

# **소프트웨어 공학 (Software Engineering)**

**코딩  
(Coding)**



## In this chapter ... (1/2)

코딩(Coding)

- 좋은 프로그램을 작성하기 위한 원리와 목표는?
- 코딩 단계의 과정과 흔히 일어나는 오류는?
- 코딩 스타일이란 무엇이며 좋은 코드가 되기 위한 가이드는?
- 리팩토링이란 무엇이며, 코드 스멜의 사례와 리팩토링 방법은?
- 코드 품질을 높이는 방법에는 어떤 것이 있는가?





## In this chapter ... (2/2)

코딩(Coding)

- 읽기 쉽고 이해가 쉬운 코딩은 후에 유지보수 작업을 수월하게 해 줌
- Weinberg의 실험

- 코딩 목표를 모두 동시에 만족시키기 어려움  
(어느 하나에 집중하면 다른 목표들은 최적의 결과를 얻기 어려움)
- 다섯 팀이 같은 문제에 대하여 프로그래밍
- 코딩 작업의 목표를 다르게 두고 작업 후 결과 비교

	결과(1 = 최적)				
	O1	O2	O3	O4	O5
코딩 작성에 드는 노력의 최소화(O1)	1	4	4	5	3
문장 수의 최소화(O2)	2-3	1	2	3	5
소요 메모리 최소화(O3)	5	2	1	4	4
프로그램 명확성의 극대화(O4)	4	3	3	2	2
출력 명확성의 극대화(O5)	2-3	5	5	1	1



## In this chapter ...

코딩(Coding)



### 8.1 코딩 원리



### 8.2 코딩 스타일



### 8.3 UML과 코딩



### 8.4 리팩토링



### 8.5 코드 품질 향상 기법



## 분리하여 구현할 수 있는 작은 단위를 프로그래밍 하는 작업

- 절차적 방법: 함수 내부를 완성
- 객체지향 방법: 메소드의 코딩

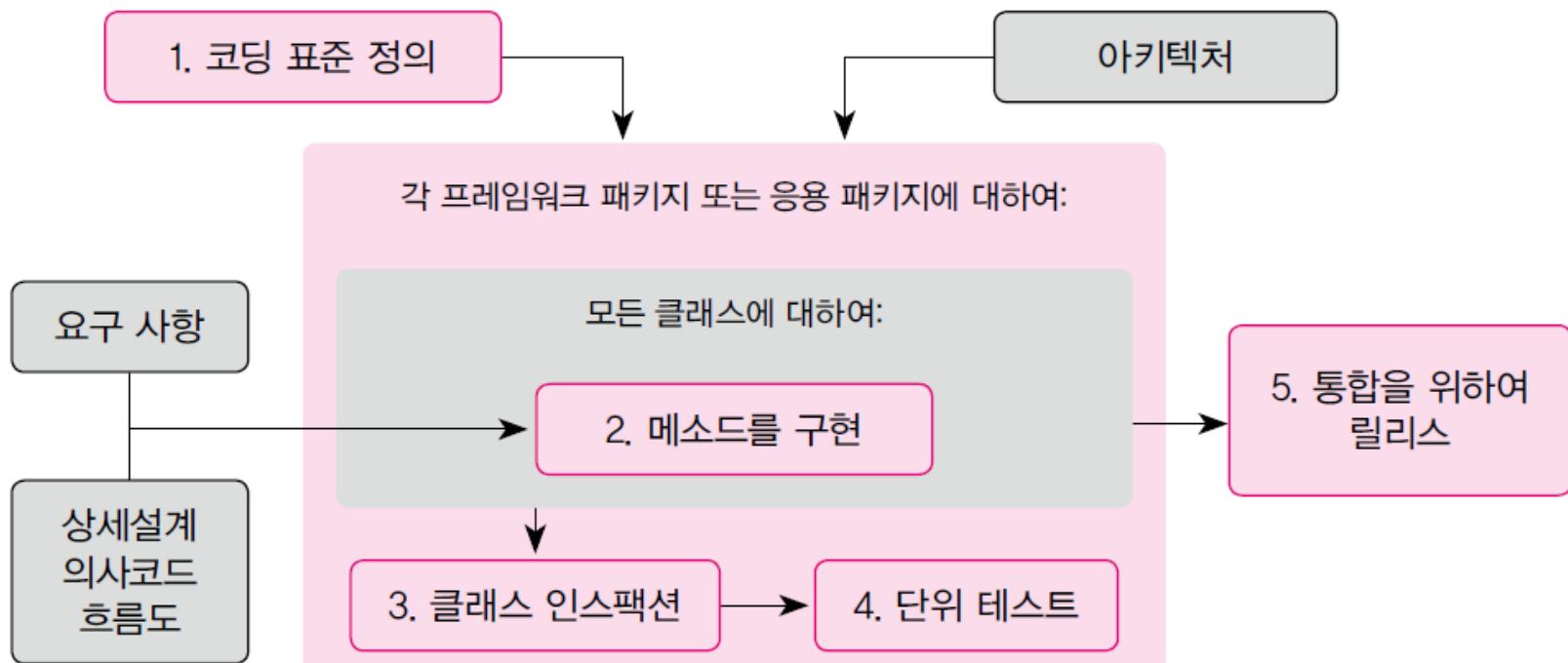
## 설계 명세에 나타난 대로 요구를 만족할 수 있도록 프로그래밍

- 앞서의 결과물인 아키텍처 설계서, 요구 분석서 등을 잘 참조하여 코딩
- 신속한 코딩 보다는 오류가 적은 품질 좋은 프로그램 작성이 중요



# 코딩 단계의 로드맵

코딩(Coding)





