Chapter 02

프로그램 설계 시에 알아야 할 좋은 코딩 습관

- 최신 표준을 따르라
- 개발 인원을 적정한 규모로 한정하라
- 프로그램을 새로 만드는 것보다 유지보수하는 경우가 많다
- 프로그램을 쉽게 수정할 수 있다는 생각을 버려라
- 새로운 기법을 도입할 때는 신중히 하라
- 'Run and Fix' 전략을 피하라

최신 표준을 따르라

표준(standard)은 산업이 발전하고, 제도가 정비되고, 관련된 사람들의 지식이 증가하고, 환경이 변화함에 따라서 더 폭넓고 정밀해진다.

이런 현상은 C 언어에서도 마찬가지다. 커니건과 리치(Kernighan & Ritchie)가 C 언어에 대한 최초의 매뉴얼을 작성한 1978년 이후로 C 언어의 표준은 계속 정립되어 왔다. 앞에서 얘기한 것처럼, 처음에는 이 두 사람이 쓴 책인 『The C Programming Language』(Prentice Hall, 1978)가 일종의 표준처럼 받아들여졌다. 이 표준을 흔히 K&R1이라고 부른다.

그리고 이 두 사람이 지은 책의 두 번째 판[『The C Programming Language, 2nd Ed.』(Prentice Hall, 1988)]이 나왔을 때도 이 책이 C 언어에 대한 표준 기술서처럼 받아들여졌다. 이 책에서 제시한 C 언어의 표준을 보통 K&R2라고 부른다.

그 후에 국제표준기구에서는 이 두 사람의 아이디어를 기반으로 그 때까지 나온 여러 표준과 프로그래머들의 의견을 반영하여 C 언어에 대한 새로운 표준을 정하였는데, 그것이 ISO/IEC 9899: 1990이다.

여기서 잠깐

프로그래밍 언어에 대한 대표적인 국제 표준 기구로는 ANSI(미표준협회)와 ISO(국제표준기구)가 있다.

이 변화의 실례 중에 가장 대표적인 것이 main 함수의 기술 방식이다. main 함수를 처음에는 다음과 같이 기술하는 것이 일반화되고 표준처럼 여겨진 적이 있었다.

```
main() {
....
}
```

하지만, main 함수도 하나의 함수기 때문에 반환값의 자료형(return value type)을 표시해야 하며, 그 함수로 인수(argument)를 전달해야 한다고 생각하게 되었다. 그리고 반환 자료나 인수가 없는 경우에는 void라고 표시하기로 약속하였다.

이 덕분에 최근에는 main 함수를 기술할 때 다음과 같이 반환값의 자료형과 인수의 자료형을 반드시 기술하는 형태로 바뀌었다.

```
int main() {
...
  return 0;
}
```

위의 두 가지 사례 중에 어느 것이 더 명료한가? 당연히 두 번째 사례가 명료하다. 이 두 번째 사례는 최신 표준에 따라 작성된 것이다. 이처럼 최신 표준을 따르게 되면 프로그램을 작성하는 일에 명료함과 정확성을 가져오는 경우가 많다.

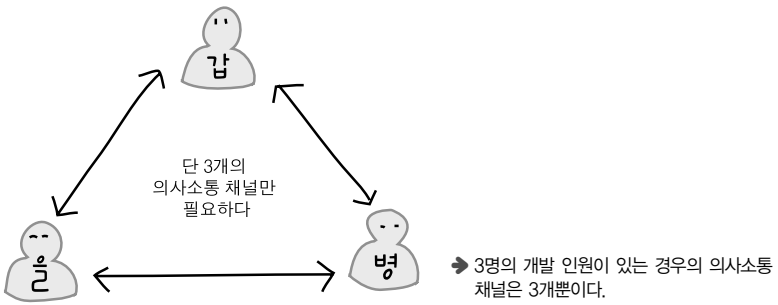
참고로, K&R 표준에서는 stdlib.h와 unistd.h 헤더를 제공하지 않았고, ANSI C와는 다른 함수 이름을 사용하였다. 두 표준의 함수 이름이 어떻게 바뀌었는지 알아보자.

