Chap 09 코딩

목 차

코딩 로드맵

- 소프트웨어 제품의 품질은 결국 원시 코드에 모두 귀결

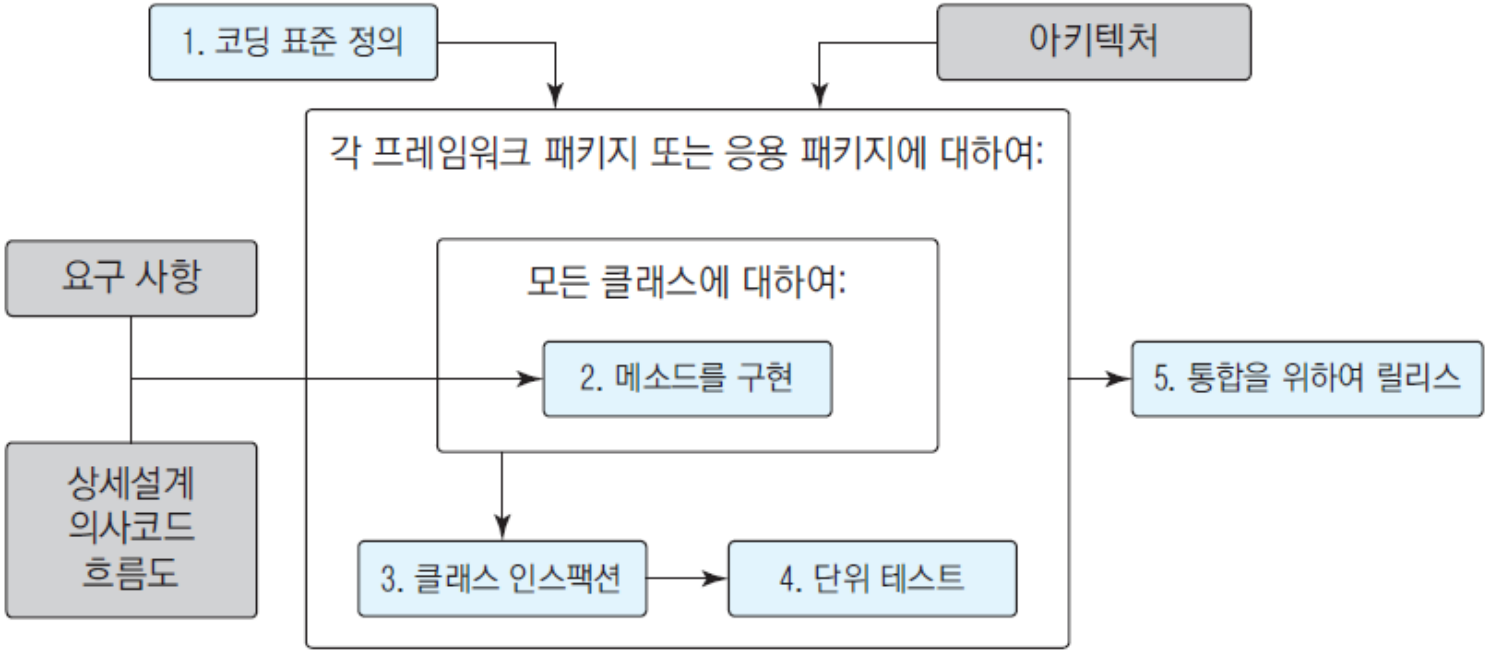


그림 9.1 코딩 작업의 로드맵

8.1 코딩 작업

- 설계 명세에 나타낸 대로 요구를 만족할 수 있도록 프로그래밍
- 오류가 적은 품질 좋은 프로그램
- 작업 과정
  1. 원시코드를 같은 스타일로 만들기 위하여 코딩 표준을 만든다.
  2. 아키텍처 설계 결과 프레임워크 패키지와 응용 패키지를 결정
  3. 클래스 구현이 끝나는 대로 인스펙션
  4. 클래스 단위로 테스트
  5. 클래스나 패키지를 릴리스 하여 응용 시스템으로 통합

자주 발생하는 오류

- 다양한 방법으로 발견되지만 흔히 발견되는 오류는 정해져 있다
  - 메모리 누수
  - 중복된 프리 선언
  - NULL의 사용
  - 별칭의 남용
  - 배열 인덱스 오류
  - 수식 예외 오류
  - 하나 차이에 의한 오류
  - 사용자 정의 자료형 오류
  - 스트링 처리 오류
  - 버퍼 오류
  - 동기화 오류

