How To Write Unmaintainable Code : Ensure a job for life ;-) This essay is a joke!

유지보수가 어렵게 코딩하는 방법

평생 개발자로 먹고 살 수 있다:-)

로에디 그린 지음, 우정은 옮김

한빛미디어 Hanbit Media, Inc.

HOW TO WRITE UNMAINTAINABLE CODE

유지보수가 어렵게 코딩하는 방법

평생 개발자로 먹고 살 수 있다 ;-)

로에디 그린 지음, 우정은 옮김

한빛미디어 Hanbit Media, Inc.

서문

자신의 무능함을 남의 탓으로 돌리지 말라.
– 나폴레옹(Napoleon)

자바 프로그래밍 분야에 종사자가 많아지기를 바라는 마음에서 아래 팁을 전수하려 한다. 이 팁은 유지보수가 어려운 코드를 작성하기로 유명한 스승으로부터 전수받은 것이다. 사람들이 스승이 남겨놓으신 코드에 간단한 수정을 추가하는데도 몇 년 이상이 걸리곤 했다. 진심을 다해 아래 규칙을 지켜 코딩한다면 본인 외에는 누구도 그 코드를 유지보수할 수 없게 된다. 즉, 평생 직장을 보장받게 될 것이다. 혹은 모든 규칙을 진심으로 따른다면 본인조차도 자신이 만든 코드를 유지보수 할 수 없는 날이 올 것이다!

상황이 이 정도까지 극단적이 되길 원하는 사람은 없을 것이다. 우리가 만든 코드가 겉보기에도 유지보수 가망이 전혀 없는 코드처럼 보이는 상황은 피해야 한다. 그렇지 않으면 우리가 만든 코드를 리팩터리하거나 최악의 경우 다시 작성해야 하는 위험에 처할 수 있다.

일반 규칙

Quidquid latine dictum sit, altum sonatur.
– 라틴어에는 뭔가 특별한 것이 있다.

유지보수를 담당하는 프로그래머를 좌절시키려면 그가 생각하는 방식을 이해해야 한다. 유지보수 담당 프로그래머는 우리가 만든 거대한 프로그램을 넘겨받았다. 그가 모든 코드를 읽기는 힘들기 때문에 해당 프로그램에 대한 이해도가 우리보다는 낮은 편이다. 아마도 그는 가능한 한 빨리 수정할 곳을 찾아내어 코드를 수정한 다음, 해당 수정으로 발생하는 부작용이 없는지 확인하고 작업을 마무리하려 할 것이다.

유지보수 프로그래머는 화장실에 걸린 두루마리 휴지 관을 통해 우리 코드를 살펴보는 것과 마찬가지 상황에 처해있다. 그가 휴지 관을 통해 우리 코드를 보면서 전체적인 그림을 그려낼 수 없게 하는 것이 핵심이다. 그가 찾고 있는 코드를 가능한 한 찾기 어렵게 만들어야 한다. 더욱 중요한 사항은 그가 안심하고 코드를 무시할 수 있도록 가능한 한 서툴게 코드를 작성해야 한다는 점이다.

프로그래머는 규약(convention)을 통해 안심하는 습성이 있다. 간혹 조금이라도 규약 위반을 발견하면 그는 돋보기를 들고 코드 전체 라인을 샅샅이 조사할 가능성이 크다.

언어의 모든 기능이 코드 유지보수를 어렵게 만든다고 생각할지 모르지만, 이는 사실이 아니다. 오직 언어의 기능을 적절하게 오용해야 유지보수가 어려운 코드를 만들 수 있다.

이름 짓기

험프티 덤프티(Humpty Dumpty)는 다소 경멸적인 어조로 말했다. "내가 단어를 사용할 때에는 더도 덜도 아닌 딱 그 의미를 전달하려는 것이다."
– Through the Looking Glass, 6 장 중에서(루이스 캐롤(Lewis Carroll) 지음)

변수와 메소드의 이름을 짓는 방법은 유지보수 할 수 없는 코드를 작성하는데 있어 상당히 중요한 기술이다. 이름은 컴파일러에 영향을 주지 않는다. 이름 짓기 기술로 유지보수 프로그래머의 정신을 혼미하게 만들 수 있다.

태아 작명법의 새로운 용도

태아 작명법 서적을 구입하자. 그러면 변수명을 뭐로 지어야 할지에 대한 고민을 덜 수 있을 것이다. Fred는 멋진 이름이며 입력하기도 쉽다. 입력이 쉬운 변수명을 원한다면 asdf를 사용해 보기 바란다.

단일 문자 변수명

변수명을 a, b, c 등으로 정한다면 간단한 텍스트 편집기로 해당 인스턴스를 검색하는데 애를 먹게 된다. 뿐만 아니라 그 변수가 무엇에 쓰이는 것인지 추측할 수 없게 방지하는 역할도 한다. 포트란(FØRTRAN)에서는 오랫동안 I, j, k를 인덱스 변수로 사용해왔다. 혹시라도 이러한 훌륭한 전통을 조금이라도 깨뜨리려는(예를 들어, ii, jj, kk 등으로 이름을 변경하려는) 사람이 있는가? 스페인 종교재판에서 이교도에게 어떠한 형벌을 가했는지를 그에게 경고하자.

창의적 오타

어쩔 수 없이 뭔가를 설명하는 변수명이나 함수명을 사용해야 하는 상황이라면 오타라는 무기를 선택하자. 몇몇 함수명과 변수명에 오타를 내고 다른 곳에서는 오타를 사용하지 않는다면(예를 들어, SetPintleOpening과 SetPintalClosing처럼) grep이나 IDE 검색 기술을 효과적으로 무력화할 수 있다. 이 방법은 생각보다 놀라운 효과를 발휘한다. 각기 다른 theatres/theaters(둘 다 극장을 의미)에 tory나 tori같이 국제적인 취향도 추가해본다.

추상화하라

가능한 한 it, everything, data, handle, stuff, do, routine, perform, 숫자 등과 같이 추상적인 단어를 변수명이나 함수명에 많이 사용하자(좋은 예, routineX48, PerformDataFunction, DoIt, HandleStuff, do_args_method).

머.리.글.자.

머리글자로 코드를 간결하게 만든다. 진짜 사나이는 머리글자를 풀어 설명하지 않고 있는 그대로를 선천적(유전적)으로 이해한다.