>> K&R C와 ANSI C에서의 함수 이름의 차이

역할	ANSI C	K&R C
문자열에서 한 문자를 찾는다.	strchr	index
문자열의 끝부터 한 문자를 찾는다.	strrchr	rindex
메모리의 두 영역을 비교한다.	memcmp	bcmp
메모리를 0으로 설정한다.	memset	bzero
배열이나 구조체를 복사한다.	memcpy	bcopy

개발 인원을 적정한 규모로 한정하라

브룩스는 그의 저서 『The Mythical Man-Month: Essays on Software Engineering, 2nd Ed.』(Addison-Wesley, 1999)에서 개발자를 더 투입한다고 생산성이 증가하는 것은 아니라고 하였다. 실제로 필자의 경험으로 보아도 프로젝트마다 적정 인원이 정해져 있는 듯하다. 어떤 프로젝트에는 더 많은 인원을 투입해도 생산성이 크게 향상되지 않았다. 왜 그럴까? 필자는 이것을 의사소통의 채널 문제라고 본다. 다음 그림을 보자.



위의 그림을 보면 개발 인원이 3명인 경우에는 3개의 의사소통 채널을 형성한다. 즉, 갑과 을이, 을과 병이, 갑과 병이 각각의 채널을 통해 의사소통을 하는 것이다. 이렇게 소규모 개발 조직의 경우에는 서로 간에 의사소통의 횟수가 많아지게 된다. 그리고 소규모인 관계로 더 인간적이고 친밀하며 깊이 있는 의사소통을 할 수 있다. 또한, 공식적인 정보와 비공식적인 정보가 모두 교환된다.

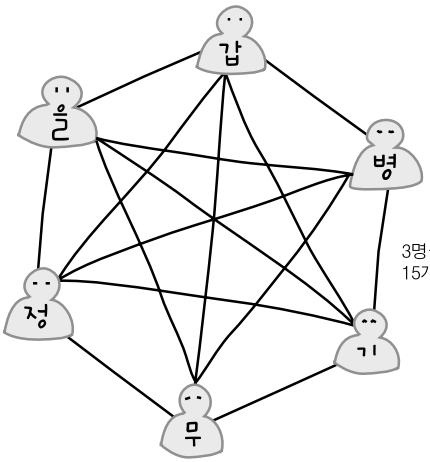
필자는 이렇게 의사소통이 이루어지는 정도를 '의사소통의 강도' 라고 표현하고자 한다. 어떤 사람은 이것을 '팀워크' 라고 표현하기도 한다. 팀워크든 의사소통의 강도든 상관없다. 의사소통이 얼마나 잘 이루어지는지가 중요한 것이다.

소규모 조직일수록 의사소통 채널의 개수는 적은 반면에, 의사소통의 강도는 높아진다고 추정해 볼 수 있다.

그렇다면 대규모 조직은 어떤가? 많은 인원이 개발에 참여하게 되는 경우에 의사소통은 주로 공식적인 방법을 택하게 된다. 그리고 규정된 문서, 규정된 방식, 규정된 채널을 통해서만 의사소통을 하도록 강요받으며, 비공식적인 의사소통은 무시되기 일쑤다. 또한 결재를 받는 데 며칠씩 걸려 의사소통의 속도가 느려질 수도 있다. 그러므로 의사소통 채널의 개수는 많아지지만 의사소통의 강도는 약해져 결국 전체적으로 의사소통의 효율이 작아지는 것이다.

