CHAPTER

09 코딩

| 키포인트 |

- 코딩 작업의 원리와 과정은 무엇인가?
- 코딩 단계의 과정과 흔히 일어나는 오류는 무엇인가?
- 코딩 스타일이란 무엇이며 좋은 코드가 되기 위해서 어떤 가이드가 필요한가?
- 리팩토링이란 무엇이며 코드 스멜, 리팩토링 방법은 무엇인가?
- 코드 품질을 높이는 방법에는 어떤 것이 있는가?

설계가 완성되면 코딩 단계가 시작된다. 각 모듈에 대한 원시 코드를 작성하며 문서화하는 단계이다. 코딩 작업은 설계서를 기본으로 한다. 코딩 단계의 완료에 대한 견해가 다를 수 있으나 일반적으로 모듈 안에 포함된 오류는 검출하고 테스트 단계로 넘어간다.

코딩 단계에 할애하는 시간은 다른 단계의 작업, 예를 들면 테스트나 유지보수보다 상대적으로 적다. 하지만 프로그래밍 작업의 결과는 소프트웨어의 품질에 미치는 영향이 매우 크다. 앞서 언급한 것처럼 코딩 작업이 끝난 후 오류를 수정하는 데 소요되는 비용이 매우 크기 때문이다. 따라서 코딩 단계에서의 목표는 구현 비용을 줄이는 것이라기보다 구현 비용이 늘어나더라도 코딩 이후의 비용을 줄이는 것이라고 할 수 있다.

설계를 원시코드로 바꾸는 작업에는 코딩 목표만이 아니라 다른 요소들도 영향을 줄 수 있다. 예를 들면 코딩에 적용되는 기초 원리들과 코딩 표준과 같은 것들이다. 특히 코딩도 여러 사람의 협력 작업이므로 서로의 코딩 스타일을 통일시킬 필요가 있다. 또한 객체지향 설계 방법에서 UML로 표현된 설계를 어떻게 객체지향 프로그래밍 언어로 옮기는지도 알아야 한다.

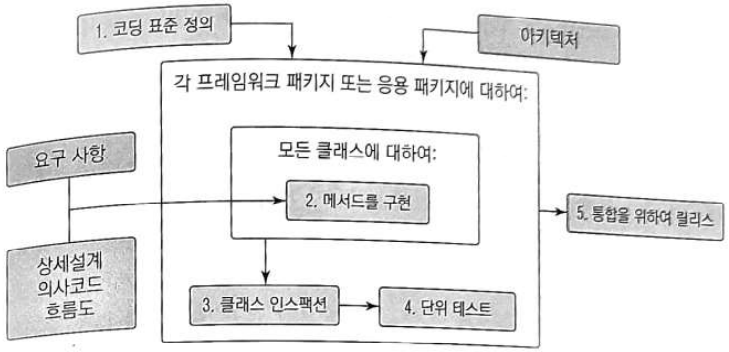

그림 9.1 코딩 작업의 로드맵

코딩 단계의 작업은 설계한 기능을 프로그램으로 구현하는 일에만 초점을 두지 않아야 한다. 가독성을 높이고 유지보수를 편하게 하는 관점으로도 작업할 필요가 있다. 코드의 기능이나 성능보다는 구조를 조명함으로써 문제가 될 가능성이 있는 부분(이를 코드 스멜이라 부름)을 찾아내고 개선하는 작업을 리팩토링이라 한다. 코딩 단계에는 구현뿐만 아니라 리팩토링 작업도 필요하다.

소프트웨어 제품의 품질은 결국 원시 코드에 모두 귀결된다. 원시 코드 이외의 설계 문서와 매뉴얼도 소프트웨어의 품질을 결정하는 결과물이지만 실행되는 원시 코드가 품질을 결정하는 핵심이다. 따라서 코딩 단계에서는 품질을 향상시키기 위한 다각도의 노력이 필요하다. 예들 들면 동료의 코드를 리뷰하거나 인스펙션 하는 작업, 정적으로 분석하는 작업, 또는 페어 프로그래밍이나 테스트 중심의 개발 기법들이 여기에 해당된다.

이 장에서는 프로그래밍 작업에서 알아야 할 기초, 즉 흔히 일어나는 코딩 오류를 설명한다. 다음에는 좋은 프로그램을 작성하기 위한 코딩 표준을 알아보며 코딩 단계에 이루어지지만 코딩과는 다른 작업인 리팩토링에 대하여 소개한다. 마지막으로 코딩의 품질을 높이기 위한 여러 가지 방안에 대하여 기술하였다.

9.1 코딩 작업

코딩 작업이란 프그래밍을 의미한다. 특히 분리하여 구현할 수 있는 작은 단위를 프로

그래밍 하는 작업이다. 절차적인 방법에서의 코딩은 주로 모듈 안의 함수 내부를 완성하는 작업이라고 볼 수 있다. 객체지향 방법은 분석 및 설계 단계에서도 상당한 부분의 코딩 작업을 할 수 있고 구현 단계에서 이루어지는 작업은 개별 메서드의 코딩이다.

구현 단계의 목표는 설계 명세에 나타낸 대로 요구를 만족할 수 있도록 프로그래밍 하는 것이다. 프로그래밍의 수행이 상세 설계나 사용자 지침서에 기술된 것과 일치되도록 코딩하여야 한다. 이렇게 하기 위하여 코딩 단계에는 전 단계의 문서들, 예를 들면 아키텍처 설계서, 요구 분석서 등을 잘 참조하여야 한다.