다양한 오류가 다양한 방법으로 발견될 수 있음

흔히 발견되는 오류는 정해져 있음

- 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

코딩(Coding)

- ▣ 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

코딩(Coding)

- 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 잘못된 사용

코딩(Coding)

- \_NULL을 포인트 하고 있는 곳의 콘텐츠를 접근하려고 하면 오류
- 이러한 오류는 시스템을 다운 시킴

```
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

코딩(Coding)

- ❶ 별칭(alias)은 많은 문제를 야기할 수 있음
- ❷ 서로 다른 주소 값을 예상하고 사용한 두 개의 변수의 값이 별칭 선언으로 인하여 같은 값이 되었을 때 오류 발생
- ❸ 사례: strcat(src, dst)의 스트링 결합 프로그램에서, src와 dst의 주소가 다를 것으로 예상하였는데, src가 dst의 별칭이라면 런타임 오류 가능성



## (5) 배열 인덱스 오류

코딩(Coding)

- 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;
```



## 수식 예외 오류

- 0으로 나누는 오류
- 부동소수점 예외 오류

## 하나 차이에 의한 오류

- 0으로 시작하여야 할 것을 1로 시작
- $\leq N$ 으로 써야 할 곳에  $< N$ 을 쓴 경우
- 컴파일러나 테스트 도구에 의하여 검출되지 않는 경우가 많음



## (8) 사용자 정의 자료형 오류

코딩(Coding)

- 오버플로우나 언더플로우 오류가 쉽게 발생
- 사용자 정의 자료형의 값을 다룰 때는 특별한 주의

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



## (9) 스트링 처리 오류

코딩(Coding)

- ④ strcpy(), sprintf() 등 많은 스트링 처리 함수가 있음
- ④ 매개변수가 NULL인 경우
- ④ 스트링이 NULL로 끝나지 않았을 경우
- ④ Source 매개변수의 크기가 destination 매개변수보다 크지 않을 경우



## (10) 버퍼 오류

코딩(Coding)

- 프로그래밍이 버퍼에 복사하여 입력 받으려 할 때 입력 값을 고의로 아주 크게 주면 스택의 버퍼에 오버플로우 발생
- 버퍼 오버플로우를 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 );  
}
```

- 입력 값이 매우 크면 s2가 오버플로우되고, mygets()의 리턴 주소를 교묘히 조작하여 해킹 프로그램의 주소로 바꾸어 놓을 수 있음



## (11) 동기화 오류

코딩(Coding)

- ④ 공통 자원을 접근하려는 다수의 스레드가 있는 병렬 프로그램에서 흔히 발생함
- ④ 데드락(deadlock): 다수의 스레드가 서로 자원을 점유하고 릴리스 하지 않는 상태
- ④ 레이스 컨디션: 두 개의 스레드가 같은 자원을 접근하려 하여 수행 결과가 스레드들의 실행 순서에 따라 다르게 되는 경우
- ④ 모순이 있는 동기화: 공유하는 변수를 접근할 때 로킹과 언로킹을 번갈아 하는 상황에서 오류가 많이 발생



- 순차, 선택, 반복으로 모든 제어 흐름을 표현
- GO TO문 사용은 구조적인 제어 흐름을 해치지 않는 범위에서 사용  
→ 가능하면 머리 속에서 "goto" 를 지우는 것이 바람직함
- GO TO문을 이용한 오류 처리 사례

```
DO 50 I=1, COUNT
...
IF (ERROR1) GO TO 60
...
IF (ERROR2) GO TO 70
...
50 CONTINUE
60 {Code for Error1 handling}
    GO TO 80
70 {Code for Error2 handling{
80 CONTINUE
```



- ④ 모듈 사이에 결합을 줄이고 시스템의 유지보수를 쉽게 만듦
  
- ④ 소프트웨어 개발에서 복잡함을 다루는 중요한 수단
  - 정보/데이터에 대해서 정해진 오퍼레이션만 정의하고 공개
  - 모듈 사이의 결합을 줄이고 시스템 유지보수를 쉽게 만드는 장점
  
- ④ C++, Java 등은 정보 은닉 실현 메커니즘을 언어 자체가 가지고 있음
  - 클래스 안에 자료를 정의하면 묵시적으로 외부에서의 접근을 방지
  
- ④ C, Pascal 등의 절차 중심 언어에서는 프로그래머가 주의하여 정보 은닉 원리를 따라야 함



## 설계 명세나 프로그램 안에 중복되는 내용이 들어있는 경우

- 중복된 부분은 비효율적
- 변경 필요 시 중복된 다른 부분도 고쳐야 함

## 중복 사례:

길이는 시작점과 끝점으로 정의되고 한 점이 변하면 길이도 변함

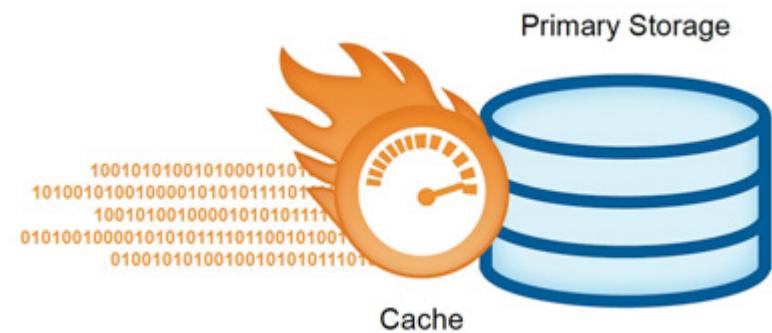
```
class Line {  
public:  
    Point start;  
    Point end;  
    double length;  
}
```



