0.";  } **else** **if** (1 == testType) {  strDecoration += "Test Type is 1.";  } **else** **if** ( 2 == testType) {  strDecoration += "Test Type is 2.";  } **else** {  strDecoration += "Not defined Test Type.";  }  }    **return** strDecoration;  } |

이해를 돕기 위하여 단순 하게 작성 되었지만 실제 코드에서도 이와 같이 여러 기능이 하나의 함수에서 수행 되는 경우를 쉽게 찾아 볼 수 있다. 이때 CC2를 낮추는 방법이 각 기능별로 함수를 분리 하는 것이며 다음은 기능 별로 함수를 분리한 결과 이다.

List 2‑2 Refactoring 후 ccComplexMethod는 CC2의 값이 1 이며 다른 함수들은 5의 값을 갖는 예

|  |
| --- |
| **public** String ccComplexMethod(String strTest, **int** testType) {  String strDecoration = **new** String("");  strDecoration = ccSetUpString(strTest, testType);  strDecoration = ccExerciseString(strTest, testType);  strDecoration = ccVerifyString(strTest, testType);  strDecoration = ccTeardownString(strTest, testType);    **return** strDecoration;  }  **private** String ccTeardownString(String strTest, **int** testType) {  String strDecoration = **new** String("");  //Teardown  **if** (strTest.equals("Teardown")) {  strDecoration += "Teardown";  **if** (0 == testType) {  strDecoration += "Test Type is 0.";  } **else** **if** (1 == testType) {  strDecoration += "Test Type is 1.";  } **else** **if** ( 2 == testType) {  strDecoration += "Test Type is 2.";  } **else** {  strDecoration += "Not defined Test Type.";  }  }  **return** strDecoration;  }  **private** String ccVerifyString(String strTest, **int** testType) {  String strDecoration = **new** String("");  //Verify  **if** (strTest.equals("Verify")) {  strDecoration += "Verify";  **if** (0 == testType) {  strDecoration += "Test Type is 0.";  } **else** **if** (1 == testType) {  strDecoration += "Test Type is 1.";  } **else** **if** ( 2 == testType) {  strDecoration += "Test Type is 2.";  } **else** {  strDecoration += "Not defined Test Type.";  }  }  **return** strDecoration;  }  **private** String ccExerciseString(String strTest, **int** testType) {  String strDecoration = **new** String("");  //Exercise  **if** (strTest.equals("Exercise")) {  strDecoration += "Exercise";  **if** (0 == testType) {  strDecoration += "Test Type is 0.";  } **else** **if** (1 == testType) {  strDecoration += "Test Type is 1.";  } **else** **if** ( 2 == testType) {  strDecoration += "Test Type is 2.";  } **else** {  strDecoration += "Not defined Test Type.";  }  }  **return** strDecoration;  }  **private** String ccSetUpString(String strTest, **int** testType) {  String strDecoration = **new** String("");  //Setup  **if** (strTest.equals("SetUp")) {  strDecoration += "SetUp";  **if** (0 == testType) {  strDecoration += "Test Type is 0.";  } **else** **if** (1 == testType) {  strDecoration += "Test Type is 1.";  } **else** **if** ( 2 == testType) {  strDecoration += "Test Type is 2.";  } **else** {  strDecoration += "Not defined Test Type.";  }  }  **return** strDecoration;  } |

ccComplexMethod 함수는 내부 기능에 따라 세분화 되었으며 CC2의 값은 1로 낮추어져 기준값을 충족 할 수 있게 되었다. 이러한 Refactoring을 통해서 얻을 수 있는 장점은 CC2의 감소뿐만 아니라 기능 별로 함수가 세분화 되어 유지 보수가 용이해 지는 이점과 함께 분리 된 기능이 재사용될 수 있다는 것이다.