(1) 메모리 누수

- 메모리가 프리 되지 않고 프로그램에 계속 할당되는 현상
- 장기로 수행되는 시스템에는 치명적인 영향을 줄 수 있음

```
char * foo(int s)
{
    char *output;
    if ( s>0 )
        output = (char *) malloc (size);
    if ( s==1 )
        return NULL;      /* if s==1 then memory leaked */
    return(output);
}
```

(2) 중복된 프리선언

- 프로그램 안에서 사용하는 자원은 먼저 할당되고 사용 후에는 프리로 선언
- 이미 프리로 선언된 자원을 또 다시 프리로 선언하는 경우 오류

```
main()
{
        char *str;
        str = ( char * ) malloc (10);
        if (global ==0 )
                free(str);
        free(str);        /* str is already freed
}
```

(3) NULL의 선언

- NULL을 포인트 하고 있는 곳의 콘텐츠를 접근하려고 하면 오류
- 이러한 오류는 시스템을 다운시킴

```
char *ch = NULL;
if ( x>0 )
{
      ch = 'c';
}
printf("%c", *ch);    // ch may be NULL
*ch = malloc(size);
ch = 'c';    // ch will be NULL if malloc returns NULL
```

(4) 별칭의 남용

- 별칭(alias)은 많은 문제를 야기

- 서로 다른 주소 값을 예상하고 사용한 두 개의 변수의 값이 별칭 선언으로 인하여 같은 값이 되었을 때 오류 발생

(5) 배열 인덱스 오류

- 배열 인덱스가 한도를 벗어나면 예외 오류 발생
- 배열 인덱스가 음수 값을 갖는 경우 오류 발생

```
dataArray[80];
for ( i=0; i<=80; i++)
        dataArray[i] = 0;
```

(6) 수식 예외 오류

- 0으로 나누는 오류
- 변동 소수점 예외 오류

(7) 하나 차이에 의한 오류

- 0으로 시작하여야 할 것을 1로 시작
- <=N으로 써야 할 곳에 <N을 쓴 경우
- 컴파일러나 테스트 도구에 의하여 검출되지 않는 경우가 많음

(8) 사용자 정의 자료형 오류

- 오버플로우나 언더플로 오류가 쉽게 발생

- 사용자 정의 자료형의 값을 다룰 때는 특별한 주의

```
typedef enum{A, B, C, D} grade;
void foo(grade X)
{
    grade l, m;
    l = x-1;    //Underflow possible
    m = x+1;  //Overflow possible
}
```

(9) 스트링 처리 오류

- strcpy, sprintf 등 많은 스트링 처리 함수가 있다
  - 예) strcpy(NULL, “hello”);

- 매개변수가 NULL

- 스트링이 NULL로 끝나지 않았을 경우

(10) 버퍼 오류

- 프로그램이 버퍼에 복사하여 입력 받으려 할 때 입력 값을 고의로 아주 크게 주면 스택의 버퍼에 오버플로 발생
- 버퍼 오버플로를 이용하여 해커들이 자신의 코드를 실행시킬 수 있음

```
void mygets(char *str) {
    int ch;
    while ( ch = getchar() != ‘\n’ && ch != ‘\0’ )
            *(str++) = ch;
    *str = ‘\0’;
}
main() {
    char s2[4];
    mygets( s2 );
}
```

(11) 동기화 오류

- 공통 자원을 접근하려는 다수의 스레드가 있는 병렬 프로그램에서 흔히 발생함
- 데드락(deadlock) – 다수의 스레드가 서로 자원을 점유하고 릴리스 하지 않는 상태
- 레이스 컨디션 – 두 개의 스레드가 같은 자원을 접근하려 하여 수행 결과가 스레드들의 실행 순서에 따라 다르게 되는 경우
- 모순이 있는 동기화 – 공유하는 변수를 접근 할 때 로킹과 언로킹을 번갈아 하는 상황에서 오류가 많이 발생

9.2 코딩 표준

- 대표적인 기준은 간결하고 읽기 쉬운 것
  - 간결함 : 복잡하지 않고 명확하여 이해하기 쉬운 것
  - 읽기 쉽다 : 프로그램을 대충 훑어보거나 이해하기 쉬운 것