```
class Line {  
public:  
    Point start;  
    Point end;  
    double length() {  
        return start.distance(end);  
    }  
}
```



- ▣ 효율을 위해 중복을 의도하는 경우가 많음 → 제어된 중복
- ▣ 대표적 사례가 캐싱(caching)
- ▣ 중복의 영향이 밖으로 노출되지 않도록 내부에 국한시킴





## In this chapter ...

코딩(Coding)

8.1 코딩 원리

8.2 코딩 스타일

8.3 UML과 코딩

8.4 리팩토링

8.5 코드 품질 향상 기법





## 袆 옷을 입는 것에도 스타일이 있다.

- 빨간색을 좋아하는 사람
- 진바지를 좋아하는 사람
- 청바지가 잘 어울리는 여자 ...

## 袆 프로그램에도 스타일이 있다?

- 정수형은 사용하지 않는 사람 → 모름지기 수란 모두 실수여~
- 다섯 줄만 넘으면 함수로 분리하는 사람 → 모듈화 안 배웠어~ 나눠! 나눠! 나눠!
- 포인터를 엄청 많이 사용하는 사람 → 난 포인터의 황제다... 어렵지~ 나만 알면 장땡~

## 袆 대표적인 기준은 간결하고 읽기 쉬운 것

- 간결하다: 복잡하지 않고 명확하여 이해하기 쉬운 것
- 읽기 쉽다: 프로그램을 대충 훑어보거나 이해하기 쉬운 것



의미 있는 이름을 붙여야 함

→ 이름만 보아도 클래스인지 멤버 함수인지 상수인지 파악할 수 있도록

Sun Microsystems의 Java Software Development Kit에 나오는 식별자  
명명 규칙을 인용

변수 타입을 고려한 명명 규칙 사례

문자형	c, d, e
좌표	x, y, z
예외	e
그래픽	g
객체	o
스트림	in, out, inout
스트링	s



## 패키지 이름

- 패키지는 이름 앞에 인터넷 도메인 이름이 붙임
- 예제) 패키지 npkg를 배포할 서버가 nserver.com이면 패키지 이름은 com.nserver.npy
- 루트 이름은 패키지의 목적과 용도가 잘 나타나도록 의미 있는 약어
- 예제) java.io, java.net

## 타입 이름

- 클래스와 인터페이스의 이름을 붙일 때 첫 글자는 대문자

```
public class PrintStream  
    extends FilterOutputStream {...}
```

- 클래스의 이름은 명사를 사용      `class CustomerAccount {...}`

- 인터페이스의 이름은 명사나 형용사를 사용

```
public interface Runnable {     public void run();   }
```



## 메소드 이름

- 첫 단어는 소문자로 시작하고 연속되는 단어의 첫 글자는 대문자로 구별  
`public void actionPerformed(int amount) { ... }`
- 첫 글자가 소문자인 이유는 메소드의 호출과 생성자 호출을 구별하기 위함
- 메소드의 이름은 통상 동사를 사용



## 변수 이름

- 첫 단어의 첫 문자는 소문자로 쓰고 연속되는 단어의 첫 문자는 대문자  
`private Phone dayTimePhone;`  
`Address oldAddress = this.address;`
- 변수의 이름은 통사 명사를 사용
- 객체의 모임을 의미하는 변수는 복수형      `vector orderItems = new Vector();`
- 클래스 안에 포함된 필드 변수는 로컬 변수와 구별하기 위하여 'this'를 붙인다



### 상수 이름

- 대문자를 사용하되 단어 사이에는 밑줄로 연결

```
class Byte {  
    public static final byte MAX_VALUE = 255;  
    public static final byte MIN_VALUE = 0;  
    public static final Class TYPE = Byte.class;  
}
```

3.1415926535897932384626433832795028



- ❶ C++ 언어에서는 포인터를 매개변수로 사용하는 것을 피함

```
swap(int *x, int *y) {  
    int t;  
    t=*x;  
    *x=*y;  
    *y=t;  
}
```