코딩 작업에서는 무엇보다도 오류가 적은 품질 좋은 프로그램을 작성하는 데 목표가 있다. 신속히 작성하여야 하는 것은 이차적인 목표다. 견고한 코드를 작성하는 것은 연습이 많이 필요하다. 하지만 경험에 의하여 좋은 프로그램을 작성하기 위한 원리와 가이드가 정리되어 있다. 정확하고 간결한 좋은 프로그램을 작성하는 것은 프로그래밍 언어의 선택과 관계가 없다.

최근의 프로그래밍 언어들은 이미 객체지향 설계 방안을 고려하여 설계되었기 때문에 설계에서 프로그래밍 언어로 자동으로 일부를 생성한다. 이 절에서는 프로그래밍 작업의 기초가 되는 코딩 과정과 흔히 일어나는 코딩 상의 오류에 관하여 알아본다.

9.1.1 작업 과정

객체지향 방법에서 구현 작업은 프로그램 안에 이미 결정된 요소가 많아 메서드 안으로 국한된다. 분석 단계의 결과인 요구 사항과 설계 단계에서 완성된 모델 및 아키텍처, 상세 설계에서 파악한 알고리즘을 작성한 의사코드나 흐름도를 가지고 다음과 같은 순서로 작업한다.

1. 원시코드를 같은 스타일로 만들기 위하여 코딩 표준을 만든다.
2. 아키텍처 설계 결과 프레임워크 패키지와 응용 패키지를 결정한다. 패키지 안에 있는 각 클래스에 대하여 요구 사항과 상세 설계를 반영하여 메서드를 코딩한다. 프레임워크 패키지는 응용 패키지가 완성되기 전에 구현되어야 한다.
3. 클래스 구현이 끝나는 대로 인스펙션 한다.
4. 클래스 단위로 테스트한다.
5. 클래스나 패키지를 릴리스 하여 다음 단계인 응용 시스템으로 통합하도록 한다.

코딩 단계에 들어가기 전 준비하여야 할 사항이 있다. 먼저 구현할 자세한 설계를 확인한다. 설계는 코딩에서 건축시공을 위한 도면과 같은 것이므로 철저히 설계서를 바탕으로 작업하여야 한다. 이때 필요한 지식이 UML 설계와 코딩의 매핑과정이다. UML의 대부분의 설계 요소는 프로그램 안에 있는 구문이나 구조로 구현된다. 예를 들어 클래스 다이어그램의 연관이나 집합 관계를 어떻게 코딩하여야 하는지 알아야 한다.

코딩 단계에 가장 중요한 작업 중의 하나는 요구하는 코딩 표준을 이해하는 것이다. 명명 규칙, 들여쓰기 규칙 등 여러 사람이 작성하는 코드를 통일하려는 목적으로 만든 가이드를 미리 숙지한다. 또한 과거 자료를 근거로 원시코드의 크기와 소요 시간을 예측하고 코딩 작업을 팀에 할당하여 진행한다.

9.1.2 자주 발생하는 오류

프로그래머는 누구든 오류를 잘 알고 고칠 수 있어야 한다. 코딩할 때 많은 시간을 버그를 찾고 제거하는 데 사용한다. 버그를 줄이기 위하여 여러 가지 방법과 도구가 사용되고 있지만 여전히 프로그램에 버그가 발견된다. 오류가 다양한 방법으로 발견되지만 흔히 발견되는 오류는 정해져 있다. 프로그래머가 이런 오류들을 피하기 위하여 사전에 알아두는 것이 좋다.

■ 메모리 누수

메모리 누수(leak)란 사용한 메모리를 free 시키지 않고 새로 계속 할당 요구하여 메모리가 고갈되는 현상을 말한다. 가비지 콜렉션이 자동으로 되지 않는 C나 C++ 언어에서는 이러한 오류가 자주 발생한다. 메모리 누수는 짧은 프로그램에는 영향이 적지만 장기로 수행되는 시스템에는 치명적인 영향을 줄 수도 있다. 메모리 누수가 일어나는 소프트웨어 시스템은 결국은 메모리가 고갈되어 비정상적으로 정지하게 된다. 다음과 같은 예를 살펴보자.

```
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);
}
```

■ 중복된 프리 선언

일반적으로 프로그램 안에서 사용하는 자원은 먼저 할당되고 사용 후에는 프리로 선언한다. 예를 들면 메모리가 할당된 후에는 소멸시킨다. 이미 프리로 선언된 자원을 또 다시 프리로 선언하는 경우 오류이다. 이런 오류의 파급 영향은 매우 심각하다. 예를 들면 다음과 같다.

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

두 개의 free 문장 사이에 메모리 할당 호출이 있다면 이런 오류의 영향은 매우 심각하다. 왜냐하면 첫 번째 프리된 곳이 새 변수에 할당되어 다음의 프리 선언으로 할당된 메모리가 소멸되기 때문이다.

■ NULL의 사용

NULL을 포인트 하고 있는 곳의 콘텐트를 접근하려고 하면 오류가 된다. 흔히 발생하는 이러한 오류는 시스템을 다운시킨다. NULL을 포인터로 접근하는 것이 특정한 경로나 환경에서만 일어난다고 하면 찾아내기가 어렵다. 여러 경로에 걸쳐 초기화되어 적절하지 못한 경우 NULL을 접근하는 오류를 범하게 된다.