진화한 유의어 사전

유의어 사전에서 되도록이면 많은 단어가 같은 동작(예를 들어, 뭔가를 보여준다는 의미를 가진 display, show, present 등과 같은)을 가리키도록 하는 것도 지루함을 달랠 좋은 방법 중 하나다. 실제 의미상 차이가 없는 단어라 할지라도 알 듯 말 듯하게 모호한 힌트를 제공하는 것도 잊지 말자. 때로는 비슷한 두 함수가 완전 다른 동작을 하는 경우가 있다. 이러한 경우에는 두 함수를 같은 단어로 설명한다(예를 들면, "파일 기록", "종이에 잉크 칠하기", "화면에 보여주기"를 print라

는 한 단어로 설명할 수 있다). 어떤 경우에도 명확한 어휘를 사용해 특수 프로젝트에 사용할 용어집을 만들어 달라는 요구에 굴복하지 말아야 한다. 이러한 요구에 굴복하는 행위는 정보 은닉(information hiding)이라는 구조화된 디자인 원칙을 위반하는 프로답지 못한 행동이다.

다른 언어의 복수형을 사용하라

VMS 스크립트는 "Vaxen(Vax 컴퓨터의 복수형)"에서 반환하는 다양한 "statii(status의 복수형)"를 추적한다. 에스페란토(현재는 거의 쓰이지 않는 인공어), Klington(스타 트렉에 등장하는 전사 종족의 언어), Hobbitese(소설에 등장하는 가상 종족 호빗의 언어) 등을 사용하면 보다 효율적이다. Pluraloj와 같이 단어 뒤에 oj를 추가해서 에스페란토 어를 모방할 수 있다. 이러한 에스페란토를 사랑하는 작은 노력이 세계 평화에 기여하는 것임을 기억하자.

새로운 개념의 낙타표기법(CapiTaliSaTion)

무작위로 단어의 중간 음절 첫 글자를 대문자로 표기하자. 예를 들면 ComputeRasterHistoGram()처럼 메소드 명을 정할 수 있다.

이름을 재사용하라

언어의 규칙이 허용하는 범위 내에서 클래스, 생성자, 메소드, 멤버 변수, 파라미터, 지역 변수에 같은 이름을 사용하자. 이에 안주하지 말고 더 나아가서 {} 블록 내에서 이미 사용되고 있는 지역 변수명을 재사용할 수 있는지 고민하자. 이렇게 함으로써 유지보수를 담당하는 프로그래머가 모든 인스턴스의 범위를 유심히 살펴보도록 만들 수 있다. 특히 자바 언어에서는 일반 메소드를 생성자처럼 보이게 할 수 있다.

강세가 있는 단어

변수명에 강세가 표시된 단어를 사용하자. 아래 코드를 살펴보면,

```
typedef struct { int i; } int;
```

두 번째 int에서 i 에 강세가 있다. 간단한 텍스트 편집기로는 미묘한 차이를 구별하는 것이 거의 불가능하다.
컴파일러의 이름 길이 제한을 악용하라

컴파일러가 인식할 수 있는 변수명의 길이는 정해져 있다. 만약 변수명의 8글자만을 인식할 수 있는 컴파일러가 있다면, var_unit_update()과 var_unit_setup()처럼 마지막 부분만 살짝 바꿔보자. 그럼 컴파일러는 두 이름을 동일한 var_unit으로 인식할 것이다.

밑줄(underscore)은 진정한 친구다

_와 __를 식별자로 사용하자.

언어를 혼용하라

두 언어(사람의 언어나 컴퓨터의 언어)를 무작위로 배치하자. 만약 상사가 자신의 언어를 사용할 것을 강요한다면 어떻게 할 것인가? 상사에게 나만의 언어를 사용해야 생각을 더 잘 정리할 수 있다고 설명하자. 신사적인 설명으로 해결되지 않는다면? 언어 차별 행위에 대해 이의를 제기하고, 당장 고용주를 고소를 할 수도 있으며 거액의 배상금을 내야 하는 상황에 처할 수 있다고 협박하자.

확장 아스키(Extended ASCII)

ß, Ð, n 등과 같은 확장 아스키 문자도 변수명에 사용할 수 있다는 사실을 잊지말자. 간단한 편집기에서는 복사/붙여넣기 말고는 확장 아스키 문자를 입력할 수 있는 방법이 없다.

다른 언어의 이름을 활용하자

외국어 사전은 다양한 변수명을 제공하는 마르지 않는 샘과 같다. 예를 들어, point 대신 독일어 punkt를 사용할 수 있다. 비록 우리가 독일어를 잘 알진 못하지만, 유지보수 코더로 하여금 의미를 해독하면서 다양한 문화를 경험할 수 있게 해줄 수 있다.

수학에서도 이름을 구할 수 있다

아래와 같이 수학적 기호를 나타내는 단어를 변수명으로 사용해보자.

```
openParen = (slash + asterix) / equals;
```

정말 멋진 이름

의미상으로 전혀 관계없는 이름을 변수명으로 사용해보라.

```
marypoppins = (superman + starship) / god;
```

이 글을 읽는 사람은 자신도 모르게 단어의 뜻에 더 집중하게 되고, 실제 로직은 이해하기가 어려워진다.

이름을 변경하고 재사용하라

이름을 변경하고 재사용하는 기법은 Ada에서 특히 잘 먹힌다. Ada는 많은 표준 역컴파일 방지(obfuscation) 기법을 무력화시키는 언어다. 현재 사용하고 있는 모든 오브젝트와 패키지 이름을 처음 붙인 사람 대부분이 멍청이다. 언제까지 그들이 변하기를 기다릴 수 없으므로, 우리가 바뀌고 또 바꿔야 한다. Ada의 subtype과 renames를 이용해 우리만의 이름으로 개정하자. 방심하는 자들을 위한 함정으로, 예전 이름에 대한 래퍼런스 몇 개는 남겨두는 것이 효과적이다.

i가 필요할 때

다른 변수는 몰라도 절대로 i 를 가장 안쪽의 루프 변수로 사용하지 말라. 이외의 용도로는 i 를 자유롭게 사용할 수 있다. 특히, 정수가 아닌 변수에 사용할 때 더욱 효과적이다. 비슷한 방법으로 루프 인덱스로 n 을 사용할 수 있다.