- 설계에서 모듈화의 목표, 높은 응집력, 낮은 결합도를 달성하였다면 모듈은 간단해짐

명명 규칙

- 카멜 표기법
  - Java 클래스, 필드, 메소드 및 변수 이름
  - 여러 단어를 함께 붙여 쓰되 각 단어의 첫 글자는 대문자
  - thisIsAnExample
- 상수는 모두 대문자로
  - public static final double SPEED_OF_LIGHT= 299792458; // in m/s
- 클래스와 인터페이스 이름
  - 명사 또는 명사구이며 대문자로 시작
  - interface AqueousHabitat { ... }
  - class FishBowl implements AqueousHabitat { ... }

메서드 이름

- 일반적으로 소문자로 시작하는 동사구
  - public void setTitle(String t) { ... }
- 함수 이름은 일반적으로 값을 설명하는 명사구로
  - public double areaOfTriangle(int b, int c, int d) { ... }
- 필드(예를 들어 title)의 값을 접근하여 리턴하는 함수는 “get”
  - public String getTitle() { ... }
- 조건을 묻는 부울 함수의 이름은 대개 “is”로 시작
  - public boolean isEquilateralTriangle(int b, int c, int d) { ... }

변수 이름

- 일반적으로 소문자로 시작
- 용도에 대한 힌트를 제공해야
- 모호한 이름을 사용하지 않아야
- 변수 이름의 길이는 적당하게
  - hypotenuseOfTriangle= 6 * (2 + hypotenuseOfTriangle) + 7 / hypotenuseOfTriangle - hypotenuseOfTriangle;
  - x= 6 * (2 + x) + 7 / x - x;
- 대상이 사용된 위치를 고려
  - 매개 변수: 되도록 짧게
  - 필드 변수: 가능한 길고 의미가 담겨 있어야

패키지 이름

- 일반적으로 모두 소문자이며 명사로

헝가리안 표기법

- 변수 이름 앞에 타입의 구별이
  - lAccountNum : long integer(“l”) 타입의 변수
  - arru8NumberList : unsigned 8-비트 정수(“arru8”) 변수의 배열
  - bReadLine(bPort,&arru8NumberList) : 바이트값의 리턴 코드를 가진 함수
  - strName : 스트링을 가진 변수
- 변수의 목적 또는 변수가 무엇인지에 대한 힌트를 줌
  - rwPosition : 행(“rw”)을 나타내는 변수
  - usName : 사용되기 전에 정제할 필요가 있는 안전하지 않은 스트링(“us”)
  - szName : ‘\0’로 끝나는 스트링(“sz”) 변수

형식

- 들여쓰기와 괄호
  - 프로그램 구조를 명확하게 하기 위해 사용
  - 문장의 일부와 선언문은 들여 쓰기

```
if (x < y) {
   x= y;
   y= 0;
}
else {
   x= 0;
   y= y/2;
}
```

```
if (x < y) {
   x= y;
   y= 0;
} else {
   x= 0;
   y= y/2;
}
```

블록은 항상 괄호를 사용

- if (flag) validate(); update();
  - 의도가 두 개의 문장이 모두 if 제어 구조에 포함된 문장이라면 잘못

```
if (flag) {
   validate();
   update();
}
```

- 키워드와 다음 괄호 사이에 공백을 두어 제어 구조와 메서드 호출과 구별하게

```
// Good if_(_username_==_null_) {...}
// Less Good if(username==null) {...}
```

문장과 수식

- 문장이나 수식은 메소드나 클래스로 패키지화
- 블록 문장
  - 제어구조가 중첩되었을 때 생기는 혼란을 줄일 수 있음

```
if (x >= 0)
   if (x > 0) positiveX();
else // Oops! Actually matches most recent if!
   negativeX()
```

- 수식
  - 괄호를 이용하여 오퍼레이션의 순서를 명확히

```
// Extraneous but useful parentheses.
int width = (( buffer * offset ) / pixelWidth ) + gap;
```