별칭(alias) 때문에 그렇게 될 수도 있다. 예를 들어 두 개의 변수가 동일한 객체를 참조하다가 하나가 프리 되었을 때 다른 하나를 이용하여 접근을 시도하면 오류가 된다. 다음의 예에서 두 번의 NULL 접근 오류를 찾아볼 수 있다.

```
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
```

NULL 접근 오류와 유사한 것으로 초기화 되지 않은 메모리를 접근하는 오류가 있다. 초기화를 한다고 했지만 이런 오류가 발생하는 경우도 있는데 다음의 코드가 그 사례를 보여주고 있다.

```
switch ( i )
{
  case 0:  s = OBJECT_1; break;
  case 1:  s = OBJECT_2; break;
}
return (s); // s not initialized for values other than 0 or 1
```

■ 별칭의 남용

별칭(alias)은 많은 문제를 야기할 수 있다. 서로 다른 주소 값을 예상하고 사용한 두 개의 변수의 값이 별칭 선언으로 인하여 같은 값이 되었을 때 오류를 발생한다. 예를 들면 strcat(src, destn)과 같은 스트링의 결합 프로그램에서 src와 destn의 주소 값이 다를 것으로 예상하고 프로그래밍 하였지만 src가 destn의 별칭으로 선언되어 있다면 심각한 런타임 오류를 초래할 수 있다.

■ 배열 인덱스 오류

배열의 인덱스가 한도를 벗어나면 예외 오류가 발생한다. 배열의 인덱스가 음수 값을 갖지 않는지 혹은 한도를 벗어나지 않는지 잘 점검하여야 한다. 다음 코드에서 i의 값이 80이 되면 한도 값 79를 벗어나 오버플로가 발생한다.

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

■ 수식 예외 오류

0으로 나누는 오류, 변동 소수점 예외 오류 등이 여기에 해당된다. 이러한 오류들은 예상하지 못한 결과를 내기도 하며 프로그램이 정지될 수도 있다.

■ 하나 차이에 의한 오류

프로그래밍 경험이 많이 있다면 이런 오류가 여러 형태로 발생한다. 예를 들어 0으로 시작하여야 할 것을 1로 한다든지, <= N으로 써야 할 곳에 < N을 쓴 경우에 발생한다.

아주 흔한 예로 두 수를 비교하는 의미를 if (number = SSN) | .... 이라고 썼을 때 컴파일러는 전혀 오류로 인식하지 못하고 지나간다. 원래 의도와는 다르게 SSN의 값을 number에 배정하여 그 값이 0인지 아닌지를 비교하기 때문이다.

이렇게 하나 차이에 의한 오류는 컴파일러나 테스트 도구에 의하여 검출되지 않고 지나가는 경우가 많다. 하나 차이에 의한 간단한 오류지만 프로그래머가 며칠간의 밤을 새우게 할 수도 있다.

■ 사용자 정의 자료형 오류

사용자 정의 자료형을 다룰 때 오버플로나 언더플로 오류가 쉽게 발생할 수 있다. 사용자 정의 자료형의 값을 다룰 때는 특별히 주의 하여야 한다. 예를 들어 다음 프로그램을 살펴보라.

```
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
}
```

■ 스트링 처리 오류

프로그래밍 언어에는 strcpy, sprintf 등 많은 스트링 처리 함수가 있다. 이들을 사용할 때 매개변수가 NULL이거나 혹은 스트링이 NULL로 끝나지 않거나 destination 매개변수의 크기가 충분히 크지 않을 경우 버퍼 초버런(overrun)오류가 발생한다. 스트링 처리 오류는 빈번히 발생한다.

■ 버퍼 오버플로우 오류

버퍼 오버플로는 소프트웨어 고장을 일으키는 흔한 원인이다. 하지만 최근에는 해커들이 자신의 코드를 실행시키기 위하여 이용하는 보안 결함을 유발시킬 수 있다.

프로그램이 버퍼에 복사하여 입력받으려 할 때 입력 값을 고의로 아주 크게 주면 스택의 버퍼에 오버플로가 일어난다. 이 때 리턴 주소가 반환되는 데 이를 이용하여 해커가 원하는 곳으로 가게 하여 정보를 무단으로 수정 또는 이용하여 해를 입히거나 컴퓨터를 마음대로 제어하는 것이다. 즉 버퍼 오버플로를 이용하여 해커들이 자신의 코드를 실행시킬 수 있다. 다음의 코드 조각은 버퍼 오버플로를 보여준다.

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

위 코드는 버퍼 오버플로를 이용하여 공격할 수 있다. 입력 값이 매우 크다면 버퍼 s2가 오버플로가 될 것이며 스택에 올라가 있는 mygets() 함수의 리턴 주소를 교묘히 조작하여 해킹 프로그램의 주소로 바꾸어 놓을 수 있다.

■ 동기화 오류

공통 자원을 접근하는 다수의 스레드가 있는 병렬 프로그램에서는 동기화 오류가 흔하다. 이런 오류들은 쉽게 발견되지가 않는다. 그러나 이런 오류가 발생되었을 때에는 시스템에 심각한 지장을 초래한다. 동기화 오류에는 다음과 같은 종류가 있다.

- 데드락(deadlock) – 다수의 스레드가 서로 자원을 점유하고 릴리스 하지 않는 상태. 대부분 록킹에 모순이 있는 경우에 발생한다. 예를 들어 다른 스레드가 점유하고 있는 자원을 또 다른 스레드가 기다리고 있는 경우.
- 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 발생한다.

9.2 코딩 표준

스타일이란 어떤 작업이나 선택에 있어서 일관된 유형을 말한다. 예를 들어, 옷을 입는 데도 스타일이 있다. 여러 선택 가능한 옷 중에 일관된 유형의 옷을 선호하여 착용하는 것을 볼 수 있다. 물론 스타일은 필요에 따라 바뀌질 수도 있고 훈련될 수도 있다.