그렇기 때문에 모든 개발 조직을 적정 인원수로 한정하는 것이 필요하다. 일반적으로 워드프로세서와 같은 대규모 프로젝트에서는 10명 내외의 핵심 개발자를 두고, 나머지 인원은 그들을 보조하는 정도의 역할을 담당하게 하는 것이 좋다. 게임 개발 같은 경우에는 4~5명의 개발자가 적정 수준으로 보인다. 이것은 공식적인

수치는 아니며 어디까지나 관찰에 의한 경험치에 불과하다. 각자의 프로젝트 성격에 맞는 적정 인원을 찾아내는 것은 각자의 몫이다.



3명을 보충하면 의사소통 채널은 15개로 늘어난다.

➔ 개발 인원이 늘어나면 의사소통 채널이 급격히 늘어난다.

프로그램을 새로 만드는 것보다 유지보수하는 경우가 많다

미국의 유명한 정보통신 전문 잡지인 「데이터메이션(Datamation)」에 따르면, 프로그램을 작성하는 데 드는 시간보다 유지보수하는 데 드는 시간이 많아지는 추세고, 그 추세는 매우 가파르다고 한다.

필자의 체험으로도 이 통계가 정확한 것 같다. 필자가 팀원들과 더불어 수백 개의 프로그램을 작성해 보았지만, 그 중에 새로 작성한 것은 30%도 되지 않는 것 같다. 나머지 70%는 기존의 프로그램을 갱신하거나 수정하는 형태로 작성한 것이다. 왜냐하면, 응용 프로그램에 있어서 업무 처리 방식은 정형화되어 있기 때문이다. 예를 들면, 회계 프로그램에 쓰이는 입력 담당 프로그램은 인사 프로그램의 입력 담당 프로그램을 거의 그대로 가져다 쓸 수 있다. 그리고 워드프로세서의 버퍼 처리 모듈은 네트워크 관리 프로그램의 버퍼 처리 모듈을 수정해 사용할 수 있다.

이런 면에서 보면 프로그래머가 하는 일의 일부는 창의적인 일이지만, 일부는 기능적인 일이다. 프로그래머의 작업 중에 기존의 프로그램을 수정하고 유지보수하는 기능적인 수준의 일이 많은 것이다.

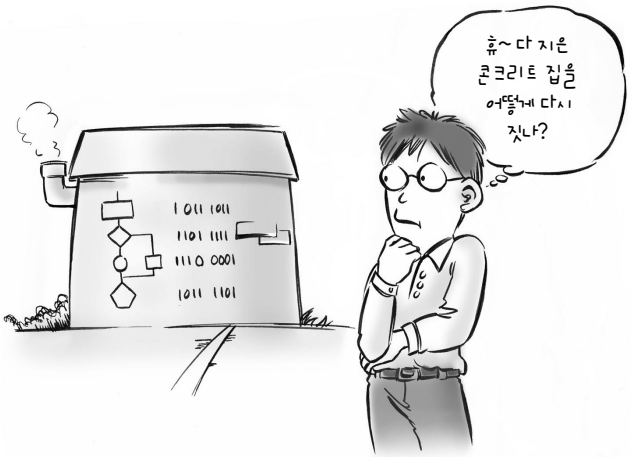
그런데 이런 기능적인 일을 하는 데 있어서 코딩 스타일을 적용하지 않고, 제멋대로 약삭빠르게 작성한 프로그램들은 방해가 된다. 주석이 없으면 프로그램을 이해하기 위해 많은 시간을 투자해야 하고, 이해하기 어려운 구문을 사용하면 일일이 컴파일하고 테스트하여 어떤 결과를 가져올 목적으로 그런 구문을 사용하려 했는지 검증해 봐야만 한다.

따라서 프로그램을 작성할 때는 반드시 코딩 스타일이 제시하는 지침들을 따라서 작성해야 한다. 코딩 스타일은 프로그램을 이해하기 쉽게 주석을 달라고 하고, 명료하고 단순하게 작성하라고 한다. 그리고 구체적인 지침을 제시한다. 프로그램을 소설이나 수필처럼 읽을 수 있을 정도로 쉽게 작성하라고 가르친다. 이런 규칙들을 따르면 프로그램을 수정 및 갱신하거나 유지보수하기가 쉬워진다.

