공학
트렌드

소프트웨어공학센터 경영지원TF팀

클린코드 Part 1 : 이해와 규칙
– 사람이 쉽게 읽을 수 있는 코드 만들기 –

2014. 12. 30. [제118호]

Contents

정보통신산업진흥원 부설
nipa 소프트웨어공학센터

Ⅰ. 클린코드란 무엇일까?

클린코드란 무엇을 말하는 것일까? 이에 대한 올바른 이해를 위하여 소프트웨어 대가들이 생각하는 클린코드는 무엇인지를 먼저 살펴보기로 하자.

- 나는 우아하고 효율적인 코드를 좋아한다. 논리가 간단해야 버그가 숨어들지 못한다. 의존성을 최대한 줄여야 유지보수가 쉬워진다. 오류는 명백한 전략에 의거해 철저히 처리한다. 성능을 최적으로 유지해야 사람들이 원칙 없는 최적화로 코드를 망치려는 유혹에 빠지지 않는다. 클린코드는 한 가지를 제대로 한다. – 비야네 스트롭스트룹 (C++창시자)
- 컴퓨터가 이해하는 코드는 어느 바보나 다 짤 수 있다. 훌륭한 프로그래머는 사람이 이해할 수 있는 코드를 짠다. – 마틴파울러 (리팩토링 저자)
- 클린코드는 단순하고 직접적이다. 클린코드는 잘 쓴 문장처럼 읽힌다. 클린코드는 결코 설계자의 의도를 숨기지 않는다. 명쾌한 추상화와 단순한 제어문으로 가득하다. – 그레디 부치 (객체지향 대가)
- 클린코드는 작성자가 아닌 사람도 읽기 쉽고 고치기 쉽다. 단위 테스트 케이스와 수용 테스트 케이스가 존재한다. 이런 이름은 의미가 있다. 특정 목적을 달성하는 방법은 (여러 가지가 아니라)하나만 제공한다. 의존성은 최소이며 각 의존성을 명확히 정의한다. API는 명확하며 최소로 줄였다. – 데이브 토마스 (실용주의 프로그래머)
- 클린코드의 특징은 많지만 그 중에서도 모두를 아우르는 특징이 하나 있다. 클린코드는 언제나 누군가 주의 깊게 짰다는 느낌을 준다. 고치려고 살펴봐도 딱히 손 댈 곳이 없다. 작성자가 이미 모든 사항을 고려했으므로. 고칠 궁리를 하다보면 언제나 제자리로 돌아온다. 그리고는 누군가 남겨준 코드, 누군가 주의 깊게 짜놓은 작품에 감사를 느낀다. 마이클 페더즈 ("Working Effective with Legacy Code" 저자)
- 코드를 읽으면서 짐작했던 기능을 각 루틴이 그대로 수행한다면 클린코드라 불러도 되겠다. – 워드 커닝엄 (위키창시자, 익스트림 프로그래밍 창시자)

소프트웨어 대가들이 정의한 클린코드를 살펴보면, 공통적으로 지적하는 것이 있다. 코드를 작성한 의도와 목적이 명확하며 다른 사람이 쉽게 읽을 수 있어야 한다는 점이

다. 코드의 가독성이 좋다는 것은 다른 사람이 코드를 이해하는데 들이는 시간을 최소화하는 방식으로 작성된다는 것을 의미한다.

II. 클린코드는 왜 필요할까?

개발자에게 클린코드가 왜 중요한 것일까? 개발자가 새로운 기능을 추가하거나 수정하기 위해 코드를 읽고 쓰는 비율을 따져본다면 읽기 : 쓰기 = 10 : 1에 달한다. 이처럼 개발과정에서 읽기 비율이 높기 때문에 읽기 쉬운 코드는 개발자에게 매우 중요하다. 아울러 기존 코드를 읽고 이해해야 새로운 코드를 짤 수 있기 때문에 읽기 쉽게 만든다는 것은 짜기도 쉬어진다고 할 수 있다. 따라서 개발시간을 단축할 수 있는 좋은 방법이 처음부터 읽기 쉬운 코드를 짜는 것이다.