프로그래머도 코딩 스타일이 있다. 같은 작업을 위하여 여러 사람이 작성한 프로그램들은 문장의 패턴이나 그 구성 등 여러 면에서 다른 스타일을 보인다. 프로그래밍 스타일은 가르칠 수 있으며 대부분의 교과서나 수업에서의 예제 프로그램을 통하여 습득하고 있다.

좋은 코딩 스타일이 무엇을 의미하는지 정의하는 것은 어렵다. 그러나 좋은 스타일인지 판별하는 대표적인 기준은 간결하고 읽기 쉬워야 한다. 간결함은 복잡하지 않고 명확하여 이해하기 쉬운 것이다. 프로그램을 이해하는 데 많은 노력이 필요하며, 고치기 힘들다면 간결한 프로그램이 아니다.

읽기 쉽다는 것은 프로그램을 대충 훑어보거나 이해하기 쉽다는 것을 의미한다. 물론 프로그램이 간결하면 읽기 쉽지만 또 다른 요소들, 문형 구조, 원시 코드의 편집 상태 등에 의하여 가독성이 달라진다. 쉽고 간결한 코드를 작성하기 위한 절대적인 규칙은 없으나 일반적인 원칙은 다음과 같다.

원시 코드의 간결성은 시스템이 얼마나 잘 설계되었는가에 좌우된다. 설계에서 모듈화의 목표, 높은 응집력, 낮은 결합도를 달성하였다면 모듈은 간결해진다. 또한 다음에 설명하는 원칙들을 지켜 나간다면 좋은 프로그래밍 스타일을 유지할 수 있을 것이다.

9.2.1 명명 규칙

프로그램 안에 있는 여러 요소들의 이름을 붙이는 것은 매우 중요하다. 의미 있는 이름을 붙여야 함은 물론이고 그 이름만 보아도 그것이 클래스인지 아니면 멤버 함수인지 아니면 상수인지 알 수 있도록 붙여야 한다. 패키지, 클래스, 인터페이스, 메서드 등 객체지향 프로그래밍 안에 있는 여러 요소에 대하여 이름 붙이는 방법을 소개한다.

■ 카멜 케이스

Java 클래스, 필드, 메서드 및 변수 이름은 카멜 케이스로 작성한다. 즉 여러 단어를 함께 붙여 쓰되 각 단어의 첫 글자는 대문자로 쓴다. 밑줄 _과 같은 단어 구분 기호는 사용하지 않는 것이 좋다.

```
thisIsAnExample
```

CamelCase에 대한 예외가 있는데 상수다. 상수는 모두 대문자로 표시하되 단어는 밑줄로 구분한다. 상수는 final 키워드를 가져야하며 일반적으로 static으로 선언한다.

```
public static final double SPEED_OF_LIGHT= 299792458; // in m/s
```

■ 클래스와 인터페이스 이름

클래스 및 인터페이스 이름은 일반적으로 명사 또는 명사구이며 대문자로 시작한다.

```
interface AqueousHabitat { ... }
class FishBowl implements AqueousHabitat { ... }
```

■ 메서드 이름

메서드 이름은 일반적으로 소문자로 시작한다. 프로시저 호출은 무엇이든지 실행하라는 명령문이므로 프로시저 이름은 일반적으로 동사구로 표현한다.

```
public void setTitle(String t) { ... }
```

함수 호출은 값을 생성하므로 함수 이름은 일반적으로 값을 설명하는 명사구로 표현한다.

```
public double areaOfTriangle(int b, int c, int d) { ... }
```

Java의 명명 규칙은 필드(예를 들어 title)의 값을 접근하여 리턴하는 함수는 "get"으로 시작하며 뒤에 필드 이름을 붙인다. 이런 규칙에 대하여 좋은 방법이 아니라고 생각하는 이견이 있을 수 있다. 함수 이름을 "title"이라고 사용할 수 있다.

```
public String getTitle() { ... }
```

조건을 묻는 부울 함수의 이름은 대개 "is"로 시작하는 동사구로 값의 의미나 조건이 맞는지를 설명한다.

```
Public boolean isEquilateralTriangle(int b, int c, int d) { ... }
```

■ 변수 이름

변수 이름은 일반적으로 소문자로 시작한다.

변수 이름은 프로그램을 읽는 사람에게 용도에 대한 힌트를 제공해야 한다. 변수의 의미에 대한 힌트를 제공하는 잘 선택된 이름은 프로그램을 문서화하여 이해하기 쉽도록 도와준다. 한편, 변수 이름을 친구, 꽃 이름 등을 사용하면 프로그램을 이해하기가 더 어려워진다. 또한 counter 또는 var 또는 data와 같은 모호한 이름을 사용하지 않아야 한다. 대신에 변수가 실제로 무엇인지 생각하고 더 구체적인 이름을 사용하여야 한다.

긴 변수 이름과 짧은 변수 이름을 쓰는 것 사이에는 갈등할 수 있다. 이름이 길면 변수의 용도를 쉽게 기억할 수 있지만 길이가 길면 전체 프로그램이 길고 복잡해 보일 수 있다. 아주 짧은 이름은 의미를 이해하기 어려울 수 있지만 프로그램을 짧게 만든다. 예를 들어, 변수 이름의 길이만으로 인해 다르게 보이는 다음 문장을 비교해 보라.

```
hypotenuseOfTriangle= 6 * (2 + hypotenuseOfTriangle) + 7 / hypotenuseOfTriangle
                - hypotenuseOfTriangle;
x= 6 * (2 + x) + 7 / x - x;
```

변수 이름은 이름을 가진 대상 자체만을 고려하면 완전하고 정확한 정의를 제공하지는 못한다. 대상이 사용된 위치를 고려하여 완전한 정의를 제공해야 한다.