프로그램을 쉽게 수정할 수 있다는 생각을 버려라

프로그램은 흔히 소프트웨어(software)라고도 불린다. 소프트웨어라는 말에는 '부드럽다', '고치기 쉽다', '손에 잡히는 물건이 아니다'와 같은 다양한 생각이 녹아 있다. 다른 의미는 접어 두고 소프트웨어가 진정으로 고치기 쉬운 것일까?

집을 짓는 경우와 소프트웨어를 개발하는 경우의 절차는 무척 비슷하다. 설계도를 그리고, 자재를 매입하고, 기술을 동원하여 건축하는 행위는 소프트웨어 개발의 그것과 동일하다. 집을 지을 때는 한 번 설계한 후에 짓기 시작하면 웬만하면 설계를 변경하지 않는다. 그런데 소프트웨어를 개발할 때는 설계도도 수시로 변경하고, 심지어 다 완성된 상태에서도 변경을 요구한다. 만약 집이라고 생각해 보라. 벽체를 뜯어내고 새 벽체를 세우는 것이 쉬운 일인가? 그런데, 왜 소프트웨어는 집의 벽체를 뜯어내는 것과 같은 작업을 쉽게 할 수 있다고 생각하는가?



➔ 소프트웨어를 고치는 것은 집을 수리하는 것만큼 어렵다.

소프트웨어라는 말 속에 숨어 있는 '고치기 쉽다'는 생각이 우리로 하여금 프로그램을 쉽게 고칠 수 있다는 편견을 가지게 하는 면이 있다. 그러나 결코 소프트웨어는 고치기 쉽지 않다.

프로그래머의 경험을 가진 사람이라면 누구나 인정할 것이다. 수백 줄이나 되는 프로그램의 논리를 바꾸라고 할 때, 화면 인터페이스를 바꾸라고 할 때, 네트워크 신호 처리 방식을 바꾸라고 할 때 차라리 기존 프로그램을 버리고 새로 작성하고 싶은 마음이 간절하다는 것을. 그리고 실제로도 기존 프로그램을 수정하는 것보다 아예 새로 작성하는 편이 더 빠른 경우도 있다.

여러 프로그램이 엮인 시스템 단위에서는 이런 문제가 더 심각하다. 작은 설계 변경이 시스템 전체에 영향을 미치기 일쑤기 때문에, 한 번의 설계 변경으로 수일에서 수십일, 때로는 몇 달의 공기가 늦추어질 수도 있다. 그 동안 투입될 인력과 비용을 감수한다고 해도 제때 마무리 되지 못한 프로젝트는 대부분 실패로 끝난다는 점에서 보면, 설계 변경은 곧 프로젝트 실패로 이어진다.

그럼에도 불구하고, 프로젝트 매니저부터 일선 프로그래머까지 고객들이 설계 변경 요구하면 쉽게 들어주곤 한다. 누구보다 프로그램 수정이 힘들다는 것을 아는 그들이 왜 건축가처럼 설계 변경을 단호하게 거절하지 못하는 것일까?

새로운 기법을 도입할 때는 신중히 하라

도끼를 아주 잘 다루어 나무를 잘 베던 사람이 한 번도 다뤄보지 못한 전동 톱으로 나무를 베야 한다면 어떤 문제가 생길까? 우선 전동 톱이 고장 나기 일쑤일 것이다. 톱날이 부러지거나, 기어가 망가질 가능성이 높다. 그리고 나무는 제대로 베이지 않을 것이고, 전동 톱을 다루던 사람이 다칠 가능성도 있다. 또한 생산성은 떨어지고, 문제는 더욱 커져만 갈 것이다. 물론, 시간이 지나면서 전동 톱을 다루는 방법을 익혀 간다면 언젠가는 전동 톱을 다루는 것이 훨씬 높은 생산성을 가져다 줄 것이다. 하지만, 지금 당장 나무를 베어야만 한다면 어떤 방법을 택할 것인가? 도끼인가, 아니면 전동 톱인가?