```
swap(int &x, int &y) {  
    int t;  
    t=x;  
    x=y;  
    y=t;  
}
```

- ❷ 왼쪽 코드: 포인터를 사용하여 매개변수를 전달받아 계속 포인터를 사용
- ❸ 오른쪽 코드: 레퍼런스를 사용하면 포인터에 신경 쓰지 않고 일반적인 코딩하는 것처럼 가능



- ④ 제어흐름(if-else 등)을 나타내는 구조에서는 {...}으로 구성되는 블록 문장 사용하는 것이 보다 명확함
- ④ 괄호를 이용하여 오퍼레이션의 순서를 명확히 할 필요가 있음
- ④ 객체의 동일성을 테스트 할 때는 ==을 사용하는 것이 아니라 equal()을 사용해야 함
- ④ !=나 ==오퍼레이터는 객체의 동일성을 비교



- ④ 코드와 문서를 항상 일치시켜야 한다.
- ④ 불필요한 것을 생략하고 명확하고 간결하게 작성한다.
- ④ 각 주석은 고유의 목적을 가지고 있어 목적에 맞는 방법으로 사용한다.
- ④ 주석은 코드가 왜 그런 작업을 수행하고 있는지를 설명한다.



## In this chapter ...

코딩(Coding)

8.1 코딩 원리

8.2 코딩 스타일

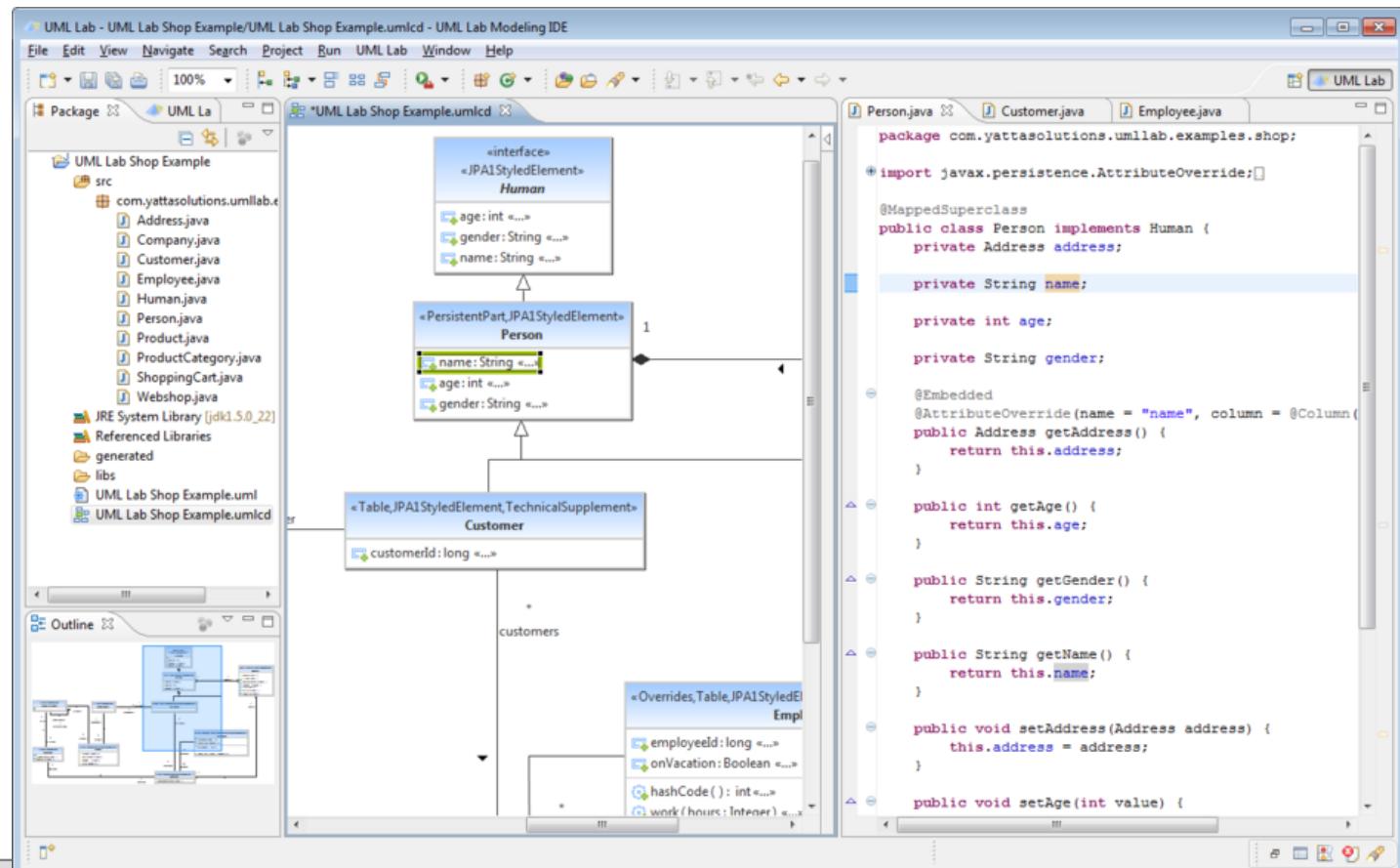
8.3 UML과 코딩

8.4 리팩토링

8.5 코드 품질 향상 기법



- StarUML을 비롯한 통합 개발 환경(IDE) 도구들은 UML 다이어그램으로부터 원시코드 골격으로 자동으로 생성
  - IDE도구에 따라 생성하는 원시코드가 다름

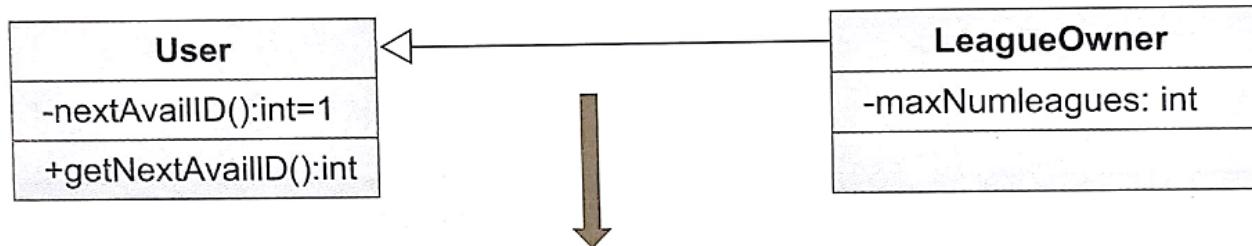




## 클래스와 인터페이스의 구현 (2/2)

코딩(Coding)

- 상속은 프로그래밍 언어 문법에 그대로 반영되므로 쉽게 구현할 수 있음