여기에 해법이 있다. 매개 변수와 지역 변수에 더 짧은 이름을 사용하고 필드와 정적 변수에 더 긴 이름을 사용하는 경향이 있다. 이유는 다음과 같다.

- 매개변수 이름 - 메서드의 정의는 모든 매개 변수의 이름을 지정하고 의미를 제공한다. 메서드 본문은 일반적으로 30~50줄 정도로 상당히 짧다. 따라서 메서드 본문을 읽을 때 매개 변수의 의미는 스크롤 없이 볼 수 있다. 따라서 매개 변수 이름은 짧을 수 있다.

```
/** Return "lengths b, c, and d are the sides of an equilateral triangle" */
public boolean isEquilateralTriangle(int b, int c, int d) {
   return b == c && c == d;
}
```

로컬 변수는 메서드 본문에 선언된 변수다. 변수 선언은 가능한 한 처음 사용하는 것에 가깝게 배치해야한다. 로컬 변수의 범위는 일반적으로 짧으며 그 의미는 선언에 대한 주석이나 그것이 사용되는 짧은 코드를 보이야 일 수 있기 때문이다. 따라서 로컬 변수의 이름이 짧을 수 있다.

- 필드 변수 - 클래스의 인스턴스 또는 클래스 변수의 선언은 사용된 곳과 수백 줄이 떨어져있을 수 있다. 또한, 필드와 클래스 변수의 의미는 일반적으로 변수가 사용되는 곳과는 거리가 먼 선언에서 주석으로 제공된다. 따라서 필드 및 클래스 변수의 이름은 가능한 길고 의미가 담겨 있어야 하며 읽는 사람에게 의미가 무엇인지에 대한 좋은 아이디어를 제공해야한다.

■ 패키지 이름

패키지 이름은 일반적으로 모두 소문자이며 명사로 정한다.

헝가리안 표기법

헝가리어 표기법은 1972년 Xerox PARC의 프로그래머였고 Microsoft의 수석 아키텍트가 된 Charles Simonyi가 발명하였다.

헝가리 사람들의 이름은 대부분의 다른 유럽 이름과 다르게 성이 이름보다 먼저 온다. 초창기 객체지향 언어인 Smalltalk 네이밍 스타일과 같이 변수 이름 앞에 타입의 구별이 앞에 나온다. 예를 들면 다음과 같다.

- 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

헝가리어 표기법은 실제 데이터 형식이 아닌 논리 데이터 형식에도 적용할 수 있다. 아래와 같이 변수의 목적 또는 변수가 무엇인지에 대한 힌트를 주는 방법이다.

- rwPosition : 행("rw")을 나타내는 변수
- usName : 사용되기 전에 정제할 필요가 있는 안전하지 않은 스트링("us")
- szName : '\0'로 끝나는 스트링("sz") 변수

헝가리어 표기법은 모든 프로그래밍 언어 및 환경에 적용될 수 있지만 C 언어, 특히 Microsoft Windows 용으로 Microsoft에서 널리 채택했으며 그 사용이 Windows API 프로그래밍에 대한 소개 책에 의해 널리 전파되었다.

9.2.2 형식

클래스 내에서 일관된 형식에 대한 규칙을 사용하여야 한다. 예를 들어 열린 괄호 "("

의 위치는 프로그램 전체에서 동일해야 한다. 한 줄에 한 줄만 입력한다. 하나 이상이 더 읽기 좋은 예외적인 경우가 있을 수 있다. 하지만 프로그램을 읽기 어렵게 만들기 위해 모든 것을 함께 묶지 말아야 한다.

가로 스크롤 없이 모든 행을 읽을 수 있는지 확인하라. 한 줄에 최대 80자를 넘지 않아야 한다.

■ 들여쓰기와 괄호

들여 쓰기는 프로그램 구조를 명확하게 하기 위해 사용된다. 기본 규칙은 다음과 같다.

문장의 일부와 선언문은 들여 쓰기를 해야 한다. 예를 들어, 클래스의 선언, 메서드의 본문, 조건문의 if-part 및 then-part는 들여쓰기 해야 한다. 대개 2가 아닌 4 개의 공백 들여 쓰기를 선호한다. Eclipse 같은 IDE에서는 메뉴 항목 환경 설정 → Java → 코드 스타일 → 포멧터를 사용하여 들여 쓰기로 빈칸의 크기를 설정할 수 있다.

IDE를 사용하면 전체 클래스를 일관되게 들여쓸 수 있다. 프로그램 문장을 선택하고 control-i를 누르면 일괄로 들여쓰기를 해준다.

아래에 표시된 대로 줄 끝에 여는 중괄호 "{"를 사용하는 것이 좋다. 닫는 괄호 "}"는 아래에 표시된 대로 자기 자리에서 들여쓰기 하면 된다.

```
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;
}
```

위와 같은 규칙을 따르지 않는다면 아래와 같이 오프닝 브레이스를 한 줄에 두게 된다. 이것은 단점이 있다. 한 번에 모니터에서 볼 수 있는 줄 수는 제한적이며 부족한 자원을 낭비하는 규칙은 좋지 않기 때문이다.

```
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;
}
```

두 번째 규칙을 사용하는 데 익숙하다면 디스플레이 자원을 절약하기 위하여 첫 번째 규칙을 사용하는 것이 좋다.