그러나 이러한 방법은 각 함수의 복잡도는 낮출 수 있으나 class의 단위의 복잡도는 오히려 증가 하는 문제점이 있다. class의 크기가 문제가 되지 않는 수준이라면 고려의 대상이 아니지만 Super Class 또는 God Class[[2]](#footnote-2)의 형태를 띠고 있다면 심각한 문제이다. N’SIQ Collector의 경우 함수 단위로 복잡도를 측정 하고 있어 클래스 단위의 복잡도는 관리의 대상이 아니지만 클래스 안의 복잡도가 증가 하는 것은 해당 클래스의 유지 보수가 어려워 지고 있다는 증거이므로 주의 하여야 한다. 이러한 부분은 클래스의 분리를 통하여 해결 할 수 있으며 2.4 절에서 보다 자세히 다루도록 하겠다.

그 외List 2-2의 경우 함수의 Parameter가 증가 했다는 문제점이 있다. 함수의 Parameter가 증가 하게 되면 기능을 이해 하기 어렵게 되어 사용에 문제점이 발생 한다. (Parameter가 없는 함수와 1개인 함수, 2개인 함수를 호출 할 경우 사용의 편리성을 생각해 보라) 예제의 경우는 Parameter를 넘기고 Return을 받는 구조로 되어 있어 더욱 사용 하기 어려운 형태가 되었으며 해당 기능에 대한 검증에 더 많은 노력이 필요 하게 되었다. 그러므로 함수를 분리 할 경우 Parameter가 임시 변수 여야 하는지 아니면 멤버 변수로 전환 하는 것이 효과적인지 반드시 고려해 보아야 한다. 예제의 경우 Parameter들이 멤버 변수로 정의 될 경우 이해 하기 쉽고 유지 보수가 편리해 질 수 있다.

그러나 무엇 보다 List 2-2의 심각한 문제는 코드의 중복이다. 코드의 중복은 복잡도가 유지 보수 과정에 다시 증가 할 수 있음을 의미 한다. 그러므로 중복 코드의 제거는 반드시 수행 되어야 하며 이를 반영한 Refactoring 최종 결과는 다음과 같다.

List 2‑3 Refactoring을 통해 Parameter를 조정한 ccComplexMethod의 예 (complexity 증가 없이 Readability 증가)

|  |
| --- |
| String strTest;  **int** testType;  **public** String ccComplexMethod() {  String strDecoration = **new** String("");    strDecoration = ccSetUpString();  strDecoration = ccExerciseString();  strDecoration = ccVerifyString();  strDecoration = ccTeardownString();    **return** strDecoration;  }  **private** String ccStringDecorationByTestStep(String testStep) {  String strDecoration = **new** String("");    **if** (strTest.equals(testStep)) {  strDecoration += testStep;  **if** (0 == testType) {  strDecoration += "Test Type is 0.";  } **else** **if** (1 == testType) {  strDecoration += "Test Type is 1.";  } **else** **if** ( 2 == testType) {  strDecoration += "Test Type is 2.";  } **else** {  strDecoration += "Not defined Test Type.";  }  }    **return** strDecoration;  }  **private** String ccTeardownString() {  **return** ccStringDecorationByTestStep("Teardown");  }  **private** String ccVerifyString() {  **return** ccStringDecorationByTestStep("Verify");  }  **private** String ccExerciseString() {  **return** ccStringDecorationByTestStep("Exercise");  }  **private** String ccSetUpString() {  **return** ccStringDecorationByTestStep("Setup");  } |

List 2-3은 List 2-2에 비하여 Readability가 증가 하였으며 중복 코드 제거를 통하여 복잡도가 낮고 유지 보수가 용이한 코드로 재 탄생 되었다.

List 2-3에서의 문제점으로 남아 있는 부분은 List 2-2는 strTest 문자열에 따라 하나의 기능만 수행 되나 List 2-3에는 그러한 목적이 코드에 명확하게 나타나지 않는 것이다. 실제 수행 결과는 동일 하지만 ccComplexMethod는 내부에 4개의 함수 호출이 모두 수행 되는 것으로 오해 될 수 있다. 그리고 아직도 유사 형태의 코드 호출이 여러 곳에서 발생 하므로 코드의 중복성 문제도 완벽하게 해결 된 상태가 아니다. 이는 코드의 Readability향상을 위한 Refactoring 수행이 필요함을 의미 한다. List 2-4는 최종 Refactoring 결과이다. 불필요한 함수들이 모두 삭제 되었으며 코드도 초기의 명확한 의미를 나타내도록 변경 되었다.