```
public class User {
    private String email;
    public String getEmail {
        return email;
    }
    public void setEmail (String value){
        email = value,
    }
    public void notify (String msg){
        //
    }
    /* Other methods omitted */
}
```

```
public class LeagueOwner extends User {
    private int maxNumleagues;
    public int getMaxNumleagues() {
        return maxNumleagues;
    }
    public void setmaxNumleagues(int value) {
        maxNumleagues = value,
    }
    /* Other methods omitted */
}
```

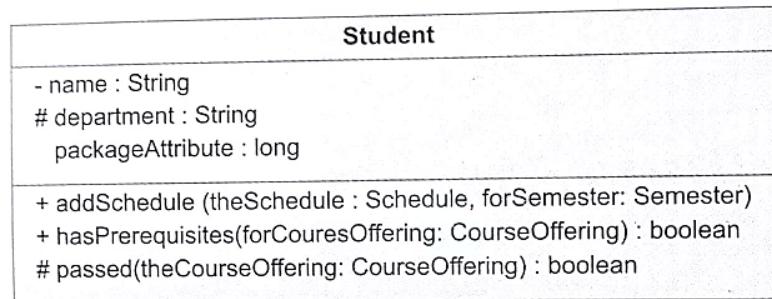


## 클래스와 인터페이스의 구현 (2/2)

코딩(Coding)

### 클래스 다이어그램의 클래스와 인터페이스를 C++, Java로 구현

- 속성: 클래스 안의 인스턴스 변수로 선언(+는 public, -는 private, #은 protected)
- 오퍼레이션: 메소드로 작성



```
public class Student
{
    private String name;
    protected String department;
    long packageAttribute;
    public void addSchedule (Schedule theSchedule; Semester forSemester) {
    }