표 1_소프트웨어 노후화 증상과 지표들

증상	설명	지표
오염 (pollution)	비즈니스 기능을 수행하지 못하는 많은 컴포넌트들이 존재한다.	사용하지 않는 데이터, 중복된 코드
문서부족 (no documents)	현재 코드와 문서가 일치하지 않으며 수정과 변경을 위한 도메인 지식은 크게 증가하는데 개발자는 따라가지 못한다.	설계 문서, API 문서, 담당자 변경 횟수
의미없는 이름 (poor lexicons)	함수, 클래스, 컴포넌트 이름들이 명확한 의미를 갖기 못하거나 실제 작동과 불일치하는 경우가 많다.	네이밍 컨벤션 준수 코드 리뷰
높은 결합도 (tightly coupling)	클래스와 컴포넌트 간에 데이터와 컨트롤 흐름이 네트워크로 복잡하게 연결되어 있다.	콜그래프(call graph), 팬인(Fan-in), 팬아웃(Fan-out)
아키텍처 침식 (architecture erosion)	아키텍처가 더 이상 구별되지 않는 여러 다른 솔루션으로 이루어져있고, 유지보수에서 긴급하게 손을 댄 조치들로 아키텍처 상 변형들로 인해 시스템의 품질이 떨어져 있다.	의존도 시스템 성능 시스템 장애빈도

새로운 신규 프로젝트라고 하더라도 이후 유지보수는 피할 수 없게 된다. 처음 시작할 때는 설계나 코드 품질에 대하여 노력을 많이 기울이지만 완료 시점이 다가올수록 시간이 부족하여 테스트나 리팩토링에 소홀해지기 쉽다. 제품 및 서비스가 론치되고 새로운 요구사항이 들어오고 변경과 결함수정을 반복된다. 점점 코드의 크기와 복잡도는 증가할

수록 기존 기능에 버그를 유발할까 두려워 리팩토링도 어려워지게 마련이다. 이러다보면 소위 냄새나는 코드 (code smell)가 만들어지고 소프트웨어는 노후화될 수밖에 없다.

Ⅲ. 클린코드를 만드는 규칙들

개발자들이 이런 냄새나는 코드를 일부러 만드는 것은 아니지만 코드는 동작하더라도 잘못된 클래스의 책임 설정이나 부적합한 이름, 모호한 경계설정 등의 설계 상의 약점들이 쌓이다보면 냄새나는 코드를 피하기 어려워진다. 그럼 개발자가 개발과정에서 어떠한 점들을 유의해야 하는지 몇 가지 규칙들을 살펴보자.

3.1 의미있는 이름 (Naming)

그림 1_의도가 명확한 코드

의도가 모호한 코드	의도가 분명한 코드

`int d; // 경과 시간 (단위: 날짜 수)`	`int elapsedTimeInDays;`
	`int daysSinceCreation;`
	`int daysSinceModification;`
	`int fileAgeInDays;`

`// 각 이름이 충분한 정보 제공을 하지 않음`
`public List<int[]> getThem() {`
`  List<int []> list1 = new ArrayList<int []>();`
`  for(int [] x: theList) {`
`  if(x[0] == 4)`
`  list1.add(x);`
`  }`
`  return list1;`
`}`
	`// 이름을 명확히 변경`
	`public List<int[]> getFlaggedCells() {`
	`  List<int []> flaggedCells = new ArrayList<int []>();`
	`  for(int [] cell: gameBoard) {`
	`  if(cell[STATUS_VALUE] == FLAGGED)`
	`  flaggedCells.add(cell);`
	`  }`
	`  return flaggedCells;`
	`}`

	`// int []을 Cell Class로 변환`
	`public List<Cell> getFlaggedCells() {`
	`  List<Cell> flaggedCells = new ArrayList<Cell>();`
	`  for(Cell cell: gameBoard) {`
	`  if(cell.isFlagged())`
	`  flaggedCells.add(cell);`
	`  }`
	`  return flaggedCells;`
	`}`

변수나 클래스, 메서드의 이름을 짓을 때 아래와 같은 질문을 통해 의도가 분명한 이름을 사용하는 것이 필요하다. 〈그림 1〉 예제를 통해 의도가 모호한 코드와 분명한 코드를 비교하였다.

- 왜 존재해야 하는가?

• 무슨 작업을 하는가?

• 어떻게 사용하는가?

클래스는 행위(멤버함수)의 주체로써 명사나 명사구로 표현하고, 함수 이름은 클래스가 행하는 행위로서 동사 또는 동사구를 사용한다. 아래 〈그림2〉의 클래스와 멤버함수를 참조하도록 한다.