전동 톱을 서투르게 다루면 다치기 쉽다.

열 중 아홉은 전동 톱보다는 도끼를 선택할 것이다. 우리는 익숙한 것과 결별하는 것을 두려워해야 한다. 그렇다고 수구적인 태도를 가지라는 말은 아니다. 그러나 새로운 기술을 도입하는 데는 학습하는 시간이 필요하다는 것을 잊지 말아야 한다. 만약 프로젝트가 새 기술을 제대로 학습하기 전에 끝나야 한다면 차라리 기존 기술을 그대로 사용하는 편이 낫다.

4세대 언어들이 본격적으로 소개되던 시절에 많은 프로그래머들은 혼란에 빠졌었다. 어떤 프로그래머는 코볼이나 C 언어를 그대로 사용하였지만, 어떤 프로그래머들은 비주얼 베이직이나 4th Dimension과 같은 소위 4세대 언어들을 쉽게 도입하였다. 그 중에 일부는 성공하였고, 일부는 실패하였다.

자바가 도입되던 시절에, 자바를 도입해 임베디드 프로그램을 작성하려고 했던 사람들 중에 많은 사람들은 실패했지만, 웹 프로그램을 작성하려고 시도한 사람들은 대부분 성공하였다.

이런 현상은 오래 전에도 있었다. 구조화 프로그래밍 기법을 도입하던 시절에 그 기법을 서둘러 적용하려고 한 프로젝트는 실패했지만, 기존 프로젝트는 기존 방식 그대로 프로그래밍하면서 구조화 기법을 천천히 학습하게 하며 도입한 회사들은 대부분의 프로젝트에 성공하였다.

지금도 CBD나 CMM을 비롯하여 프로그램 작성과 관련된 다양한 기법이 소개되고, 다양한 도구(tool)나 언어들이 소개되고 있다. 새로운 기법들은 마치 우리에게 구세주인 것처럼 행세한다. 모든 프로젝트의 공정을 효율적으로 만들어주고, 납기일정을 제대로 맞출 수 있으며, 프로젝트 비용을 대폭적으로 줄여준다는 장밋빛 환상을 보여준다.

그러나 새 기법이나 새 도구가 외우기만 하면 문제를 풀게 만들어 주는 주문은 아니다. 전동 톱처럼 위험한 물건이다. 배우는 데 시간이 필요하고, 제대로 다루지 않으면 손해를 입힐 수 있는 것들이다.

새 기법이나 도구를 도입하기보다는 기존 도구를 개량하여 사용하는 것이 어떤 면에서는 더 생산적이다. 새 기법이 널리 알려지고 안정되기까지는 시간이 걸리지 않는 기술을 사용하는 편이 더 바람직하다.

이런 면에서 보면, 코딩 스타일은 혁신주의자도 보수주의자도 아닌 개량주의자라고 할 수 있다. 코딩 스타일은 우리에게 익숙한 언어를 그대로 사용하면서, 조금만 더 개량하라고 한다. 코딩 스타일을 익히고 적용하는 데 오랜 시간이 걸리지도 않고, 어렵지도 않다. 또한 안정되고 입증된 기술이다. 그러므로 필자는 새로운 프로그래밍 기법이나 언어 또는 CASE와 같은 도구를 도입하기 전에, 먼저 코딩 스타일을 도입하고 교육시키며 지키게 하기를 권한다.

'Run and Fix' 전략을 피하라

25년 전에 IBM은 프로젝트를 빨리 마치는 데 중점을 두고 프로젝트를 진행한 결과, 오히려 비용과 일정이 초과되기만 했다고 발표했다. 반대로, 소프트웨어의 결함을 제거하고 품질을 높이는 데 힘쓴 프로젝트의 경우에는 오히려 일정을 맞출 수 있었고, 생산성도 높았다고 발표하였다.