오류 처리

- 잘못된 데이터를 어떻게 다룰 것인가
  - 예: 실제로는 없는 계좌번호
- 매개변수 오류
  - 예: evaluate()라는 메소드가 매개변수로 ‘car’, ‘truck’, ‘bus’만을 받아들이므로, 자료형으로 제한

    evaluate(String vehicleP) : 오류 가능하므로 비추천

    evaluate(SpecializedVehicle vehicleP)
- 입력 오류 방지
  - 리스트 박스
  - 디폴트 값을 지정

주석

- 디버깅하는 동안 얻을 수 있는 도움
- 다른 사람들이 프로그램을 이해하기 쉽도록 함
- 주석 다는 법
  - 불필요한 단어는 생략
  - 과도한 주석 피함
- 클래스 불변조건(invariant)
  - 클래스의 속성에 대한 의미와 제약 기술

```
private int hr; // The hour of the day, in 0..23.
private double[] temps; // temps[0..numRecorded-1] are the recorded temperatures
private int numRecorded; // number of temperatures recorded
```

메서드 주석

- 메서드가 하는 일을 설명하는 Javadoc 스펙을 선행 조건과 함께 쓴다

  @param b one of the sides of the triangle

  @return The area of the triangle

- 무언가를 수행한다는 문장으로 작성하여 관련된 모든 매개 변수를 언급

  /** Print the sum of a and b. */

  public static void printSum(int a, int b) { ... }

클래스, 문장 주석

- 파일의 시작 부분에는 클래스의 용도를 설명하는 주석이 포함되어야

```
/** An object of class Auto represents a car.
Author: Eun Man Choi.
Date of last modification: 25 November 2019 */
public class Auto { ... }
```

- 문장 주석
  - 논리적 단위로 그룹화

```
// Truthify x >= y by swapping x and y if needed.
if (x < y) {
   int tmp= x;
   x= y;
   y= tmp;
}
```

9.3 설계에서 코드 생성

- IDE: Integrated Development Environment, 통합개발환경
  - UML 다이어그램으로부터 원시코드 골격을 자동 생성
  - 이클립스, 비주얼 스튜디오, 엑스 코드 등
- 연관의 코딩
  - 1대 1 연관
  - 1대 N
  - N대 N

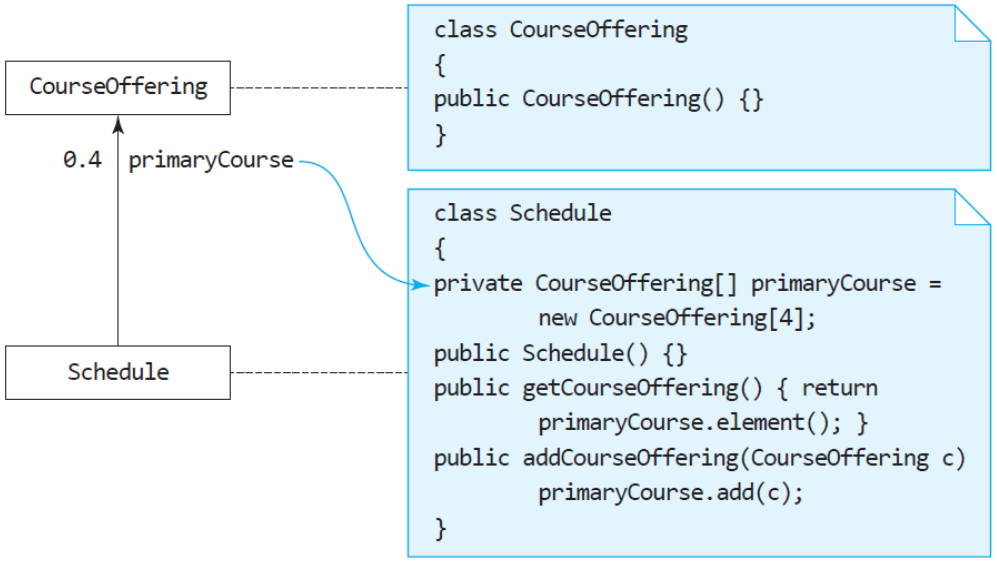


그림 9.2 1대 N 연관의 구현

연관의 구현

- 클래스 A와 B사이에 1대 1연관 관계가 있다
- 1대 1 연관
  - A에서 B의 함수를 호출할 필요가 있다면 A가 B에 대한 참조를 갖도록 구현
  - 반대로 B에서 A의 함수를 호출할 필요가 있다면 B가 A에 대한 참조를 갖도록 구현
- 1대 다
  - 클래스 A에서 인스턴스 B의 메소드를 호출할 것이 있다면 클래스A가 클래스B의 참조를 모음으로 가지고 있도록 구현
- 다대 다
  - 중간에 연관 클래스를 도입하여 1대 다의 관계로 바꾸어 설계하기도 한다