규칙에 얽매이지 말지어다

썬 마이크로 시스템즈 스스로도 지키지 않는 선 자바 코딩 규칙(Sun Java Coding Conventions)은 가볍게 무시하자. 특수한 상황에서만 뜻이 미묘하게 달라지도록 이름을 정해보자. 어쩔 수 없이 낙타 표기 규칙을 따라야만 한다면 모호한 상황을 적극 활용해야 한다. 예를 들어, inputFilename와inputfileName을 혼용하는 것도 좋은 방법이다. 창의력을 발휘해서 이름을 복잡하게 지을 수 있는 자신만의 비법을 개발하자. 좋은 비법을 개발했는데도 따르지 않는 자가 있다면 바로 질책하자.

소문자 l과 숫자 1은 닮았다

Long 상수를 표현할 때 소문자 l을 사용해 보라. 예를 들어, 10l로 표기하면 10L이 아닌101로 착각하기 쉽다. uvw wW gq9 2z 5s il17|!j oO08 ’ ” ;,. m nn rn {[()]} 등의 문자를 명확하게 구분해주는 폰트를 멀리하자. 창의력을 발휘해보자.

전역으로 사용한 이름을 지역에 재사용하라

모듈 A에 전역 배열을 선언하고 같은 이름의 배열을 모듈 B의 헤더파일에 쥐도 새도 모르게 선언하자. 그러면 십중팔구 B에 선언한 배열을 전역 배열로 착각할 것이다. 물론 주석에 이와 같은 중복 선언이 있다는 사실을 알리는 행동은 삼가야 한다.

함수의 선언과 구현의 재활용

때로는 변수명을 정반대로 재활용해서 혼란을 야기하는 방법도 있다. 지역 변수 A, B와 지역 함수 foo, bar를 선언한다고 가정하자. 일반적으로 A는 foo의 매개변수로, B는 bar의 매개변수로 사용한다. 그러나 함수 선언을 실제 사용과 반대로 foo(B)와 bar(A)로 정의한다. 이렇게 해서 같은 변수명을 다른 용도로 사용하는 것처럼 만들 수 있다. 유지보수 프로그래머는 우리가 쳐놓은 혼란의 거미줄에서 빠져나가기 힘들어진다.

변수를 재사용하라

변수 존재 범위 규칙이 허용한다면 아무 관련 없는 기존의 변수명을 재사용해보라. 하나의 임시 변수를 전혀 관련이 없는 다양한 상황에 사용할 수 있다(변수를 재사용함으로써 스택 슬롯을 절약하는 것처럼 위장할 수 있다). 조금 더 사악한 방법을 원한다면 변수 자체의 의미를 변형하는 기법을 이용하라. 코드의 길이가 매우 긴 메소드의 가장 윗 부분에서 변수에 값을 할당한 다음 중간 어딘가에서 슬그머니 변수의 의미를 바꿀 수 있다(예를 들어, 0 기반 좌표를 1 기반 좌표로 바꾸는 등). 물론 이와 같은 변경을 문서화 해놓는 실수를 범하지 않아야 한다.

Cd wrttn wtht vwls s mch trsr

변수명이나 메소드명에 약어를 사용할 때에는 비록 이름이 길어질 수 있겠지만, 같은 단어에 다양한 변형을 더해 지루함을 없앨 수 있다. 이 기법은 문자열을 검색해서 우리 프로그램의 일부 기능을 이해해 보려는 게으름뱅이를 효과적으로 골탕먹일 수 있다. 철자를 어떻게 변형할 수 있는지 생각해 보라. 예를 들어, 국제적으로 사용하는 colour에 미국식 color 그리고 격식 없이 사용하는 kulerz 등 색을 가리킬 수 있는 단어를 다양하게 혼용할 수 있다. 이름을 생략하지 않으면 다양성을 상실한다. 결국 유지보수 프로그래머가 기억하기 쉬운 이름이 될 수 있다. 단어를 축약하는 방법은 다양하므로 한 단어를 같인 목적으로 사용하는 경우라도 다양하게 축약할 수 있다. 의도한 것은 아니지만 다양한 축약 표현을 사용하다보면 유지보수 프로그래머는 각각의 축약 단어가 서로 다른 변수라는 사실조차 눈치채지 못할 수 있다.

삼천포로 인도하는 이름

메소드의 이름이 의미하는 것보다 더 많은(혹은 더 적은) 동작을 수행하도록 프로그래밍하자. 간단한 예로 isValid(x)라는 메소드에 기능을 추가해 x값을 이진수로 변환하고 결과를 데이터베이스에 저장하도록 구현한다면 모두를 깜짝 놀랄 것이다.

m_

C++의 세계에서는 멤버 이름 앞에 “m_”을 붙이는 규약이 있다. 이는 메소드와 멤버를 구별하려고 만든 규약인데, 이 규약을 만든이는 메소드(method) 역시 “m”으로 시작한다는 사실을 잊은 듯 하다.

o_apple obj_apple

클래스의 인스턴스명을 “o”나 “obj”로 시작함으로써 우리가 크고 다형성을 갖춘 그림을 염두에 두고 있다는 사실을 보여주자.

헝가리 표기법

헝가리안 표기법은 소스 코드 판독을 어렵게 하는 핵폭탄급 기법 중 하나다. 헝가리안 표기법을 사용해보자! 소스코드는 방대하므로 헝가리안 표기법을 활용해 적절하게 코드를 오염시킨다면, 그 어떤 방법보다도 효율적으로 유지보수 엔지니어를 쓰러뜨릴 수 있다. 아래는 헝가리안 표기법의 본래 의도를 무력화하는 팁이다.

C++에서 "c"를 const에 사용하라(C++). "c"는 C++ 이외의 언어에서는 보통 변수가 상수임을 가리킨다.

다른 언어에서는 다른 의미로 해석되는 헝가리안 물혹(wart, 덧붙이는 음절이나 단어)을 찾고 사용하라. 예를 들어, C++ 코딩에서 모든 제어 형식에 "l_"과 "a_"와 같은 범위를 가리키는 접두어(파워빌더에 l은 지역을 a는 매개변수를 가리킨다)와 VB 스타일의 헝가리안 물혹을 사용하라. MFC 소스 코드에서 제어 형식에 헝가리안 물혹 표기법을 사용하지 않는 다는 사실을 마치 모르는 것처럼 행동하라.