서두르면 일을 망친다는 것은 누구나 아는 사실이다. 그럼에도 불구하고 여전히 많은 프로그래머들은 일정 단축의 미신에 사로잡혀 있다. 프로그래머들의 세계에서는 오히려 더 서둘러야 일을 성사시킬 수 있다는 미신이 존재하고 있으며, 많은 프로그래머들은 프로그램을 작성하기 전에 충분히 생각하는 것을 귀찮아한다. 일단 프로그램부터 작성하여 결과부터 보려고 한다.

성급한 프로그래머

프로그램은 보이지 않는 추상적인 논리로만 만들어져 있기 때문에, 이 논리만으로 결과를 예측하는 것이 힘들다는 것은 사실이다. 그래서 충분히 훈련되어 논리만으로 최종 결과를 머리 속에 그려 볼 수 있는 프로그래머가 아니라면, 일단 어떤 형태의 결과든 출력물의 형태로 그것을 확인해보고자 하는 것이 프로그래머들의 본능이다. 그렇게 해서 결과를 확인하면 문제가 확실하므로 고치면 된다고 생각하곤 한다.

➔ 눈으로 확인하고 나서야 오류를 발견하는 프로그래머

이것을 필자는 '일단 실행하여 결과를 확인하고 고친다' 라는 행동 양식으로 규정하겠다. 영어로 'Run and Fix 전략(이하 RAF 전략)' 이라고 부를 수 있을 것이다. 이 전략은 소규모 프로그램이든, 대규모 프로젝트든지 상관없이 아주 광범위하게 퍼져 있다. 그러나 이 전략은 전투에서부터 전쟁까지를 모두 실패로 이끈다.

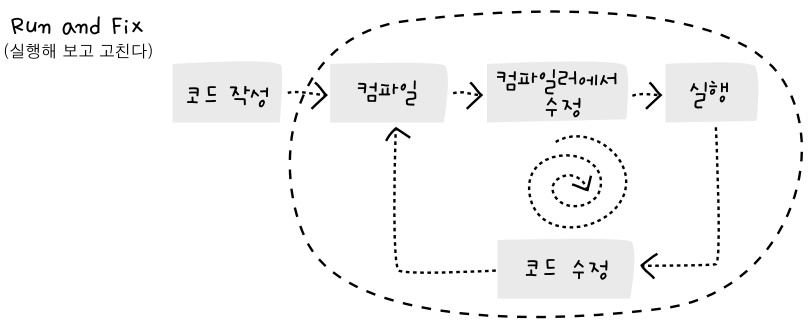
➔ Run and Fix 전략

우선 소규모 전투부터 살펴보자. 1,000줄 이내의 작은 단위 프로그램을 RAF 전략에 따라서 작성한다고 가정해 보자. 처음에는 대충 코딩한 다음에 일단 결과부터 확인하려고 보면 온갖 컴파일 오류를 만나게 될 것이다.

보통 이런 경우 컴파일 오류는 수백 개에서 수천 개에 이른다. 프로그래머는 컴파일 오류를 하나하나 수정하게 된다. 결국 컴파일 오류는 수백 개에서 수십 개 그리고 마침내는 10개 이내로 줄어들게 된다. 그 동안 프로그래머는 계속해서 코드를

수정하고 컴파일하고 오류를 확인해야 한다.

그런데 이런 'RAF 전략'을 쓰는 경우에 최종적으로 남은 몇 개 안 되는 컴파일 오류가 큰 문제가 되는 경우가 많다. 이런 오류들은 코드에서 쉽게 발견하기 어렵다. 결국 프로그래머는 코드를 작성하는 시간만큼 컴파일 오류를 잡아내는 데 시간을 투자해야 한다.



원 안에서 상당한 시간 동안에 걸쳐서 맴돌게 된다.

➔ RAF 전략은 어리석은 짓이다.