시퀀스 다이어그램의 구현

- A 객체에서 B 객체로 나가는 메시지가 표시되어 있다면
  - 메소드는 B 클래스에 정의

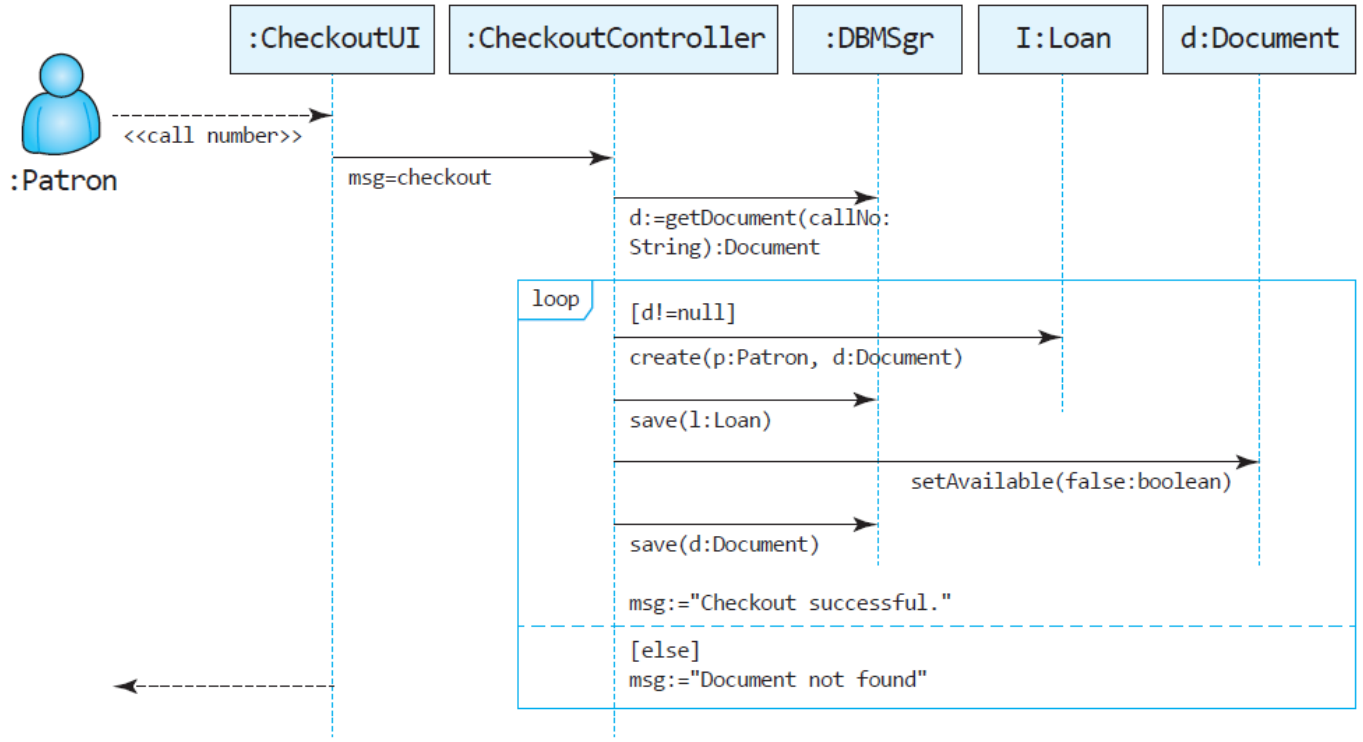


그림 9.3 Checkout Document 사용 사례

8.4 리팩토링

- 결과의 변경 없이 코드의 구조를 재조정
- 이미 존재하는 코드의 디자인을 안전하게 향상시키는 기술
- 가독성을 높이고 유지보수를 편하게 하기 위한 것

기본 개념

- 코드스멜
  - 프로그램에 대한 작업을 어렵게 만드는 것
  - 읽기 어려운 프로그램
  - 중복된 로직을 가진 프로그램
  - 실행 중인 코드를 변경해야 하는 특별한 동작을 요구하는 프로그램
  - 복잡한 조건문이 포함된 프로그램