■ 블록은 항상 괄호를 사용

다음 코드는 버그를 유발할 수 있는 형식이다.

```
if (flag)  validate();
```

다음과 같이 썼다면 여기서 버그를 발견할 수 있는가?

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

의도가 두 개의 문장이 모두 if 제어 구조에 포함된 문장이라면 잘못 표현된 버그다. 제어 구조에는 항상 괄호를 사용한다.

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

이 규칙은 if-else 문 for 루프, while 루프와 같은 구조에도 적용된다. 코딩에서 중괄호가 빠지는 경우가 많고 이는 매우 많은 컴파일 오류를 유발한다.

키워드와 다음 괄호 사이에 공백을 두어 제어 구조와 메서드 호출과 구별하게 하여야 한다. 아래와 같이 if 키워드와 시작 괄호를 띄어 쓰면 메서드 호출과 구별하기가 쉽다.

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

9.2.3 문장과 수식

프로그램에서 되풀이되는 문장이나 수식은 메서드나 클래스로 패키지화 한다. 이렇게 하면 이해하기도 쉽고 변경이 로컬화 되어 유지보수 및 테스트에 드는 노력을 덜 수 있다.

■ 블록 문장

제어흐름을 나타내는 구조에서는 수식을 사용하는 것보다 블록 문장을 사용하는 것이 좋다. Java 블록 문장은 여러 개의 문장을 단일 복합 문장으로 취급하기 때문에 블록 문장이 어디에서나 보통 문장과 같이 사용될 수 있다.

블록 문장은 제어구조가 중첩되었을 때 생기는 혼란을 줄일 수 있으며 원시코드의 이해를 높이기 위한 방법을 제공한다. 다음의 코드 조각은 잘못된 들여쓰기 때문에 else 부분이 첫 번째 if와 관련된 것으로 착각하기 쉽다.

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

Java 언어에서는 이를 dangling else 문제라 부르는데 다음과 같이 블록 문장을 사용하면 이 문제를 없앨 수 있다.

```
if (x >= 0) {
  if (x > 0) positiveX();
}
else {
  negativeX(); // This is what we really wanted!
}
```

■ 수식

괄호를 이용하여 오퍼레이션의 순서를 명확히 할 필요가 있다. 긴 수식에서 오퍼레이션의 순서가 항상 분명하지는 않다. 작성하는 입장에서 순서가 분명하다 할지라도 읽는 사람을 위하여 명확히 한다.

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


Java 언어의 수식사용에서 주의할 것 중의 하나는 ==의 사용이다. 객체의 동일성을 테스트할 때는 ==을 사용하는 것이 아니라 equals()을 사용하여야 한다. Java에서 date와 string을 다루는 다음 예에 이런 잘못이 있다.

```
Date today = new Date();
while (date != today) {
  ...
}
String name;
...
if (name == "Bob") {
  hiBob();
}
```

Java에서 !=나 == 오퍼레이터는 객체의 값을 비교하는 것이 아니라 객체의 동일성을 비교한다. 실제 스트링의 값을 비교하려면 equals 메서드를 사용하여야 한다.

```
Date today = new Date();
while (!date.equal(today)) {
  ...
}
String name;
...
if ("Bob".equals(name)) {
  hiBob();
}
```

name.equals("Bob")이라고 쓰지 않고 "Bob".equals(name)이라는 수식을 사용하는 이유는 name이 null일 때 예외를 발생하지 않기 때문이다.

9.2.4 오류처리

소프트웨어를 개발하는 사람들은 항상 잘못된 데이터를 어떻게 다룰 것인가 하는 문제가 큰 이슈이다. 잘못된 데이터란 예를 들어 은행에 실제로는 없는 계좌번호 같은 것이다. 구현을 가능하면 간단하게 만들려고 해도 실제는 간단하지 않다. 프로그래밍의 대부분은 오류를 처리하기 위한 것이라 해도 과언이 아니다. 따라서 체계적인 접근 방법이 필수적이다.

■ 매개변수 오류

예를 들어, evaluate()라는 메서드가 매개변수로 'car', 'truck', 'bus'만을 받아들인다면 매개변수로 스트링 타입을 사용하지 않는 것이 바람직할 것이다. 그 이유는 잘못된 매개변수가 전달될 가능성이 있기 때문이다. 따라서 private 생성자나 팩토리 메서드의 리턴 타입은 SpicializedVehicle과 같은 클래스 타입으로 정의하고,

```
SpecializedVehicle createACar()
SpecializedVehicle createATruck()
SpecializedVehicle createABus()
```

함수 evaluate()는 다음과 같이 스트링 타입의 매개변수를 가진 함수 대신에

```
evaluate( String vehicleP )  // problem with illegal string
```

다음과 같은 확실한 매개변수를 가진 함수를 사용하는 것이 좋다.

```
evaluate( SpecializedVehicle vehicleP ) // parameter value cannot be illegal
```

오류 데이터를 타입 제한으로 미리 배제하는 방법 이외에 입력이 처리되기 전에 미리 데이터의 소스와 상호 작용하여 정상적인 것으로 바꾸어 놓는 방법이 있다. UI 프로그래밍에서 이런 방법을 동원하여 정상적인 입력만 받아들일 수 있다.

■ 입력 오류

텍스트 필드에서 'car', 'truck', 'bus'만을 입력할 수 있는 스트링 타입이라면 잘못된 입력을 방지하는 것은 쉬운 일이다. 리스트 박스가 이런 목적으로 사용되는 UI 요소다.