List 2‑4 Readability 향상을 위한 Refactoring

|  |
| --- |
| String strTest;  **int** testType;  **public** String ccComplexMethod() {  String strDecoration = **new** String("");    strDecoration += strTest;  **if** (0 == testType) {  strDecoration += "Test Type is 0.";  } **else** **if** (1 == testType) {  strDecoration += "Test Type is 1.";  } **else** **if** ( 2 == testType) {  strDecoration += "Test Type is 2.";  } **else** {  strDecoration += "Not defined Test Type.";  }    **return** strDecoration;  } |

## 조건연산 단순화[[3]](#footnote-3)

Boolean 연산자가 증가 하면 Complexity도 높아지게 된다. 그러므로 CC2를 감소 시키기 위해서는 조건연산을 단순화 하는 것이 필수적이다.

본 절에서는 조건연산을 단순화 하여 CC2를 감소 시키는 Refactoring 기법을 소개 한다.

### 조건 연산 분리

List 254는 기간에 따라 호텔 숙박 비용을 계산 하는 함수이다. 숙박 비용은 여름 철 일 때와 아닐 때로 구분되어 계산 되고 있다. 일반적으로 예제 정도의 복잡도가 문제가 되지는 않지만 호텔 숙박 비용을 나누는 기준이 여름, 가을, 겨울, 성수기, 비수기 등으로 다양해지면 CC2가 급격 하게 증가 할 수 있다.

경험적으로 if문 안의 조건문은 유지 보수가 진행 됨에 따라 복잡도가 증가 하는 경향이 있으며 이는 코드의 Readability를 저하 시키고 테스트 수행을 어렵게 만든다. 이러한 문제점을 해결 하는 방안이 조건 연산 분리이다. List 2-6는 if문의 조건 연산을 별도의 함수로 분리 한 Refactoring 결과 이다. List 2-5의 CC2 값은 3이며 Refactoring을 통하여 2로 감소 되었다.

List 2‑5 조건 연산의 예

|  |
| --- |
| **public** **double** getSeasonCharge(Date date, **int** quantity) {  **double** charge = 0;    **if** (date.before (*SUMMER\_START*) || date.after(*SUMMER\_END*)) {  charge = quantity \* *\_winterRate* + *\_winterServiceCharge*;  } **else** {  charge = quantity \* *\_summerRate*;  }  **return** charge;  } |

List 2‑6 조건 연산의 Refactoring의 예

|  |
| --- |
| **public** **double** getSeasonCharge(Date date, **int** quantity) {  **double** charge = 0;    **if** (notSummer(date)) {  charge = winterCharge(quantity);  } **else** {  charge = summerCharge (quantity);  }    **return** charge;  }  **private** **boolean** notSummer(Date date) {  **return** date.before (*SUMMER\_START*) || date.after(*SUMMER\_END*);  }  **private** **double** summerCharge(**int** quantity) {  **return** quantity \* *\_summerRate*;  }  **private** **double** winterCharge(**int** quantity) {  **return** quantity \* *\_winterRate* + *\_winterServiceCharge*;  } |

List 2-6의 경우 CC2의 감소뿐만 아니라 조건 연산을 함수로 분리 함으로써 보다 명확한 의미를 전달 할 수 있게 되었다. List 254의 경우 개발자는 해당 조건문이 의미 하는 것을 파악 하기 어렵다 그러나 List 2-6의 경우 해당 함수명이 notSummer로 명확하게 조건연산의 의미를 알려 주고 있다.

예제에서의 복잡도 1감소가 무시할 만한 수준으로 생각 되어 Refactoring이 의미가 없는 것으로 오해 될 수 있다. 만일 그러한 의심이 드는 독자라면 다음의 경우를 가정해 보자. “여름 요금에 대한 내용에 추가하여 겨울 요금, 주말 요금 그리고 명절 기간에 대한 요금이 추가 되었다.” 요구 사항 변경에 따라 겨울 기간에 의한 복잡도가 최소 2 증가 하게 될 것이며 주말 여부를 확인 하기 위한 조건도 추가 될 것이다. 거기에 더해 명절에 따른 복잡도와 명절과 주말이 겹칠 경우를 예상 해 보면 코드의 복잡도는 짧은 시간 내에 제어 가능한 범위를 벗어나게 된다. 이러한 요구 사항 변경에도 효과적으로 복잡도를 조절할 수 있는 방법이 조건문의 분리이다.