공통적으로 자주 사용하는 변수는 되도록이면 추가 정보를 포함하지 말아야 한다는 헝가리안 원칙을 항상 위반하자. 위에서 설명한 기법들을 총 동원하고 각 클래스 형식은 커스텀 접두 물혹을 가지고 있다고 주장하므로 이를 달성할 수 있다. 물혹이 없는 것은 클래스임을 의미한다는 사실을 간파할 수 없게 해야 한다. 이는 정말 중요한 원칙이다. 이 원칙을 지키지 못하면 소스코드에 모음/자음 비율이 높아지면서 짧은 변수명이 범람하게 된다. 최악의 경우 소스코드 판독 방해 작전이 실패할 수 있고 자신도 모르는 새에 영어 표기법이 코드에 나타날 수 있다!

함수 파라미터와 여타 심볼은 이름을 통해 의미를 나타내야 한다는 헝가리안 개념을 노골적으로 위반하자. 그러나 헝가리안 형식 물혹 자체의 사용이 변수를 임시 변수로 보이게 할 수 있다.

의미상 전혀 연관이 없는 헝가리안 물혹 여러 개를 덧붙여 사용해보자. 실생활에서 활용된 예를 들면 "a_crszkvc30Last-NameCol"같은 변수를 만들 수 있다. 유지보수 엔지니어 팀 전체가 이 변수명을 "이 변수는 const이고 레퍼런스 형식으로 함수 매개변수로 사용되는데 테이블의 기본 키 가운데 하나인 'LastName'이라는 이름의 Varchar[30] 형식의 데이터베이스 열에서 가져온 데이터를 담고 있다."라고 해독하는데 3일이 걸렸다. "모든 변수는 public이어야 한다"라는 규칙을 이 기법에 접목하면 수천 라인의 코드를 대체할 수 있는 막강한 파워를 발휘할 수 있다!

사람의 뇌는 동시에 오직 7 개의 정보를 유지할 수 있다는 원칙을 마음껏 활용하자. 위 규칙을 따른다는 것은 다음과 같은 코드를 작성한다는 것을 의미한다.

1. 하나의 할당문에 14개의 형식과 이름 정보를 사용한다.
2. 하나의 함수가 3개의 매개변수를 전달하고 29 개의 형식과 이름 정보를 가진 결과값을 할당한다.
3. 적당히 복잡한 중첩 구조를 이용하면 단기 기억의 한계를 가볍게 초과할 수 있다. 특히 유지보수 프로그래머가 블록의 시작과 끝을 한눈에 확인할 수 없는 경우에 효과가 커진다.
4. 유지보수 프로그래머가 각 블록을 한 화면에 확인하기 어렵게 만들 수 있다면 중첩 구조체로도 단기 기억 메모리 한계를 간단히 초과할 수 있다.

헝가리안 표기법의 변형

헝가리안 표기법을 활용한 또 다른 술책으로 "변수명은 그대로 사용하되 변수 형식을 바꾸는" 방법이 있다. 이 방법은 윈도 응용 프로그램이 Win16 WndProc(HWND hW, WORD wMsg, WORD wParam, LONG lParam)에서 to Win32 WndProc(HWND hW, UINT wMsg, WPARAM wParam, LPARAM lParam)로 변경되는 경우와 같은 상황에서 어김없이 등장한다. 여기서 w는 words임을 가리키는 듯 하지만 실제로는 long을 가리킨다. Win64로 응용 프로그램을 변경하는 경우 이 사실이 더욱 명확해진다. Win64에서는 파라미터가 64비트이지만 기존의 "w"와 "l" 접두어는 변하지 않는다.

줄이고, 재사용하고, 재활용하라

콜백(callback)에 사용할 데이터를 저장할 구조체를 정의해야 한다면, 그 구조체를 PRIVDATA라고 부르자. 모든 모듈은 자신만의 PRIVDATA를 정의할 수 있다. 이 구조체로 VC++의 디버거를 교란시킬 수 있다. 변수 watch 윈도우에 PRIVDATA 변수가 있는 상태에서 해당 변수를 펼치려고 하면 디버거는 어느 PRIVDATA를 의미하는 것인지 결정할 수 없어 아무것이나 선택한다.

쉽게 찾지 못하게 숨겨라

16진수 값 $0204FB를 할당할 상수 변수명으로 blue 대신 LancelotsFavouriteColour와 같은 이름을 사용하라. 화면에는 완전한 파랑색이 나타나겠지만, 유지보수 프로그래머는 0204FB값을 판독(아마 그래픽 도구를 이용해서)해야 의미를 파악할 수 있을 것이다. 몬티 파이썬의 성배(Monty Python and the Holy Grail)라는 1975년 영국 영화를 좋아하는 광팬이라면 랜슬롯(Lancelot)이 좋아하는 색이 파랑색이라는 사실쯤은 금방 알아차릴 수도 있을 것이다. 몬티 파이썬의 성배 영화 전체 내용을 기억하지 못하는 유지보수 프로그래머가 있다면 프로그래머로써 자질이 없는 분이라고 생각할 수 밖에 없다.

위장술

표면으로 올라오는 시간이 오래 걸리는 버그일수록 찾기가 어렵다.
– 로에디 그린(Roedy Green)

위장술, 숨기기, 어떤 것을 마치 다른 것처럼 보이게 하기 등의 기술은 유지보수 할 수 없는 코드에 필수적인 기법이다. 이런 기술 중 대다수는 사람의 눈이나 텍스트 편집기로는 알아채기 힘들다는 약점을 이용한다.

주석으로 위장한 코드와 코드로 위장한 주석

실제로는 주석처리 되었지만 얼핏 보면 주석처리 되지 않은 것처럼 보이게 할 수 있다.

```
for(j=0; j<array_len; j+ =8)
    {
    total += array[j+0 ];
    total += array[j+1 ];
    total += array[j+2 ]; /* 속도 향상을 위해
    total += array[j+3]; * 루프의 코드를 길게
    total += array[j+4]; * 펼쳐 놓았다.
    total += array[j+5]; */
    total += array[j+6 ];
    total += array[j+7 ];
    }
```

위 코드에서 다른 색으로 표시하지 않았다면 세 줄의 코드가 주석처리 되었다는 사실을 알 수 있었을까?