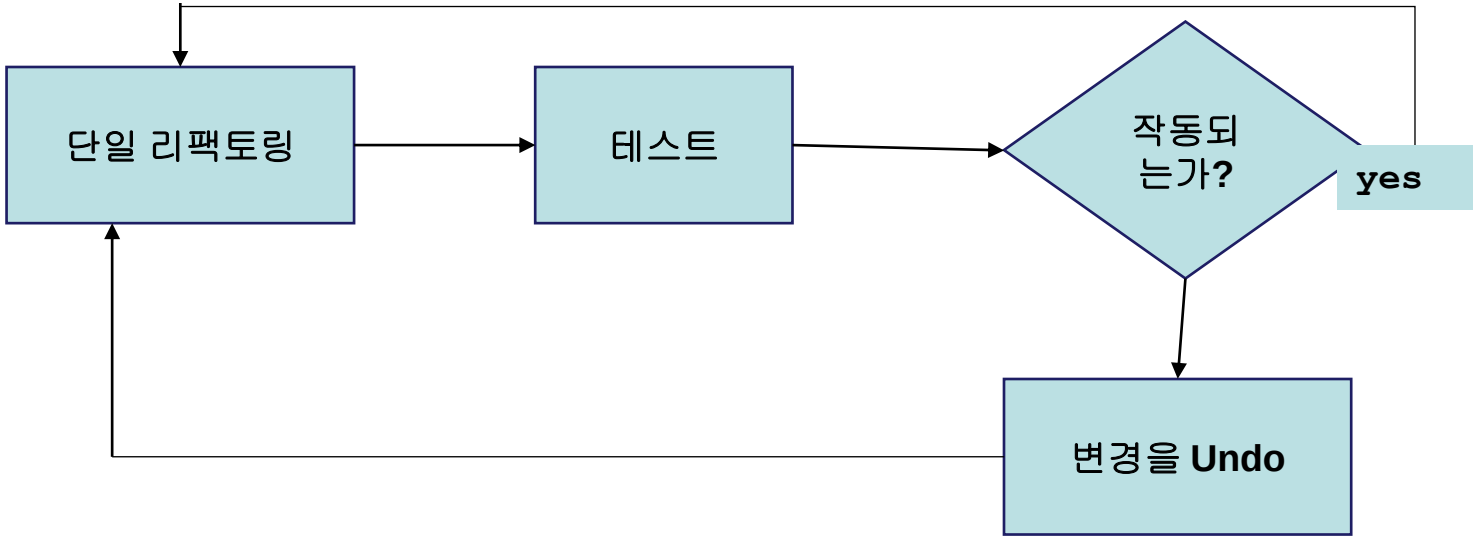
- 소프트웨어를 보다 쉽게 이해할 수 있고 적은 비용으로 수정할 수 있도록 겉으로 보이는 동작의 변화 없이 내부구조를 변경하는 것

리팩토링 목적

- 소프트웨어의 디자인을 개선시킨다
- 소프트웨어를 이해하기 쉽게 만든다
- 버그를 찾는데 도움을 준다
- 프로그램을 빨리 작성할 수 있게 도와준다

리팩토링 과정

- 소규모의 변경 – 단일 리팩토링
- 코드가 전부 잘 작동되는지 테스트
- 전체가 잘 작동하면 다음 리팩토링 단계로 전진
- 작동하지 않으면 문제를 해결하고 리팩토링 한 것을 undo하여 시스템이 작동되도록 유지



코드 스멜

- 리팩토링으로 해결될 수 있는 문제가 있다는 징후를 알려주는 것

코드 스멜	설명	리팩토링
중복된 코드	기능이나 데이터 코드가 중복된다	중복을 제거한다
긴 메소드	메소드의 내부가 너무 길다	메소드를 적정 수준의 크기로 나눈다
큰 클래스	한 클래스에 너무 많은 속성과 메소드가 존재한다	클래스의 몸집을 줄인다
긴 파라미터 리스트	메소드의 파라미터 개수가 너무 많다	파라미터의 개수를 줄인다
두 가지 이상의 이유로 수정되는 클래스 (Divergent Class)	한 클래스의 메소드가 2가지 이상의 이유로 수정되면, 그 클래스는 한 가지 종류의 책임만을 수행하는 것이 아니다	한 가지 이유만으로도 수정되도록 변경한다