그림 2_클래스와 멤버함수 네이밍

클래스	멤버함수
CIOCPServer	startService
CIOCPServerDlg	startStationServer
CIOCPServerDlg	makeLogPath
CIOCPServerDlg	writeLog
CMessageProcFunc	getSharedQuota
CMessageProcFunc	isAvailableToAddPhysicalDiskQuota
CMessageProcFunc	getUserPath
CMessageProcFunc	getAppendedUserPath

코드 상에 나만 아는 숫자를 쓴다든지 추상적인 변수명으로 무엇을 하는지 알 수 없는 이름도 사용하지 않도록 한다. 가령, 아래 조건문에서 사용된 숫자 25는 코드를 작성한 사람이 아니면 의미를 알 수 없다. 아울러 do( ), process( )와 같이 너무 일반적인 이름보다는 이름만으로도 언제 이 메서드를 호출해야 하는지 의미를 파악할 수 있도록 구체적으로 작성하도록 한다.

```
if (count > 25) {
    return;
}
```

아래와 같이 변수명을 i, j, k,로 쓰는 것도 사람들에게 혼동을 가져다주어 j 대신 i를 깜박하고 쓰게 할 수도 있다. 이러한 경우 club_i, member_j, user_k와 같이 변수명이 길어지기는 하지만 구분할 수 있는 변수명을 쓰도록 한다. 이는 통해 적절한 위치에서 사용될 수 있게 해주면 버그를 유발하는 오류를 제거할 수 있다.

```
for (int i = 0; i < clubs.size(); i++)
    for (int j = 0; j < clubs[i].members.size(); j++)
        for (int k = 0; k < users.size(); k++)
            if (clubs[i].members[k] == users[j])
                cout << "user[" << j << "] is in club[" << i << "]" << endl;
```

문맥에 맞는 단어를 사용하고 일관성 있는 단어를 사용하도록 한다. 한 단어를 여러 목적으로 사용하는 것도 피하도록 한다. 아래와 같이 select로 시작하는 메서드 중간에 갑자기 find로 시작하는 메서드가 나온다던지, get 이외에 fetch를 쓰는 경우 처음보는 개발자는 다른 메서드라고 생각하고 동일한 기능을 하는 코드가 있음에도 새로운 코드를 추가할 수 있다.

selectBook(...),

selectMusic(....),

findVideo(...)

```
public class Vehicle{

        public double getFuelRemaining();
        public double retrieveDegreeOfTireAbrasion();
        public double fetchCurrentSpeed();

}
```

3.2 명확하고 간결하게 주석달기 (Comment)

이해를 돕기 위해 주석을 코드에 적게 되는데, 왜 주석을 달아야 하는지 생각해봐야 한다. 간혹 필요 없는 주석이나 이상한 주석은 코드를 읽는 것을 더 어렵게 하는 경우도 있기 때문이다. 따라서 주석이 필요한 경우 최대한 간결하고 명확하게 작성하여 읽는데 많은 시간을 사용하지 않도록 주의한다. 코드 네이밍을 구체적이고 명확하게 하는 것이 주석이 필요하지 않도록 만드는 좋은 방법이다. 아울러 코드 안에 변경이력이나 저자 등의 기록은 형상관리도구(SCM)을 사용하여 해결하도록 한다.

가령 아래 코드를 보면 주석을 읽지 않으면 레지스터리를 지우는 것으로 오해할 수 있다. 실제 레지스트리는 지우지 않고 키에 대한 핸들만 풀어주는 것이기 때문이다.

```
// Releases the handle for this key. This doesn't modify the actual registry.
void DeleteRegistry(RegistryKey* key);
```

이런 경우는 주석을 달기 보다는 아래처럼 코드를 수정하면 주석이 없더라도 원하는 바를 명확하게 전달할 수 있게 된다.

```
void ReleaseRegistryHandle(RegistryKey* key);
```

주석을 작성하는 기본은 바로 해당 코드를 처음 접하는 사람이 코드를 읽으면서 궁금

한 점이 어디일까를 생각해보는 것이다. 가령 아래 코드를 읽다보면 볼드 처리된 부분에 의문이 생길 수 있다. 왜 clear인데 swap을 할까? 이런 의문이 드는 곳이 바로 주석이 필요한 곳이다.

```
struct Recorder {
    vector<float> data;
    ...
    void Clear() {
        vector<float>().swap(data);
    }
};
```

따라서 바로 그 부분에 아래와 같이 주석을 해두면 코드를 읽는 사람이 바로 주석을 보면서, 의문점 없이 코드를 이해할 수 있게 된다.

```
// Force vector to relinquish its memory (look up "STL swap trick")
vector<float>().swap(data);
```

3.3 보기좋게 배치하고 꾸며라 (Aesthetics)