### 조건문의 통합 분리

일반적으로 함수의 초기 부분에는 변수의 선언과 함께 함수 내부에서 쓰이는 변수들에 대한 검증 부분이 존재 하게 된다. 그러한 부분을 하나의 함수로 묶어서 분리 하면 CC2를 낮출 수 있으며 코드의 Readability가 증가 한다.

List 2-7은 근로자의 장애 등급을 판정 하는 예제이다. 함수의 앞 부분에서 변수들에 대한 검증을 수행 하며 조건에 맞지 않을 경우 0을 리턴 하도록 되어 있다.

List 2‑7 OR 조건문 통합 전

|  |
| --- |
| **double** disabilityAmount() {  **double** amount = 0.0;  **if** (\_seniority < 2) **return** 0;  **if** (\_monthsDisabled > 12) **return** 0;  **if** (\_isPartTime) **return** 0;  // compute the disability amount    **return** amount;  } |

List 2-8은 조건문을 통합 하여 함수로 분리한 Refactoring 결과 이다.

List 2‑8 OR 조건문 통합 후

|  |
| --- |
| **double** disabilityAmount() {  **double** amount = 0.0;  **if** (isNotEligableForDisability()) {  **return** 0;  }  // compute the disability amount    **return** amount;  }  **private** **boolean** isNotEligableForDisability() {  **return** ((\_seniority < 2) || (\_monthsDisabled > 12) || (\_isPartTime));  } |

List 2-8에서와 같이 조건문을 통합하고 함수로 분리 해 냄으로써 Readability가 증가 하였으며 disabilityAmount의 CC2값은 감소 하게 되었다.

List 2-9은 AND 조건에 대한 조건문 통합 분리 예제이다.

List 2‑9 AND 조건문 통합 전

|  |
| --- |
| **public** **int** ccAndOperation(**boolean** firstCondition, **boolean** secondCondition) {  **if** (firstCondition) {  **if** (secondCondition) {  **return** 1;  }  }    **return** 0;  } |

List 2-10는 AND 조건문을 통합하여 함수로 분리 한 결과이다. 향후 조건이 추가 되는 경우에도 ccAndOperation의 CC2는 증가 하지 않으며 유지 보수가 용이 하게 된다.

List 2‑10 AND 조건문 통합 후

|  |
| --- |
| **public** **int** ccAndOperation(**boolean** firstCondition, **boolean** secondCondition) {  **if** (isAllConditionMet(firstCondition, secondCondition)) {  **return** 1;  }    **return** 0;  }  **private** **boolean** isAllConditionMet(**boolean** firstCondition,  **boolean** secondCondition){  **return** firstCondition && secondCondition;  } |

### Control Flag의 제거

Control Flag은 Loop 내부에서 특정 조건을 만족 하였을 경우 Loop를 빠져 나오게 하거나 또는 계속 수행 하게 하는 조건을 의미 하며 이는 CC2를 증가 시킬 뿐만 아니라 코드의 내용을 이해 하기 어렵게 만든다.

List 2-11은 Control Flag를 가진 예제 이다.

List 2‑11 Control Flag 예제

|  |
| --- |
| **void** checkSecurity(String[] people) {  String found = "";  **for** (**int** i = 0; i < people.length; i++) {  **if** (found.equals("")) {  **if** (people[i].equals ("Don")){  sendAlert();  found = "Don";  }  **if** (people[i].equals ("John")){  sendAlert();  found = "John";  }  }  }  someLaterCode(found);  } |

List 2-11에서 found는 결과 값인 동시에 Control Flag로 동작 하고 있다. 예제에서는 Control Flag로 인하여 코드의 복잡도가 증가 하였다. 해당 코드에서 for문을 별도의 함수로 분리 하고 Control Flag를 제거 함으로써 CC2의 값을 감소 시킬 수 있으며 결과는 List 2-12과 같다.