    public boolean
        hasPrerequisites(CoursesOffering forCourseOffering) {
    }
    protected boolean
        passed(CoursesOffering theCourseOffering) {
    }
}
```



## 1대 1 연관

- A에서 B의 함수를 호출할 필요가 있다면 A가 B에 대한 참조를 갖도록 구현

## 1대 다

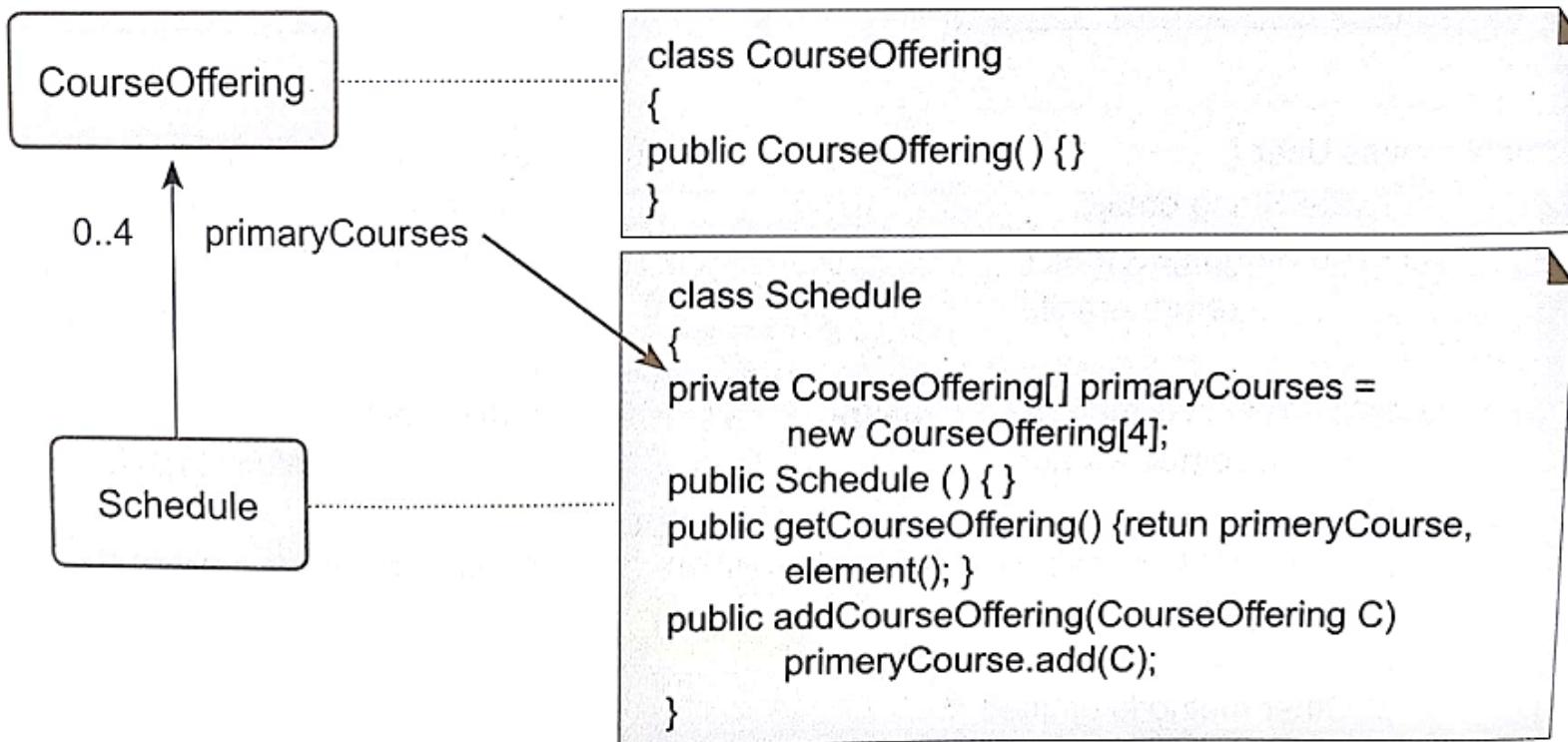
- 클래스 A에서 인스턴스 B의 메소드를 호출할 것이 있다면, 클래스A가 클래스B의 참조를 모음으로 가지고 있도록 구현
- 예제: 다음 페이지 그림에서 Schedule이 클래스 A, CourseOffering이 클래스 B에 해당

## 다대 다

- B 객체에 대한 참조 모임을 A가 갖게하고, 반대로 A 객체에 대한 참조 모임을 B 객체가 갖도록 함
- 중간에 연관 클래스를 도입하여 1대 다의 관계로 바꾸어 설계하기도함



- 1대 다 예제: 클래스 Schedule에서 클래스 CourseOffering 네 개에 대한 참조를 가지고 있음

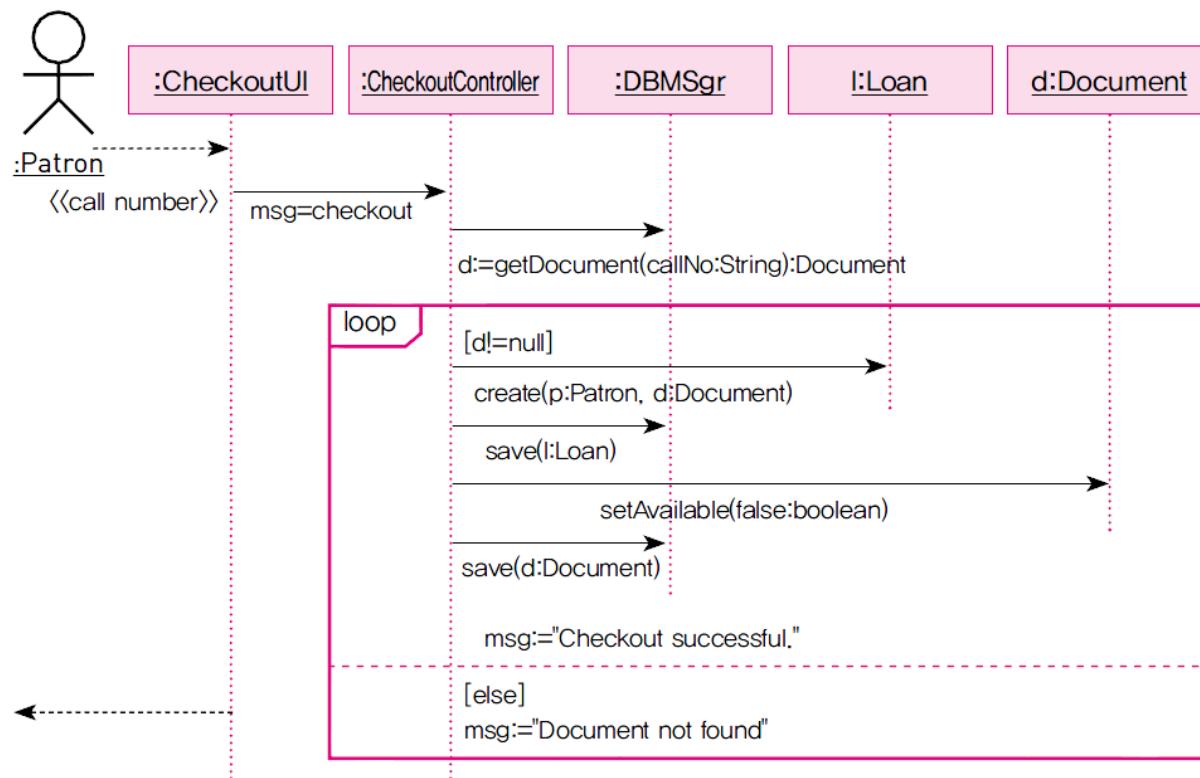




# 시퀀스 다이어그램의 구현 (1/2)

코딩(Coding)

- 시퀀스 다이어그램: 객체의 동작적인 측면을 모델링 한 것
- 오퍼레이션의 종류와 클래스 사이의 관계를 파악하기 위하여 각 객체가 주고받는 메소드를 나타냄 ( $\rightarrow$  결국, 메소드로 구현됨)
- 예제: 시퀀스 다이어그램의 CheckoutController의 구현





# 시퀀스 다이어그램의 구현 (2/2)

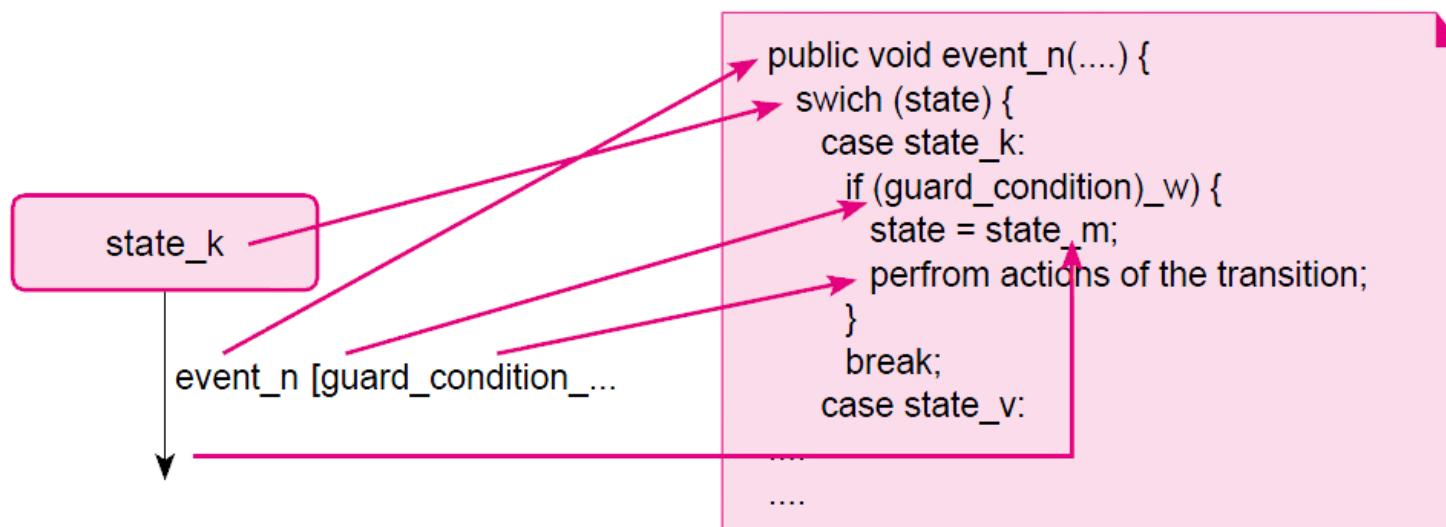
코딩(Coding)

- IDE가 원시 코드의 골격만 구현하며, 내부는 프로그래밍하여 완성