책을 읽을 때 같은 내용이라도 문단을 어떻게 표시하고, 폰트와 배열을 하는지에 따라 읽기가 한결 수월해지게 마련이다. 마찬가지로 코드에서도 읽는 사람이 편하게 읽을 수 있도록 구성하는 것이 중요하다. 아래 데이터베이스 기능을 테스트하기 위한 예제코드를 보면 코드가 지저분하고 읽기가 어렵다. 코드 중간에 반복되는 내용과 줄이 넘어가면서 한눈에 코드가 보이지 않는다. 코드에서 반복되는 ExpandFullName(database_connection, …) 구문은 새로운 함수로 선언하여 정리하고 배열을 정리해보자. 이는 리팩토링에서 사용되는 방법이기도 하다.

```
DatabaseConnection database_connection;
string error;
assert(ExpandFullName(database_connection, "Doug Adams", &error)
    == "Mr. Douglas Adams");
assert(error == "");
assert(ExpandFullName(database_connection, " Jake  Brown ", &error)
    == "Mr. Jacob Brown III");
assert(error == "");
assert(ExpandFullName(database_connection, "No Such Guy", &error) == "");
assert(error == "no match found");
assert(ExpandFullName(database_connection, "John", &error) == "");
assert(error == "more than one result");
```

헬프메서드(help method) 방법을 활용하여 CheckFullName 함수를 정의하고 이를 4개의 다른 변수를 통해서 테스트할 수 있다. 아래와 같은 방법을 통해 코드를 보기 좋게 만들 수 있을 뿐 만 아니라 구조적인 측면에서도 좋은 결과를 만들어준다.

```
CheckFullName("Doug Adams", "Mr. Douglas Adams", "");
CheckFullName(" Jake  Brown ", "Mr. Jake Brown III", "");
CheckFullName("No Such Guy", "", "no match found");
CheckFullName("John", "", "more than one result");


void CheckFullName(string partial_name,
                   string expected_full_name,
                   string expected_error) {
    // database_connection is now a class member
    string error;
    string full_name = ExpandFullName(database_connection, partial_name, &error);
    assert(error == expected_error);
    assert(full_name == expected_full_name);
}
```

한 가지 다른 사례를 살펴보자. 아래는 선언시점에 유사한 속성들로 분리해서 정리할 수 있는데, 하지 않았기 때문에 코드가 산만해 보인다.

```
class FrontendServer {
  public:
    FrontendServer();
    void ViewProfile(HttpRequest* request);
    void OpenDatabase(string location, string user);
    void SaveProfile(HttpRequest* request);
    string ExtractQueryParam(HttpRequest* request, string param);
    void ReplyOK(HttpRequest* request, string html);
    void FindFriends(HttpRequest* request);
    void ReplyNotFound(HttpRequest* request, string error);
    void CloseDatabase(string location);
    ~FrontendServer();
};
```

아래와 같이 의미가 유사한 것들을 묶어서 주석을 달아 구분되는 영역별로 설명을 해 둔다면 훨씬 보기 좋은 코드가 만들어질 수 있다. 보기 좋은 코드는 다른 개발자가 작성한 사람의 배려를 느낄 수 있게 해주며, 작업도 훨씬 용이하게 만들어준다.

```
class FrontendServer {
  public:
    FrontendServer();
    ~FrontendServer();

    // Handlers
    void ViewProfile(HttpRequest* request);
    void SaveProfile(HttpRequest* request);
    void FindFriends(HttpRequest* request);

    // Request/Reply Utilities
    string ExtractQueryParam(HttpRequest* request, string param);
    void ReplyOK(HttpRequest* request, string html);
    void ReplyNotFound(HttpRequest* request, string error);

    // Database Helpers
    void OpenDatabase(string location, string user);
    void CloseDatabase(string location);
};
```

Part 2에서는 클린코드를 만드는 규칙들 중에서 클래스와 객체 및 데이터 구조, 함수를 어떻게 다루고 코드 품질을 높이기 위한 오류 및 경계 값 처리, 테스트 방법에 대하여 사례를 소개하고자 한다. 아울러 클린코드 작성을 위해 조직에서 할 수 있는 프랙티스들도 간단하게 설명할 것이다.

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. The Art of Readable Code: Simple and Practical Techniques for Writing Better Code, Dustin Boswell, Trevor Foucher, O'Reilly, 2012
4. 리팩토링 : 코드 품질을 개선하는 객체지향 사고법, 마틴파울러(저)/김지원(역), 한빛미디어, 2012
5. 소프트웨어 프로세스 이야기, http://swprocess.egloos.com/