코드 스멜

여러 클래스를 동시에 수정 (Shotgun Surgery)	특정 클래스를 수정하면 그때마다 관련된 여러 클래스들 내에서 자잘한 변경을 해야 한다	여러 클래스에 흩어진 유사한 기능을 한곳에 모이게 한다
다른 클래스를 지나치게 애용(Feature Envy)	빈번히 다른 클래스로부터 데이터를 얻어 와서 기능을 수행한다	메소드를 그들이 애용하는 데이터가 있는 클래스로 옮긴다
유사 데이터들의 그룹 중복(Data Clumps)	3개 이상의 데이터 항목이 여러 곳에 중복되어 나타난다	해당 데이터들은 독립된 클래스로 정의한다
기본 데이터 타입 선호 (Primitive Obsession)	객체 형태 그룹을 만들지 않고, 기본 데이터 타입만 사용한다	같은 작업을 수행하는 기본 데이터의 그룹을 별도의 클래스로 만든다
Switch, If문장	switch 문장이 지나치게 많은 case를 포함한다	다형성으로 바꾼다(같은 메소드를 가진 여러 개의 클래스를 구현한다)
병렬 상속 계층도 (Parallel Inheritance Hierarchies)	[Shotgun Surgery]의 특별한 형태로서, 비슷한 클래스 계층도가 지나치게 많이 생겨 중복을 유발한다	호출하는 쪽의 계층도는 그대로 유지하고 호출당하는 쪽을 변경한다

사례: 메서드 추출

```
...
  if (i != min) {
     int temp = num[i];
     num[i] = num[min];
     num[min] = temp;
  }
...
```

```
  if (i != min) {
     swap(ref num[i], ref num[min]);
  }
...
void swap(ref int a, ref int b) {
   int temp = a;
   a = b;
   b = temp;
  }
```

사례: 클래스 추출

```
public class Customer {
    private String name;
    private String workPhoneAreaCode;
    private String workPhoneNumber;
}
```

```
public class Customer {
    private String name;
    private Phone workPhone;
}
public class Phone {
    private String areaCode;
    private String number;
}
```

8.5 코드 품질 향상 기법

- 코드 인스펙션
  - 설계나 프로그램 코드에서 결함을 찾아 확인
- 정적 분석
  - 수행되지 않는 데드코드가 없는지, 선언이 되지 않고 사용한 변수가 없는지 등을 검사
- 페어 프로그래밍
  - 애자일 방법에서 프로그래밍과 테스팅을 담당하는 두 사람이 머신을 공유하며 코딩

(1) 코드 인스펙션

- 프로그램이 성공적으로 컴파일 되고 정적 분석 도구에 의하여 검사된 후에 이루어짐
- 코드에 묻혀있는 결함을 찾아내는 것
  - 효율성, 코딩 표준의 준수 여부 등
- 심각도
  - 결함의 우선순위를 나타냄
- 결함의 타입
  - 로직 문제
  - 컴퓨팅 문제
  - 인터페이스/타이밍 문제
  - 데이터 처리

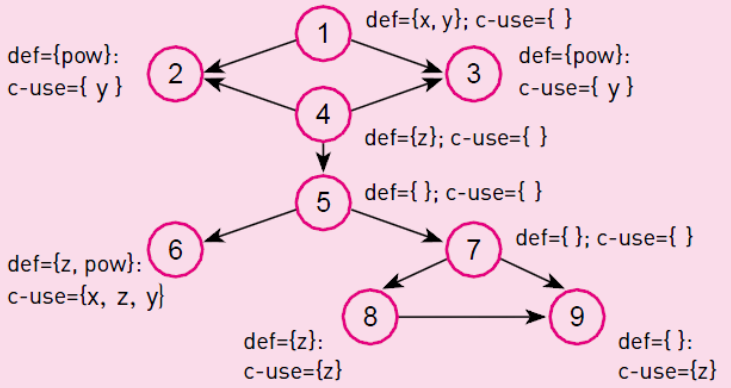
(2) 정적 분석

- 프로그램 텍스트를 조직적으로 분석하여 결함을 찾아내는 것
- 소프트웨어 도구를 이용하여 자동으로 가능
- 결함을 찾는데 두 가지 방법 사용
  - 코드에 존재하는 결함으로 나타날 비정상적인 패턴이나 원하지 않는 패턴을 찾는 방법
  - 실행할 때 프로그램의 고장을 일으킬 코드 상에 존재하는 결함을 직접 찾는 방법