List 2‑12 Control Flag 제거

|  |
| --- |
| **void** checkSecurity(String[] people) {  **String** found = foundMiscreant(people);  someLaterCode(found);  }  String foundMiscreant(String[] people){  **for** (**int** i = 0; i < people.length; i++) {  **if** (people[i].equals ("Don")){  sendAlert();  **return** "Don";  }  **if** (people[i].equals ("John")){  sendAlert();  **return** "John";  }  }  **return** "";  } |

### 다형성을 이용한 switch 사용의 최소화

switch문의 case는 쉽게 증가 하는 경향을 갖는다. List 2-12는 직원에 대한 class이다. 직원은 역할 (엔지니어, 매니저, 영업)에 따라 각기 다른 급여를 받는 구조로 되어 있다. Employee 클래스는 향후 역할이 추가 되면 그에 따른 case가 추가 되어야 하며 그 결과로 CC2가 증가 하게 된다. 역할 별로 처리 하는 부분이 봉급에 제한 된다면 큰 문제가 생기지 않을 수도 있다. 그러나 역할 별로 휴가 산정이 다르고 지급되는 보너스가 다를 경우 모든 처리 함수 마다 동일한 형태의 switch문이 존재 하게 되어 역할이 추가 될 때 관련된 모든 부분에 변경이 이루어져야 하는 문제점이 발생 하게 된다. 또한 이에 따라 CC2도 증가 하게 된다.

List 2‑13 근무 타입에 따른 switch문의 예

|  |
| --- |
| **public** **class** Employee {  **private** **static** **final** **int** *ENGINEER* = 0;  **private** **static** **final** **int** *SALESMAN* = 1;  **private** **static** **final** **int** *MANAGER* = 2;  **private** **int** type;  **private** **int** \_monthlySalary;  **private** **int** \_commission;  **private** **int** \_bonus;  **public** **int** payAmount() {  **switch** (type) {  **case** *ENGINEER*:  **return** \_monthlySalary;  **case** *SALESMAN*:  **return** \_monthlySalary + \_commission;  **case** *MANAGER*:  **return** \_monthlySalary + \_bonus;  **default**:  **throw** **new** RuntimeException("Incorrect Employee");  }  }    **public** **int** getEmployeeType() {  **return** type;  }  } |

List 2-13의 문제는 다형성 (Polymorphism)을 이용 하여 해결 할 수 있다. 다형성과 Factory 패턴을 이용하면 신규 역할 추가에 따른 switch문의 추가가 발생 하지 않으며 switch문을 생성 과정에 한정 되어 사용 할 수 있게 된다. List 2-14은 Refactoring 결과 이다.

List 2‑14 다형성을 이용한 Refactoring 예제

|  |
| --- |
| **public** **abstract** **class** Employee {  **public** **abstract** **int** payAmount();  }  **public** **interface** EmployeeFactory {  **public** Employee makeEmployee(EmployeeRecord r) **throws** InvalidEmployeddType;  }  **public** **class** EmployeeFactoryImpl **implements** EmployeeFactory {  **private** **static** **final** **int** *ENGINEER* = 0;  **private** **static** **final** **int** *SALESMAN* = 1;  **private** **static** **final** **int** *MANAGER* = 3;  **public** Employee makeEmployee(EmployeeRecord r) **throws** InvalidEmployeddType {  **switch**(r.type) {  **case** *ENGINEER*:  **return** **new** Engineer(r);  **case** *SALESMAN*:  **return** **new** Salesman(r);  **case** *MANAGER*:  **return** **new** Manager(r);  **default**:  **throw** **new** InvalidEmployeddType(r.type);  }  }  }  **public** **class** Engineer **extends** Employee {  **public** Engineer(EmployeeRecord r) {  }  **public** **int** payAmount() {  **return** \_monthlySalary;  }  } |

List 2-14에서는 다형성을 이용하여 Refactoring을 수행 하였다. switch문은 EmployeeFactoryImpl 클래스 내부에서만 한정 되어 사용된다. 이를 통하여 여러 함수에서 반복적으로 나타나게 되는 switch문을 효과적으로 제어 할 수 있으며 CC2를 감소 시킬 수 있다. 또한 추가적인 직원 역할이 필요할 경우에는 EmployeeFactoryImpl내부의 makeEmployee만 변경 하면 되므로 코드의 유지 보수성이 향상 된다.