우여곡절 끝에 컴파일 오류를 다 잡아내어 'Compile Error:0, Warning Error:0'이라는 기쁜 메시지를 보게 되었다고 하자. 일단 이 상태에서 소스 코드는 실행 코드로 완전하게 바뀐다. 즉, 소스 코드가 실행 가능한 프로그램이 되는 것이다.

그러나 이제부터가 문제다. 첫 실행 결과가 의도했던 대로 나타나는 경우는 극히 드물다. 화면에 출력된 데이터의 줄 간격이 안 맞을 수도 있고, 입력 데이터 검증을 제대로 처리하지 못하는 경우도 있고, 아예 화면에 아무런 결과도 나타나지 않는 경우도 있다. 심지어 'Run Time Error'라는 기분 나쁜 메시지만 보여주고 프로그램이 종료되는 경우도 있다.

이렇게 실행 시에 문제가 있으면 프로그래머는 다시 그 문제를 해결해야만 한다. 결국 코드를 수정하고, 다시 컴파일하고, 수정하는 도중에 실수하면 컴파일 오류를 다시 잡아야 하고, 또 런타임 오류를 잡아내야 한다. 한 번에 문제가 해결되면 다행이다. 'RAF 전략'을 사용하는 프로그래머들을 살펴본 결과, 이런 경우에 대

부분 수회에서 수십 회까지 이런 과정을 반복하는 것을 볼 수 있다. 결국 'RAF 전략'은 소규모 단위 프로그램에서조차 생산성을 해치는 결과를 낳는다. 소규모 전투에서 패하게 된 것이다.

이제 대규모 전투 현장으로 가보자. 'RAF 전략'을 70% 정도의 팀이 사용한다는 믿을 만한 통계에 따르면, 거대한 시스템을 구성하는 단위 프로그램의 70% 정도는 'RAF' 전략에 따라서 만들어졌다고 추정해 볼 수 있다.

그런데 단위 프로그램의 실행 시에는 전혀 발생하지 않던 문제들이 단위 프로그램들을 시스템적으로 엮어서 실행하게 되면 나타나게 된다. 어떤 단위 프로그램에서는 결과값을 잘못 계산하기도 하고, 또 어떤 단위 프로그램에서는 부동 소수점 상의 계산 착오가 발생하기도 한다.

이런 '잠재된 문제'들은 통합 테스트 단계에서 많이 발견된다. 특히 'RAF 전략'을 사용하는 프로그램들이 이런 문제들을 발생시킨다. 결국 통합 테스트에서 심각한 문제가 발견될 것이며, 시스템 단위로 대폭적인 수정 작업에 들어가야 할 것이다.

그러나 소규모 전투 부대들, 즉 'RAF 전략'을 사용하는 팀들이 여전히 존재하기 때문에 결국 대규모 전투 부대인 SI 업체나 프로젝트 담당 부서도 실패할 수밖에 없게 된다. 모든 소속 소대의 70%가 전투에서 패배하는데, 사단이 전투에서 승리할 리 없는 것이다.

필자는 어떤 경우에라도 'RAF 전략'을 피하도록 권고한다. 만약 'RAF 전략'을 사용하는 프로그래머나 팀이 있다면, 아예 프로젝트 조직에서 제외시키는 것이 좋다. 만약 제외할 수 없다면 'RAF 전략'을 사용하지 못하도록 하는 것이 좋다.

그렇다면 어떻게 해야 할까? 해결 방법은 다음과 같다.

1. 프로그램의 구상 단계에 투자하는 시간을 늘려라.
2. 플로 차트나 의사 코드 등을 도입하여 프로그램의 논리를 충분히 구상해 보라.
3. 종이에서 작업하는 시간을 컴퓨터에서 작업하는 시간보다 더 많게 하라.
4. 사전에 프로그램의 결함을 예측하고, 그것들을 제거할 수 있는 방법을 찾아보라.
5. 충분히 심사숙고한 뒤에 코드를 작성하라.