하지만 이상한 오류가 들어올 수도 있다. 예를 들어 사용자가 생년월일에 1/1/80과 나이에 30으로 입력하였다면 일관성 체크가 가능하다. 그러나 설계하는 입장에서는 모든 가능한 일관성과 경계 체크로 상당한 부담이 된다. 체크를 완벽하게 하기 어렵고 많은 필드가 개입되어 있기 때문에 쉬운 일이 아니다. 자료를 공급하는 외부 요소가 별도의 애플리케이션이라면 오류의 가능성은 더욱 많아진다.

예를 들어 심장의 기능을 모니터링 하여 환자에게 산소 공급을 제어하는 애플리케이션이 있다고 하자. 메서드 process(int measurementType, ...)를 코딩하는데 measurementType은 양의 정수이어야 한다. 메서드 안에서 양수인지 또한 범위 안의 값인지 체크하는 부분이 없다면 이상한 값이 들어와도 통과된다.

애플리케이션 내부의 간단한 코드를 놓쳐서 잘못된 입력이 전달되어 시스템 전체가 다운되게 할 수는 없다. 따라서 코드에서는 입력을 체크한 후 다음과 같은 방식으로 프로그래밍 되어야 한다.

먼저 파라미터의 안전한 디폴트 값을 지정하고, 전체 애플리케이션을 디폴트 오퍼레이션 모드로 설정한 후 예외처리를 위하여 throw 문장을 작성하여 호출자에게 잘못된 입력에 대한 처리 책임을 넘긴다. 그러면 애플리케이션은 오류 데이터 때문에 정지되지 않고 경고 메시지를 보낸다.

9.2.5 주석

프로그램에 주석, 커멘트를 다는 이유는 두 가지가 있다.

먼저 프로그래머는 프로그램을 작성하고 디버깅하는 동안 얻을 수 있는 모든 도움이 필요하다. 즉, 코딩, 디버깅하는 중간에 주석을 작성해야 한다. 프로그램이 완료된 후까지 연기해서는 안된다. 대부분 끝까지 기다렸다가 주석을 단다. 이렇게 되면 더 많은 오류가 발생하고 프로그래밍에 더 많은 시간이 걸리며 프로그램에 더 많은 버그가 생길 수 있다. 필드를 선언 할 때 필드를 문서화하고 본문을 작성하기 전에 메서드를 지정하는 습관을 가져야 한다. 메서드가 수행해야 할 작업을 변경하기로 결정한 경우 먼저 명세를 수정하라.

다른 사람들이 프로그램을 이해하기 쉽도록 프로그램을 잘 문서화하여야 한다. 전문 프로그래머는 다른 사람이 코드를 쉽게 관리할 수 있는지 금방 알 수 있다. 프로그램이 잘 문서화되어 있으면 유지보수가 쉽다.

주석에서 불필요한 단어는 생략하여야 한다. 살아 있는 사람에게 하는 말같이 주석을 쓰는 것이 좋다. 다음과 같은 메서드에 주석을 달 때 적용해보자. 예를 들어, "This function searches list x for a value y and ..." 또는 "Function isIn searches list x for a value y ..." 라는 주석은 명세서 같다. 이런 주석은 너무 말이 많으며 주석이 아니라 설명이다. 대신 다음과 같이 주석을 단다.

```
/* Return (the value of the sentence) "y is in list x" */
boolean isIn(int y, List x)
```

■ 과도한 주석

어떤 사람들은 거의 모든 줄에 "//" 주석을 넣는 경향이 있다. 이런 주석은 소음과 같아 프로그램을 읽기가 어렵다. 다음과 같은 주석도 피해야 한다.

```
i= i+1; // add one to i
```

읽는 사람은 Java에 대한 경험이 있고 기본 문장을 이해한다고 가정한다.

■ 클래스 불변조건

클래스 불변조건(invariant)은 클래스의 속성에 대한 의미와 제약의 모음이다. 클래스 불변조건은 일반적으로 각 개별 필드 선언에 대한 주석에 배치되거나 필드 앞에 단일 주석으로 배치한다.

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

위와 같이 주석에 /**를 달면 IDE에서 javadoc가 자동 문서화 하여 변수가 사용될 때 선언문을 팝업으로 보여준다. 자동 문서화가 필요 없다면 선언과 같은 줄에 한 줄짜리 주석으로 원시코드를 읽는데 방해가 되지 않는 스타일로 쓴다.

```
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
```

생성자의 목적은 객체가 불변조건을 만족하도록 모든 필드를 초기화하는 것이다. 그 후에는 메서드를 작성할 때나 메서드가 호출될 때 클래스 불변 값이 true라고 가정하고 객체의 불변조건 값이 true로 종료되도록 메서드 본문을 작성한다. 클래스 불변조건을 자주 보면 필드의 의미나 제약 조건을 잊어버리는 실수를 예방할 수 있다.

■ 메서드 주석

모든 메서드 앞에는 빈 줄을 넣고 그 다음에 메서드가 하는 일을 설명하는 Javadoc 스펙을 선행 조건과 함께 쓰는 것이 좋다. 메소드를 호출할 때 매개변수에 대한 제약 조건이다.