```
public class CheckoutController {  
    Patron p;  
    public String checkout(String callNo) {  
        BDNgr dbm=new DBMge();  
        Document d=dbm.getDocument(callNo);  
        String msg=" ";  
        if (d!=null) {  
            Loan l=new Loan(p, d);  
            dbm.save(l);  
            d.setAvailable(false);  
            dbm.save(d);  
            msg="Checkout successful.";  
        } else {  
            msg="Document not found.";  
        }  
        return msg;  
    }  
}
```



- 상태 – 상태정보를 저장하기 위한 속성을 클래스에 추가
- 이벤트 – 객체의 상태를 변화시키는 것 (메소드로 구현)
- 컨디션 – 상태변화를 일으키는 조건을 나타내는 것
- 액션 – 메소드 안에 탑재





## In this chapter ...

코딩(Coding)

8.1 코딩 원리

8.2 코딩 스타일

8.3 UML과 코딩

8.4 리팩토링

8.5 코드 품질 향상 기법



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

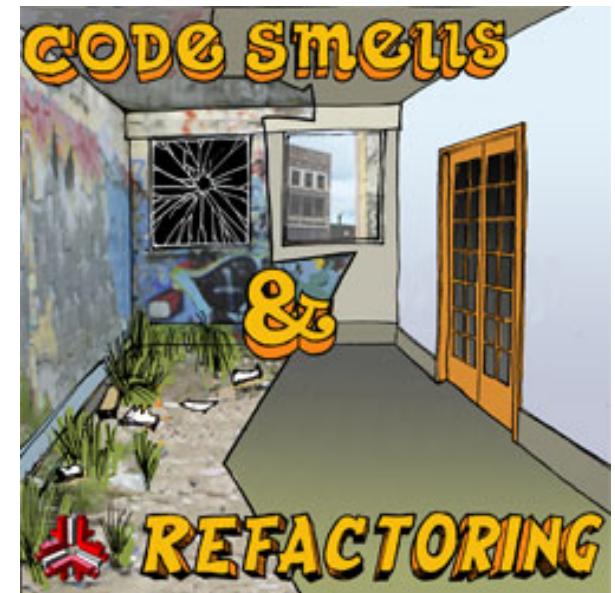
## 리팩토링 이유

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



### 코드 스멜 (Code Smell)

- 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





# 리팩토링 목적

코딩(Coding)

- 외부에서 보이는 소프트웨어 기능을 변경하지 않으면서도, 다음의 코드 개선 목적을 달성한다.
- 소프트웨어의 디자인을 개선시킨다.
- 소프트웨어를 이해하기 쉽게 만든다.
- 버그를 찾는데 도움을 준다.
- 프로그램을 빨리 작성할 수 있게 도와준다.



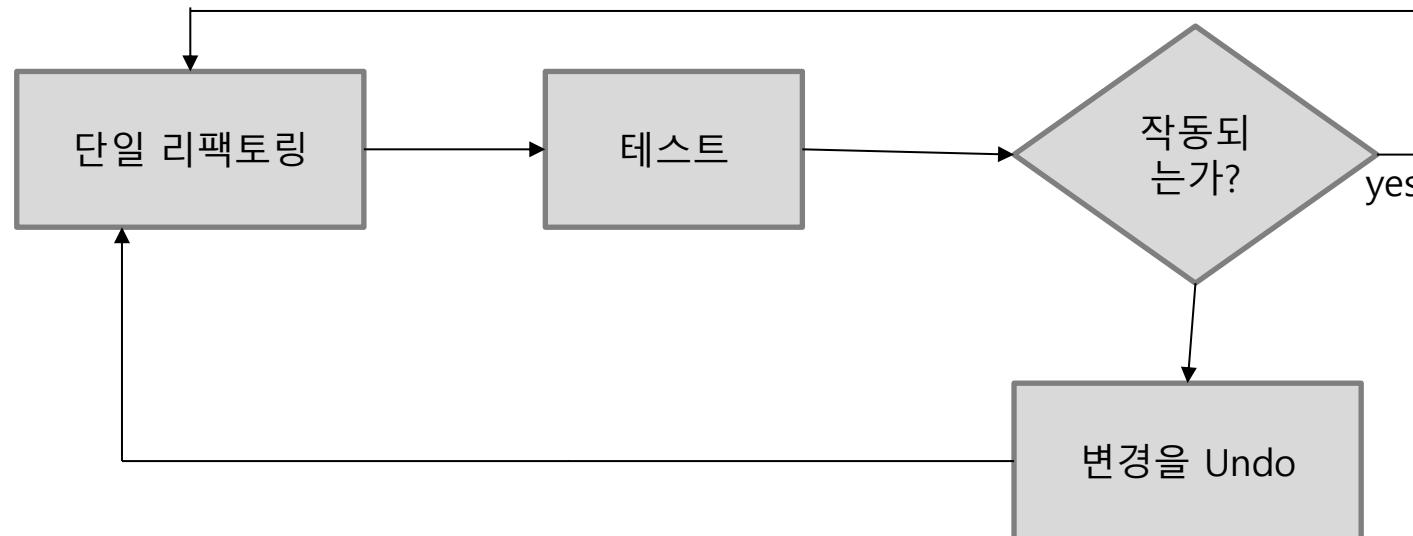
## 국립 대학교 스트링 str안의 문자 c가 포함되어 있는지 체크하는 부분을 별도의 메소드로 빼내어 리팩토링하는 사례임