### NULL Object

대부분의 코드에서는 함수 호출을 통해 얻어오는 객체가 NULL인지 여부를 검사 하는 코드를 가지고 있다. NULL을 리턴 하는 함수에 대하여 검사를 수행 하지 않을 경우 심각한 오류가 발생 되므로 NULL 검사는 필수적인 요소 이다. 그러나 NULL에 대한 검사로 인하여 코드의 복잡도가 증가 하는 문제가 발생 하게 된다. 함수의 Parameter에 NULL을 전달 하는 경우도 코드의 복잡도를 증가 시킨다. 만일 Parameter에 NULL 값이 전달 될 수 있다면 함수는 수행 전 반드시 Parameter에 대한 검사를 수행 하여야 한다. List 2-15에서는 Customer의 인스턴스를 얻기 위하여 getCustomer를 호출 하고 있다. 해당 함수는 NULL을 리턴 할 수 있으므로 customer를 사용 하기 전에는 반드시 NULL 검사를 수행 하여야 한다. 그러므로 getCustomer에 대한 호출이 증가 하게 되면 CC2도 따라서 증가 하게 되며 개발자 실수에 의한 심각한 오류 발생 확률도 높아진다.

List 2‑15 함수 호출에 따른 NULL확인의 예

|  |
| --- |
| **class** Site {  Customer \_customer;  Customer getCustomer() {  **return** \_customer;  }  }  **class** Customer {  **public** String getName() {...}  **public** BillingPlan getPlan() {...}  **public** PaymentHistory getHistory() {...}  }  **public** **class** PaymentHistory {  **public** **int** getWeeksDelinquentInLastYear() {...}  }  Customer customer = site.getCustomer();  BillingPlan plan;  **if** (customer == **null**) plan = BillingPlan.basic();  **else** plan = customer.getPlan();  ...  String customerName;  **if** (customer == **null**) customerName = "occupant";  **else** customerName = customer.getName();  ...  **int** weeksDelinquent;  **if** (customer == **null**) weeksDelinquent = 0;  **else** weeksDelinquent = customer.getHistory().getWeeksDelinquentInLastYear(); |

이러한 문제점을 해결 하기 위한 방법으로 NULL Object가 사용 될 수 있다. NULL Object는 사전에 정의된 NULL의 동작에 대한 속성을 갖는 Object를 의미 한다. 즉, 모호한 NULL을 전달 하는 것이 아닌 사전에 정의된 동작을 하는 Object를 전달 하는 것이 NULL Object이다.[[4]](#footnote-4)

List 2-16는 NULL Object를 이용하여 List 2-15를 Refactoring한 결과 이다. List 2-15에 비교 하여 Customer를 상속 받는 NullCustomer와 PaymentHistory를 상속 받는 NullPaymentHistory가 신규 클래스로 생성 되었다.

Site의 getCustomer 함수에서는 멤버 변수인 \_customer의 상태를 무조건 리턴 하는 것에서 NULL인 경우 Customer.newNull()을 호출 하는 것으로 변경 되었다. newNull은 NullCustomer에 대한 객체를 만들어서 리턴 하게 되므로 getCustomer 호출에서는 어떠한 경우에도 NULL이 리턴 되지 않으며 이를 통해서 NULL에 대한 검사를 완전히 배제 할 수 있게 된다. 생성된 NullCustomer는 사전에 정의된 동작들을 수행 한다. List 2-15에서 getCustomer 호출 후 NULL이 아닐 경우 customer.getName을 호출 하고 NULL 일 경우 customerName에 임의의 값 “occupant”를 할당 하고 있다. 이와 동일한 동작을 하도록 하기 위해 List 2-16의 NullCustomer에서 getName은 “occuptant” 만을 리턴 하도록 하였다. 이런 과정을 거쳐 \_customer에 대한 NULL 검사는 getCustomer 내부에서만 수행 되며 호출 하는 곳에서는 NULL에 대한 검사를 수행 할 필요가 없어지게 되어 코드의 복잡도를 낮출 수가 있게 된다. 또한 NULL 포인트 검사 누락으로 인한 오류도 방지 할 수 있다.