네임스페이스

C는 Struct/union와 typedef struct/union의 네임스페이스를 구별한다(그러나 C++에서는 구별하지 않는다). 구조체든 유니언 네임스페이스든 같은 이름을 사용하자. 가능하다면 둘이 서로 호환되게 하자.

```
typedef struct {
char* pTr;
size_t lEn;
} snafu;

struct snafu {
unsigned cNt
char* pTr;
size_t lEn;
} A;
```

매크로 정의를 숨겨라

자질구레한 주석을 이용해 매크로 정의를 숨길 수 있다. 보통 프로그래머라면 지루한 주석을 끝까지 읽지 않으므로 절대 매크로를 찾을 수 없다. 매크로를 만들 때는 다음과 같이 특이한 동작을 써서 평범한 할당문처럼 보이게 만들어야 한다.

```
#define a=b a=0-b
```

매우 바쁜 것처럼 보여야 한다

다음과 같이 define 문을 이용해서 함수를 만들고 매개변수는 그냥 주석 처리한다.

```
#define fastcopy(x,y,z) /*xyz*/
...
fastcopy(array1, array2, size); /* does nothing */
```

define문을 여러 줄에 걸쳐 기술하면서 변수를 숨겨라

나쁜 예,

```
#define local_var xy_z
```

“xy_z” 를 두 줄로 분리한 좋은 예,

```
#define local_var xy\
_z // local_var OK
```

이렇게 하면 xy_z를 검색해도 나오지 않는다. C 전처리 프로그램은 줄의 끝 부분에 나오는 “₩” 를 다음 줄로 이어진다는 의미로 해석한다.

키워드를 위장한 이름

문서화 할 때에는 “file “과 같이 파일명을 표시해야 하고 Charlie.dat” 또는 “Frodo.txt” 처럼 파일명을 명백히 표시하지 말아야 한다. 되도록이면 가능한 한 예약어처럼 보이는 이름을 사용하는 것이 좋다. 예를 들어, “bank”, “blank”, “class”, “const”, “constant”, “input”, “key”, “keyword”, “kind”, “output”, “parameter” “parm”, “system”, “type”, “value”, “var” and “variable” 등을 매개변수나 변수명으로 사용해야 한다. 실제 예약어를 임의적으로 사용하면 명령어 프로세서나 컴파일러가 처리를 거부할 수 있다. 이를 잘 활용하면 사용자는 우리가 만든 임의의 이름과 예약어를 혼동하게 만들 수 있다. 누군가 딴지를 걸면, 사용자가 각 변수의 이해를 적절히 돕기 위해 사용한 것이라고 발뺌하면 그만이다.

코드에 사용한 이름은 화면 표시 이름과 달라야 한다.

화면에 표시되는 값과 변수명은 전혀 관련이 없도록 해야 한다. 예를 들어, 화면에는 “Postal Code” 로 표시되는 변수의 이름을 “zip” 과 같이 색다르게 정할 수 있다.

이름을 변경하지 마라

전체적으로 이름을 바꾸는 방법으로 두 섹션 코드를 동기화하는 것보다는 같은 심볼에 여러 TYPEDEF문을 사용하는 것이 바람직하다.

금지된 지역변수를 감추는 방법

전역 변수는 "악"과 같은 존재이므로 전역적으로 사용할 모든 데이터를 저장할 구조체를 정의하고 EverythingYoullEverNeed와 같이 똘똘한 이름을 붙여준다. 모든 함수가 이 구조체에 대한 포인터(포인터명은 handle이라고 함으로써 혼란을 더할 수 있다)를 갖게 할 수 있다. 실제로는 "handle"을 통해 전역변수를 마음껏 사용하면서 다른 이에게는 우리가 전역 변수를 사용하지 않는다는 인상을 줄 수 있다. 전역 변수를 사용하는 모든 코드에서 정적 변수를 선언하는 것도 좋은 방법이다.

동의어로 인스턴스 숨기기

유지보수 프로그래머가 뭔가를 수정하고 그로 인해 발생할 수 있는 부수효과를 확인할 때 일반적으로 프로그램 전체에서 사용된 변수명을 검색할 것이다. 동의어 사용이라는 간단한 방법으로 이러한 유지보수 프로그래머의 시도를 좌절시킬 수 있다.

```
#define xxx global_var // in file std.h
#define xy_z xxx // in file ..\other\substd.h
#define local_var xy_z // in file ..\codestd\inst.h
```

위 정의를 서로 다른 include 파일에 흩어놓아야 한다. 특히 include 파일이 서로 다른 디렉터리에 위치한 경우 효과적이다. 가능한 모든 범위에서 이름을 재사용하는 기법도 있다. 컴파일러는 정확하게 모든 이름을 구별할 수 있겠지만, 단세포적인 텍스트 검색기로는 이름을 구별하기 어려울 것이다. 불행하게도 SCID(Source Code in Database)가 점점 발전하면서 편집기가 컴파일러처럼 범위 규칙을 이해하게 되면 간단한 기법은 더 이상 사용할 수 없게 될 것이다.

길고 비슷한 변수명

변수명이나 클래스명은 되도록이면 길게 만들고 두 개 이상의 이름이 필요할 경우 한 글자만 바꿔놓거나 대소문자만 다르게 한다. 변수명 swimmer와 swimner는 좋은 예다. 대부분의 폰트로는 ilI1|나 oO08를 명확하게 구별하기 어렵다는 점을 악용하자. 예를 들어, parseInt와 parseInt 혹은 D0Calc와 DOCalc를 명확히 구분하기 어렵다. 이 중에서도 l은 얼핏 보기에 1과 구별하기 힘들기 때문에 변수명으로 사용하기 가장 좋은 알파벳 중 하나다. 뿐만 아니라 대부분의 폰트에서 rn은 m처럼 보이는 경우가 많다. 따라서 swimmer와 쉽게 구별하기 어려운 swirnrner도 좋은 변수명이다. HashTable과 Hashtable처럼 한 글자의 대소문자만 살짝 변경해서 변수명을 만드는 것도 좋은 방법이다.