```
void foo(char c) {
    int i, count, len
    char str[MAX];
    cin>>str
    len=strlen(str);
    count=0;
    for(i=0;i<=len;i++)
        if(str[i]==c) {
            count++;
            str[i]='\n';
        }
    cout<<str<<"<<count<<"\n";
}
```

```
void foo(char c) {
    int i, count, len
    char str[MAX];
    cin>>str
    len=strlen(str);
    newfun(count, len, str, c);
    cout<<str<<"<<count<<"\n";
}
newfun(int&count, int len, char*str, char c) {
    int i;
    count=0;
    for(i=0;i<=len;i++)
        if(str[i]==c) {
            count++;
            str[i]='\n';
        }
}
```



1. 소규모의 변경 – 단일 리팩토링
2. 코드가 전부 잘 작동되는지 테스트
- 3-1 전체가 잘 작동하면 다음 리팩토링 단계로 전진
- 3-2 작동하지 않으면 문제를 해결하고 리팩토링 한 것을 undo하여 시스템이 작동되도록 유지





## ④ 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 (2/2)

코딩(Coding)

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 탐색 선호 (Primitive Obsession)	객체 형태 그룹을 만들지 않고, 기본 데이터 탐색만 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



## In this chapter ...

코딩(Coding)

8.1 코딩 원리

8.2 코딩 스타일

8.3 UML과 코딩

8.4 리팩토링

8.5 코드 품질 향상 기법



## 코드 인스펙션

- 프로그램을 읽어보고 눈으로 확인하는 방법
- 품질 개선과 결함을 초기에 발견하기 위한 효과적인 방법

## 정적 분석

- 수행되지 않는 데드코드가 없는지, 선언이 되지 않고 사용한 변수가 없는지 등을 검사
- 소프트웨어 도구를 이용하여 자동으로도 가능

## 페어 프로그래밍

- 애자일 방법에서 프로그래밍과 테스팅을 담당하는 두 사람이 머신을 공유하며 코딩
- 프로그래밍에 재미를 더하고, 팀의 의사 소통을 향상 시킬 수 있음



## In this chapter ...

코딩(Coding)

8.1 코딩 원리

8.2 코딩 스타일

8.3 UML과 코딩

8.4 리팩토링

8.5 코드 품질 향상 기법



# Homework #4

코딩(Coding)