Customer의 getHistory 부분도 NullCustomer내에서 PaymentHistory.newNull를 호출 하도록 변경 되었으며 해당 호출에 대한 NULL 검사를 생략 할 수 있다. Refactoring을 통해서 실행 코드의 복잡도는 4에서1로 감소 하였다.

List 2‑16 NULL에 대한 확인

|  |
| --- |
| **class** NullCustomer **extends** Customer {  **public** **boolean** isNull() {  **return** **true**;  }  **public** String getName(){  **return** "occupant";  }    **public** PaymentHistory getHistory() {  **return** PaymentHistory.newNull();  }  }  **class** NullPaymentHistory **extends** PaymentHistory {  **int** getWeeksDelinquentInLastYear() {  **return** 0;  }  }  **class** Site {  Customer getCustomer() {  **return** (\_customer == **null**) ? Customer.*newNull*(): \_customer;  }  Customer \_customer;  }  **class** Customer {  **protected** Customer() {} //needed by the NullCustomer  **static** Customer newNull() {  **return** **new** NullCustomer();  }  **public** String getName() {...}  **public** BillingPlan getPlan() {...}  **public** PaymentHistory getHistory() {...}  }  **public** **class** PaymentHistory {  **public** **int** getWeeksDelinquentInLastYear() {...}  **static** PaymentHistory newNull() {  **return** **new** NullPaymentHistory ();  }  }  Customer customer = site.getCustomer();  BillingPlan plan = customer.getPlan();  ...  String customerName = customer.getName();  ...  **int** weeksDelinquent = customer.getHistory().getWeeksDelinquentInLastYear(); |

## 클래스의 분리 및 중복 제거

함수의 분리 및 중복 제거와 동일한 형태로 클래스의 분리 및 중복 제거도 복잡도를 낮추고 유지 보수성을 증가 시키기 위해 필수적인 요소 이다. 클래스의 크기와 복잡도가 증가 하게 되면 서로 연관 있는 기능을 다른 클래스로 분리 하는 것이 효과적이다. 여러 클래스에 중복된 코드가 존재 하는 경우 해당 코드 부분을 별도의 클래스로 분리 하여 코드의 중복을 제거 할 수 있다.

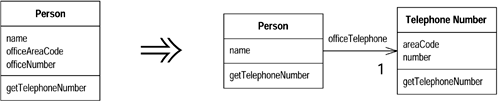


그림 2‑1 Person 클래스를 Person과 TelephoneNumber로 분리 한 경우

그림 2-1은 Person 클래스에서 전화번호 관련 부분을 TelephoneNumer 클래스로 분리 하는 예제이다. List 2-17에서는 전화번호에 대한 속성이 Person 클래스 내에 포함 되어 있다. 만일 Office와 같이 전화 번호를 이용하는 새로운 클래스가 필요할 경우 Person과 Office에는 전화번호 처리를 위한 코드의 중복이 발생 하게 된다. 이러한 문제를 방지 하기 위해 전화번호에 대한 속성은 별도의 클래스로 분리 되는 것이 바람직하다.

List 2‑17 전화번호와 사람의 속성이 섞여 있는 Person Class의 예

|  |
| --- |
| **class** Person {  **public** String getName() {  **return** \_name;  }  **public** String getTelephoneNumber() {  **return** ("(" + \_officeAreaCode + ") " + \_officeNumber);  }  String getOfficeAreaCode() {  **return** \_officeAreaCode;  }  **void** setOfficeAreaCode(String arg) {  \_officeAreaCode = arg;  }  String getOfficeNumber() {  **return** \_officeNumber;  }  **void** setOfficeNumber(String arg) {  \_officeNumber = arg;  }  **private** String \_name;  **private** String \_officeAreaCode;  **private** String \_officeNumber;  } |

List 2-18은 Person에서 전화 번호의 속성을 분리 한 예이다.