비슷하게 발음되고, 비슷하게 보이는 변수명

xy_z라는 변수명 이외에 xy_Z, xy__z, _xy_z, _xyz, XY_Z, xY_z, Xy_z처럼 다양한 변수명을 사용하지 말라는 법은 없다.

때로는 변수명을 소리나는 대로 혹은 스펠링으로 기억하는 프로그래머를 많이 볼 수 있는데 대소문자나 밑줄로만 구별되는 변수명이 이들을 혼란에 빠뜨릴 것이다.

오버로드 그리고 당황

C++에서 #define을 사용해 라이브러리 함수를 오버로드하자. 얼핏 보면 친숙한 함수를 쓰고 있는 것처럼 보이겠지만 사실은 완전 다른 기능을 하게 할 수 있다.

효율적인 오버로드 연산자 선택하기

C++에서 +,-,*,/ 등과 같은 연산자를 사칙 연산의 의미와 전혀 관련 없는 동작을 하도록 오버라이드 하자. 스트로우스트룹(Stroustroup)도 쉬프트 연산자를 I/O에 사용했는데 우리도 그처럼 창의적이지 못할 이유가 없지 않은가? +를 오버로드할 때에는 i = i + 5;가 i += 5;와 같은 의미를 갖지 않도록 해야 한다. 최첨단 연산자 혼란 오버로딩 기법을 소개하겠다. 클

래스의 '!' 연산자를 오버로드 하면서 뭔가를 뒤집거나 부정하는 동작과는 아무 관련이 없는 동작을 하게 하는 것이 핵심이다. '!' 연산자가 정수를 반환하게 한다. '!'를 논리 연산자로 사용하려면 '!!'로 표기해야 한다. 그러나 '!' 연산 자체가 로직을 변경시키므로 결국 하나를 더 붙여서 '!!!'를 사용해야 한다. 여기서 말하는 ! 연산자는 불린 값 0이나 1을 반환하는 연산자로 비트단위의 논리 부정 연산자 ~와 혼동하지 말자..

new를 오버로드하라

"new" 연산자를 오버로드하라. 이는 +-/*를 오버로드하는 것보다 훨씬 위험하다. 기존 함수를 뭔가 다른 기능(그러나 오브젝트의 기능에 필수적인 함수이므로 변경하기 쉽지 않다)으로 오버로드한다면 큰 혼란을 야기할 수 있다. 사용자가 동적 인스턴스를 생성할 때에 온전한 인스턴스가 아닌 잘려나간 인스턴스 조각만 얻게 하는 것이 핵심이다. "New"라는 멤버 변수를 추가하므로 대소문자를 이용한 혼란 기법을 가미할 수 있다.

#define

C++의 소스코드 판독을 어렵게 하는데 #define의 활용도는 무궁무진해서 이에 대한 내용만 따로 집필할 수 있을 정도다. 소문자로 된 #define 변수로 원래 변수를 대체할 수 있다. 선처리 함수에는 절대 파라미터를 사용하지 말아야 한다. 전역 #define으로 원하는 모든 기능을 수행하자. 누군가는 #define을 활용해서 실제 컴파일이 이루어질 때까지 CPP를 다섯 번 통과하게 만들었다고 한다. 필자가 들어본 사례 중 가장 창의적인 활용방법이다. 똘똘하게 define과 ifdef를 사용해 각 헤더 파일에서 몇 번이나 해당 구문을 include했느냐에 따라 결과가 달라지게 할 수 있고, 이로써 코드는 혼란의 경지에 이르게 된다.

```
#ifndef DONE

#ifdef TWICE

// 세 번째 정의 내용
void g(char* str);
#define DONE

#else // TWICE
#ifdef ONCE

// 두 번째 정의 내용
void g(void* str);
#define TWICE

#else // ONCE

// 첫 번째 정의 내용
void g(std::string str);
#define ONCE

#endif // ONCE
#endif // TWICE
#endif // DONE
```

이제 얼마나 많이 헤더를 include했느냐에 따라 결과가 달라지므로 g() 함수에 char*를 전달해 호출하면 어떤 재미있는 일이 벌어지는지 구경하는 일만 남았다.

컴파일러 지시어

컴파일러 지시어는 같은 코드를 상황에 따라 다르게 동작하도록 만들어졌다. Boolean 쇼트 서킷 지시어와 long strings 지시어를 반복적으로 줄기차게 껐다 켜기를 반복하자.

문서화

진실을 말하는 것은 쉽다. 그러나 거짓말을 잘 하려면 센스가 필요하다.
– 사무엘 버틀러(Samuel Butler) (1835 – 1902)

종종 잘못된 문서화는 아예 문서화를 하지 않은 것보다 더 나쁜 결과를 초래한다.
– 베르트랑 메이어(Bertrand Meyer)

컴퓨터는 주석과 문서화 부분은 무시한다. 따라서 온 힘을 기울여 주석과 문서화를 활용한다면 불쌍한 유지보수 프로그래머를 좌절시킬 수 있을 것이다.

주석에 거짓말을 추가하라

적극적으로 거짓말을 할 필요는 없다. 그냥 자연스럽게 주석을 업데이트 하지 않아 내용이 맞지 않는 것처럼 보이게 하자.

명백한 사실을 문서화하라

코드에 /* add 1 to i */와 같은 양념을 추가한다. 중요한 점은 패키지나 메소드의 전체 목적과 같은 어려운 부분은 절대 문서화하지 않는다는 사실이다.

이유는 빼고 어떻게에 대해서만 문서화하라

프로그램이 무엇을 하는지에 대한 세부 사항 그리고 프로그램이 무엇을 달성하지 않는 것인지에 대해 문서화하라. 버그가 생기면 수정을 담당하는 프로그래머는 해당 코드가 무엇을 수행해야 하는지 알 수 없게 된다.

“명백하게” 문서화하지 말아라

예를 들어, 항공기 예약 시스템을 구현하고 있는데 다른 항공편을 추가하려면 25 군대를 수정해야 한다고 가정하자. 물론 어디를 수정해야 할지를 문서화하면 안 된다. 나중에 누군가 우리 코드를 수정하려면 전체 라인을 완벽하게 이해해야만 원하는 수정을 할 수 있을 것이다.