Java에는 매개 변수 및 메서드 결과값을 설명하기 위한 특정한 규칙이 있다. 예를 들어, 매개 변수 b와 결과값을 설명하기 위해 Javadoc 스펙 내에 다음과 같이 작성한다.

```
@param b one of the sides of the triangle
@return The area of the triangle
```

스펙 기능을 사용하지 않고도 보다 간결하고 명확하게 주석을 작성할 수 있다. 메서드 주석을 다는 규칙은 메서드가 무언가를 수행한다는 문장으로 작성하여 관련된 모든 매개 변수를 언급하는 방법이다.

```
/** Print the sum of a and b. */
public static void printSum(int a, int b) { ... }
```

함수에 대한 주석은 일반적으로 반환 대상을 표시하기 위해 작성한다.

```
/** Return area of triangle whose side lengths are a, b, and c. */
public static double area(double a, double b, double c) { ... }
    or
/** = area of triangle whose side lengths are a, b, and c. */
public static double area(double a, double b, double c) { ... }
```

■ 클래스 주석

각 public 클래스는 별도의 파일에 저장한다. 따라서 파일의 시작 부분에는 클래스의 용도를 설명하는 주석이 포함되어야 한다. 간단하고 짧은 요약일 수 있으나 저자, 마지막 수정 날짜 등에 관한 정보를 여기에 써넣는다.

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

■ 문장 주석

산문에서 문장이 단락으로 그룹화되어 있는 것처럼, 메서드 본문의 문장 순서는 논리적 단위로 그룹화되어야 한다. 문장 주석은 논리 구조 앞에서 의미를 설명하는 주석으로 선행하여야 그 명확성이 향상된다. 문장 주석은 논리 단위에 대한 일종의 명세로 사용된다. 따라서 논리 구조의 기능을 정확하게 기술하여야 한다.

논리 단위에 대한 주석을 문장 주석이라고 한다. 블록이 무엇을 하기 위한 것인지 주석으로 작성되어야 한다.

```
// 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

객체지향 패러다임은 분석에서 설계, 설계에서 프로그래밍으로의 연결이 매우 자연스럽다. 따라서 설계에서 구현으로 전환하는 것도 상대적으로 쉽다. 설계나 IDE 도구는 UML 다이어그램으로부터 원시코드 골격을 자동 생성한다. 하지만 IDE 도구에 따라 생성하는 원시코드가 다르다. IDE 도구를 사용하지 않는다면 설계를 보고 직접 코딩하여야 하는데 UML 다이어그램에서 코드 골격을 생성하기 위한 규칙을 알아본다.

9.3.1 연관의 코딩

주어진 클래스 다이어그램에서 클래스, 속성, 오퍼레이션, 상속의 구현은 쉽다. 하지만 연관은 그렇게 쉽지 않다. 집합 관계를 살펴보자. 클래스 B는 클래스 A의 집합이라고 하자. A의 인스턴스가 B의 인스턴스의 일부라는 사실을 나타내기 위하여 B에서 A를 지칭하는 참조가 필요하다는 것이 핵심이다. 연관 관계를 구현하는 여러 가지 방법이 있다. 포인터를 사용하는 방법이 있는데 다음과 같다.

- 1대 1 연관 클래스 A와 클래스 B 사이에 1대 1 연관 관계가 있다면 A에서 B의 함수를 호출할 수 있고 A가 B에 대한 참조를 갖도록 구현한다. 반대로 B에서 A의 함수를 호출할 필요가 있다면 B가 A에 대한 참조를 갖도록 구현한다.
- 1대 N 클래스 A와 클래스 B 사이에 1대 N의 연관 관계가 있고 클래스 A에서 인스턴스 B의 메서드를 호출할 것이 있다면 [그림 9.2]와 같이 클래스 A(Schedule)가 클래스 B(CourseOffering)의 참조를 모음으로 가지고 있도록 구현한다. 반대로 B의 인스턴스가 A의 메서드를 호출할 일이 있다면 B가 A에 대한 참조를 가지도록 구현한다.
- N대 N N대 N의 관계는 중간에 연관 클래스를 도입하여 1대 N의 관계로 바꾸어 설계하기도 한다. N대 N을 직접 구현하려면 B 객체에 대한 참조 모임을 A가 갖게 하고 반대로 A 객체에 대한 참조 모임을 B 객체가 갖도록 한다.

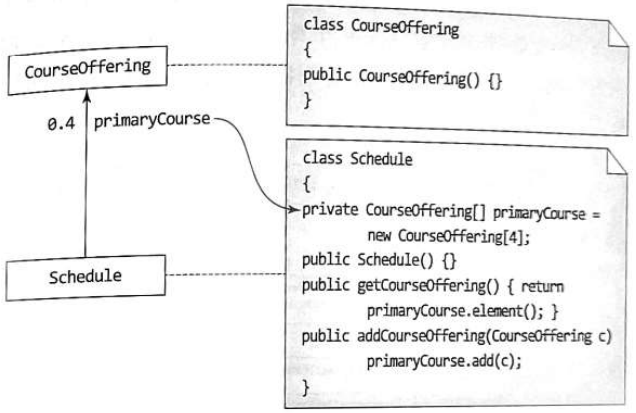

그림 9.2 1대 N 연관의 구현

위에서 A에서 B를 참조하는 변수와 B에서 A를 참조하는 변수를 모두 사용하면 A가 B를 호출하고 또한 B가 A를 호출하는 경우 성능을 향상시킨다. 하지만 연관을 추가하고 삭제하는 것은 참조를 변경시킬 필요가 있다. 연관 관계를 구현하는 다른 방법은 연관을 나타내는 새로운 클래스를 도입하는 것이다. 예를 들어 클래스 A와 클래스 B사이의 연관을 AssocAB라는 클래스를 정의하여 구현하면 다음과 같이 된다. 양쪽 클래스에 상대 인스턴스의 모임을 갖는 참조를 두면 된다.

```
public class AssocAB {
  private static Collection instance;
  private A a;
  private B b;
```