List 2‑18 Person에서 전화 번호 속성을 분리한 예

|  |
| --- |
| **class** Person {  **public** String getName() {  **return** \_name;  }  **public** String getTelephoneNumber(){  **return** \_officeTelephone.getTelephoneNumber();  }  TelephoneNumber getOfficeTelephone() {  **return** \_officeTelephone;  }  **private** String \_name;  **private** TelephoneNumber \_officeTelephone = **new** TelephoneNumber();  }  **class** TelephoneNumber {  **public** String getTelephoneNumber() {  **return** ("(" + \_areaCode + ") " + \_number);  }  String getAreaCode() {  **return** \_areaCode;  }  **void** setAreaCode(String arg) {  \_areaCode = arg;  }  String getNumber() {  **return** \_number;  }  **void** setNumber(String arg) {  \_number = arg;  }  **private** String \_number;  **private** String \_areaCode;  } |

List 2-18에서는 전화번호가 필요한 객체가 추가될 경우 TelephoneNumber를 포함 하면 되므로 코드의 중복 문제가 발생 하지 않으며 특정 클래스의 복잡도가 증가 하게 되는 문제도 발생 하지 않는다.

# 요약

본 문서에서는 코드의 복잡도를 낮추기 위한 Refactoring 방법에 대하여 설명 하였다. 처음부터 완벽한 코드를 만드는 일은 이론적으로는 가능하지만 매우 어려운 것이 사실이다. 그러므로 코드 작성 과정에서 기능을 구성 하고 해당 코드의 개선점을 찾고 Refactoring을 수행 하는 것은 개발 및 유지보수 과정에서 반복적으로 끊임 없이 이루어져야 한다.

이러한 반복 과정에서의 오류를 막고 Refactoring을 안전하게 수행 하기 위해서는 Test (Unit/Acceptance Test)가 반드시 필요하다. 테스트는 해당 수정으로 인한 Side effect 발생 여부를 쉽게 확인 할 수 있게 해줌으로써 개발자들이 자신감을 갖고 Refactoring을 수행 할 수 있게 도와 준다.

만일 코드의 복잡도를 낮추어야 하는 부분에 Test가 작성 되어 있지 않다면 (대부분의 경우가 여기에 해당 될 것이다.) 개발자의 최초 임무는 해당 부분에 Test를 추가 하는 것이다. Test가 추가 되고 나면 Refactoring을 수행 하고 그것에 맞추어 Test를 수정 하고 다시 Refactoring 하는 과정을 반복적으로 수행 하여 원하는 결과를 얻어야 한다. 이렇게 Legacy Code에 조금씩 테스트가 추가 되는 과정을 거쳐 Legacy Code는 Test Coverage가 높은 안전한 코드로 변화 될 수 있다. 로마가 하루 아침에 이루어 지지 않았듯이 Test Coverage가 높은 안전한 코드도 하루 아침에 이룰 수 없다. 지속적이고 꾸준한 개발자들의 관심과 노력이 모여 시간에 따라 안정성이 높아지는 코드를 생산 할 수 있게 되는 것이다.

막막한 Legacy Code 앞에서 좌절 하지 않고 개선 활동을 꾸준히 수행해 나간다면 해당 코드의 품질뿐만 아니라 관련된 모든 부분의 품질 향상이 이루어 질 것이다.

본 문서에서 기술한 Refactoring 방법들은 사용 하기 용이 한 것에서부터 사용에 주의를 기울여야 하는 방법들도 있다. 특히, NULL Object와 다형성은 설계 변경이 발생하게 되는 부분으로 사용에 세심한 주의를 기울여야 한다. 해당 Refactoring 기법을 사용하기 전 관련 지식을 충분히 습득 한 뒤에 사용 하기를 권고 한다.

모든 개발자 분들의 즐겁고 성공적인 Refactoring을 기원 하며 문서를 마치도록 하겠다.

1. 출처: <http://www.aivosto.com/project/help/pm-complexity.html> [↑](#footnote-ref-1)
2. <http://en.wikipedia.org/wiki/God_object> 너무 많은 기능을 수행 하는 클래스를 지칭 하는 것으로 대표적인 Anti-pattern 중의 하나이다. [↑](#footnote-ref-2)
3. 출처Martin Folwer et al. Refactoring improving the Design of Existing Code Addison-Wesley 1999 [↑](#footnote-ref-3)
4. 출처: <http://www.cs.oberlin.edu/~jwalker/nullObjPattern/> [↑](#footnote-ref-4)