문서화 템플릿의 적절한 활용

함수 문서화 프로토타입을 이용해 자동으로 코드에 문서화 틀을 제공할 수 있다. 이 때 다른 함수(혹은 메소드나 클래스)에서 복사해서 사용하고 절대 필드에는 문서화 틀을 사용하지 말아야 한다. 어쩔 수 없이 필드에 문서화 프로토타입을 사용해야 한다면 모든 함수에서 같은 파라미터 이름을 사용하도록 하고 주의사항도 같게 하자. 물론 이 주의사항이 현재 함수와 전혀 관련이 없는 것일수록 좋겠다.

디자인 문서의 적절한 활용

상당히 복잡한 알고리즘을 구현해야 할 때에는 코딩을 하기 전에 디자인을 확실하게 해야 한다는 고전 소프트웨어 엔지니어링 원칙을 지켜야 한다. 상당히 복잡한 알고리즘의 각 단계에 대한 설명을 포함할 수 있도록 매우 세밀하게 디자인 문서를 작성한다. 이 문서를 세부적으로 만들수록 좋다.

디자인 문서에서 알고리즘을 구조화된 여러 단계로 나눠서 각 문단에 자동으로 계층적인 번호를 추가할 수 있다. 헤딩은 적어도 5단계로 만들자. 가능하면 구조를 잘게 나눔으로써 최종 결과물이 나왔을 때에 500개가 넘는 문단에 자동으로 번호를 추가할 정도가 돼야 한다. 다음은 실생활에 사용하는 어느 문단의 예다. 1.2.4.6.3.13 – 선택한 마이그레이션을 적용했을 때 발생하는 모든 효과를 표시하라 (간단한 의사코드는 생략).

그리고...(이제 본격적인 혼란의 세계로 빠져든다) 코드를 작성할 때에는 각 문단에 대응하는 전역 함수를 만든다

```
Act1_2_4_6_3_13()
```

디자인 문서에 나와있으므로 위 함수에 대한 문서화는 따로 필요치 않다!

디자인 문서의 번호는 자동으로 매겨지는 것이므로 변경이 발생했을 때 이를 일일이 코드에 반영하는 것은 정말 어렵다(함수명은 자동으로 변경되는 것이 아니므로). 그럼 큰일 아닌가? 우리는 문서를 최신으로 유지하지 않으면 되므로 걱정할 것 없다. 오히려 문서 추적에 필요한 사항을 모두 파괴하는 것이 좋다.

운이 좋다면, 우리의 뒤를 이어 작업할 사람은 지금은 멸종한 286 컴퓨터와 먼지가 수북이 쌓여있는 창고 선반 뒤에 숨겨진 앞뒤가 맞지 않는 초안 한 두 개를 건질 수 있을 것이다.

측정 단위

피트, 미터, 통과 같은 측정 단위를 변수, 입력, 출력, 매개변수에 문서화는 절대 하지 않는다. 이는 엔지니어링 작업에서 매우 중요한 요소다. 마찬가지로 변환 상수의 측정 단위나 값이 어떻게 전달되는지 등도 문서화하지 않는다. 주석에 잘못된 측정 단위를 슬쩍 넣는 것은 유치하지만 효과적인 방법이다. 좀더 사악한 방법을 원한다면 자기 자신만의 측정 단위를 만들어 보는 방법도 있다. 자신의 이름이나 다른 아무개 이름을 사용하고 해당 단위에 대해 정의하지 않는다. 누군가가 우리의 작업 방식에 이의를 제기한다면 소수점 연산보다 정수 연산을 잘 할 수 있도록 그렇게 한 것이라는 등의 동문서답을 이용하자.

문제점

코드의 문제점을 문서화하지 않는다. 클래스에 버그가 있을 수 있다는 사실을 발견했으면 혼자만의 비밀로 간직한다. 코드를 어떻게 재조직하거나 재작성해야 할지 아이디어가 떠올랐을지라도 문서로 남겨놓지 않는다. 영화 밤비(Bambi)에서 귀염둥이 썸퍼(Thumper)의 말을 기억하자. “좋은 말을 하지 않으려거든, 그 입 닫으라”. 코드를 만든 프로그래머가 우리의 주석을 보고 어떻게 생각하겠는가? 회사 사장이 본다면? 고객이 본다면? 심지어 해고될 수 있다. “수정해야 함!” 이라는 익명의 주석을 활용할 수 있다. 특히 이 주석이 어느 부분을 가리키는 것인지 명확하지 않을수록 좋다. 내용을 되도록 흐지부지하게 만들어서 누군가를 비난하고 있다는 느낌이 들지 않도록 주의해야 한다.

변수 문서화

변수 선언에는 절대로 주석을 달지 않는다. 변수를 어떻게 사용해야 하고, 경계 값은 무엇이며, 사용할 수 있는 값은 무엇이고, 함축된/표시된 십진수 점, 측정 단위, 표현 형식, 데이터 입력 규칙(전체 값을 채워야 하는지, 반드시 입력해야 하는 지와 같은), 값을 신뢰할 수 있는 조건 등과 같은 정보는 코드를 통해 충분히 얻을 수 있다. 주석을 기록하도록 상사가 강요한다면 메소드 바디에 변수를 넣어서 사용하자. 그러나 이 경우에도 절대 임시로라도 변수 선언에 주석을 추가하면 안된다.

폄하하는 말을 주석에 사용하기

외부 회사와 유지보수 계약을 체결할 수 없도록 다른 소프트웨어 선도 업체를 폄하하는 글을 추가한다. 특히 현재 회사와 계약할 수 있는 가능성이 있는 회사를 공격할수록 좋다. 예를 들면, 다음과 같다.

```
/* 내부 루프 최적화
Software Services Inc.,의 굼뱅이들은 아래와 같은 코드를 꿈에도 몰랐겠지.
그 녀석들은 아마 답답한 <math.h>의 기능을 이용해서 50배나 느리고 메모리도 많이 사용했을 꺼야.
*/
class clever_SSInc
    {
    .. .
    }
```

가능하다면 주석 뿐만 아니라 코드 구문상 중요한 부분에 모욕적인 발언을 추가하는 것이 좋다. 그러면 나중에 유지보수가 필요할 때 관리자는 해당 코드를 제거하려고 애쓸 것이다.