정적 분석

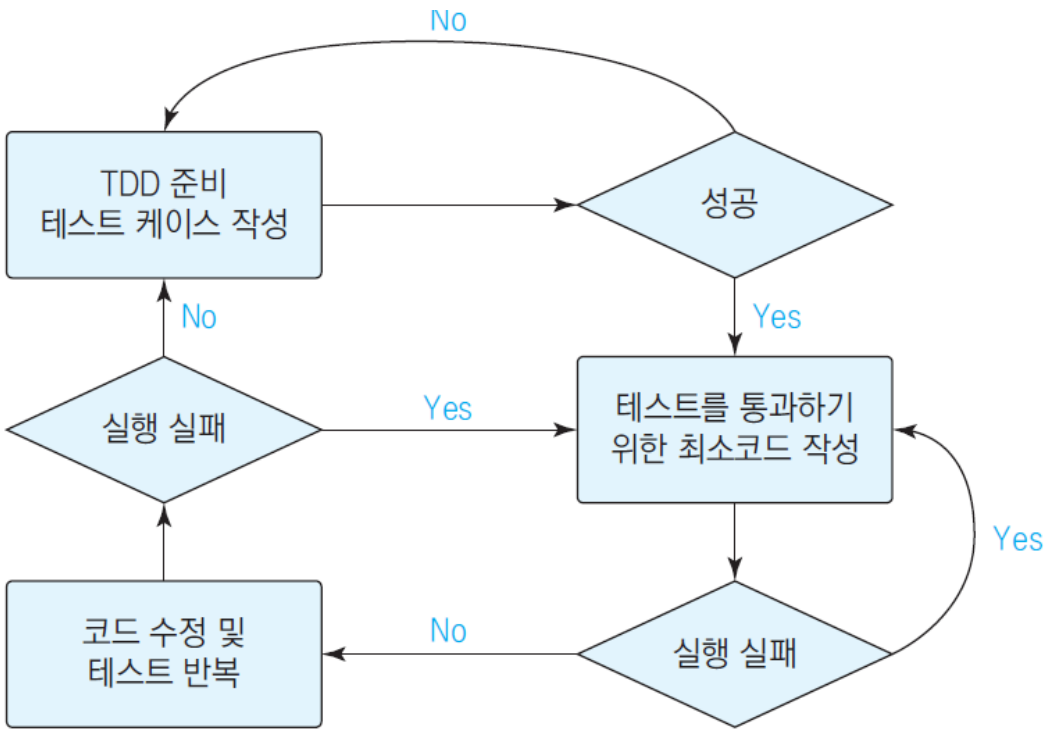
- 자료 변칙(data anomaly)
  - 자료 흐름도를 분석하면 자료가 정의되지 않고 사용되거나 정의되고 사용되는 비정상적인 패턴을 찾는 것
- 사족(redundant)
  - 컴파일러에 의하여 검출되지 않는다
  - 중복된 배정문, 데드 코드, 조건 중복 등

```
(1)  scanf(x, y);
     if (y < 0)
(2)      pow = 0 - y;
(3)  else  pow = y;
(4)  z = 1.0;
(5)  while (pow != 0)
(6)        { z = z * x; pow = pow - 1; }
(7)  if ( y < 0 )
(8)        z = 1.0/ z;
(9)  printf(z);
```



테스트 중심 개발

- 테스트를 위한 코드를 작성한 후 기능을 구현
- 주로 클래스 안에 있는 메서드를 시험
- 개발 과정



테스트 코드

- 대상

```
public class MyUnit {
   public String concatenate(String one, String two){
      return one + two;
   }
}
```

- 테스트

```
import org.junit.Test;
import static org.junit.Assert.*;
public class MyUnitTest {
@Test
   public void testConcatenate() {
      MyUnit myUnit = new MyUnit();
      String result = myUnit.concatenate(“one”, “two”);
      assertEquals(“onetwo”, result);
   }
}
```

짝 프로그래밍

- 두 사람이 같은 컴퓨터를 사용하면서 같이 프로그램 한다
- 스트레스와 성공의 기쁨을 나눌 상대가 있기 때문에 프로그래밍에 재미가 있고 압박을 줄일 수 있다
- 파트너와 아이디어를 교환하기 때문에 팀의 소통을 향상시킬 수 있음
- 팀 구성원이 서로를 이해하고 여러 가지 방법으로 서로에게서 배우기 때문에 상호 이해와 협력이 향상
- 토론이 창의적인 사고로 발전하여 문제 해결에 도움이 되므로 간단하고 효율적인 해법을 만들어 낼 수 있음