펀치 카드의 코볼(CØBØL)을 사용한 것처럼 주석을 추가하라

개발 환경 분야의 발전 특히, SCID 등의 사용을 거부하라. 모든 함수와 변수 선언이 한번의 클릭으로 가능할 것이라는 헛소문에 현혹되지 말자. 비주얼 스튜디어 6.0으로 개발한 코드는 edlin이나 vi를 사용하는 개발자가 유지보수 할 것이라 가정하자. 드라콘 식의 주석 규칙을 따르는 것이 소스 코드를 올바로 작성하는 것이라는 신념을 갖자.

몬티 파이썬 주석

makeSnafucated라는 메소드에는 /* make snafucated */과 같은 자바독(JavaDoc) 주석을 추가한다. 어디에도 snafucated의 의미를 정의하지 않는 것이 핵심이다. snafucated의 의미를 모르는 사람은 바보임에 틀림없다. Sun AWD 자바독에는 이와 같은 기법을 사용한 고전 예제가 많다.

프로그램 디자인

유지보수 할 수 없는 코드를 작성하는 기본 규칙은 가능한 한 여러 장소에 가능한 다양한 방법으로 사실을 기록하는 것이다. – 로에디 그린

유지보수가 쉬운 코드를 작성의 핵심 요소는 응용 프로그램의 각 요소를 한 곳에 정의하는 것이다. 바꾸어서 생각해보면 우리가 수정해야 할 코드가 한 곳에 모여있음을 의미한다. 이렇게 하면 수정을 하더라도 전체 프로그램 수행에 영향을 최소화할 수 있다. 즉, 유지보수가 어려운 코드를 만들려면 요소를 반복적으로 가능한 한 여러 장소에 기술해야 한다. 다행히도 자바와 같은 언어로 이와 같이 유지보수가 어려운 코드를 비교적 쉽게 작성할 수 있다. 예를 들어, 폭넓게 사용하는 변수는 여러 가지 형 변환 및 변환을 거치고 있으며, 관련 형식의 임시 변수가 사용되고 있을 가능성이 크므로 형식을 변환하기가 쉽지 않다. 더욱이 화면에 뭔가를 출력하는 변수일 경우라면 출력과 데이터 입력 관련 코드를 수동으로 수정해야 한다. C와 자바를 포함한 Algol 계층 언어는 데이터를 배열, 해시테이블, 파일, 데이터베이스에 저장하는 문법이 완전 다르다. Abundance와 같은 언어나 Smalltalk 확장 언어에서는 데이터 저장 문법은 같고 선언만 다르다. 따라서 자바의 부족한 기능을 공략하자. 현재 RAM으로 감당할 수 없이 크기가 커질 데이터를 배열로 저장하자. 그러면 유지보수 프로그래머는 나중에 배열을 파일로 바꿔야만 하는 악몽같은 작업을 피할 수 없을 것이다. 마찬가지로 데이터베이스에 작은 파일을 사용하자. 그러면 유지보수 프로그래머는 성능 최적화 때 해당 파일을 배열 접속 방식으로 바꿔야 하는 즐거운 경험을 맛볼 것이다.

자바 형변환

자바의 형변환 스킴은 하느님의 귀중한 선물이다. 형변환은 언어에서 필요한 기능이므로 이를 아무 거리낌없이 남용할 수 있어야 한다. Collection에서 오브젝트를 가져왔으면 원래 형식으로 형변환시켜야 한다. 어떤 때는 변수 형식 종류가 수십이 넘기도 한다. 나중에 데이터의 형식을 바꾸려면 모든 형변환도 바꿔야 한다. 운이 없는 유지보수 프로그래머가 모든 형변환을 적절하게 처리하지 못한 경우(혹은 너무 많은 변환을 한 경우) 컴파일러가 그 사실을 알려줄 수도 있지만, 그렇지 못한 경우도 있다. 마찬가지로 변수 형식이 short에서 int로 변경하면 관련 형변환도 모두 (short)에서 (int)로 바꿔야 한다. 일반 캐스트 연산자인 (cast)와 일반 변환 연산자 (convert)라는 새로운 연산자에 대한 필요성이 논의되고 있다. 이들 연산자는 변수 형식이 변경되어도 유지보수 할 필요성이 없게 해주는 새로운 연산자가 될 전망이다. 이런 이단적인 연산자가 언어 스펙에 포함되게 보고만 있어야 하는가? RFE 114691에서 형변환의 필요성을 제거하기 위한 genericity 부분에 적극 투표하길 바란다.

자바의 중복성 남용하기

자바에서는 모든 형식을 두 번 지정해야 한다. 자바 프로그래머는 이러한 관습에 익숙하기 때문에 두 형식을 아래처럼 살짝 바꾸어 놓아도 눈치챌 수 있는 사람은 많지 않다.

```
Bubblegum b = new Bubblegom();
```

불행히도 ++ 연산자의 대중성 때문에 다음과 같은 의사 중복 코드를 성공시키기가 쉽지 않다.

```
swimmer = swimner + 1;
```

검증을 멀리하라

입력 데이터에 대한 어떤 종류의 불일치 검사나 정확성 검사를 수행하지 않는다. 즉, 우리는 회사 장비를 온전히 신뢰하고 있으며 모든 프로젝트 파트너와 시스템 오퍼레이터를 신뢰하는 완벽한 팀원임을 보여줄 수 있다. 입력 데이터가 이상하거나 문제가 있는 듯 보이더라도 항상 합리적인 값을 반환하기 위해 노력해야 한다.

예의를 지키고 무턱대고 주장(Assert)하는 일을 피하라

assert() 메커니즘은 3일짜리 버그 축제를 10분짜리로 만들어 버릴 수 있으므로 피해야 한다.

캡슐화를 멀리하라

효율성 측면을 고려할 때 캡슐화를 멀리해야 한다. 메소드 호출자는 메소드가 어떻게 동작하는지를 알 권리가 있다.

복사하고 수정하라

효율성이라는 명목으로 잘라내기/붙이기/복사하기/수정하기를 남발하자. 이 방식은 작은 재사용 가능한 모듈 여럿을 사용하는 것보다 실행 속도가 빠르다는 장점이 있다. 특히 이 방식은 우리가 작성하는 코드 라인 수를 업무 진행 척도로 여기는 곳에서 일할 때 유용하다.

정적 배열을 사용하라

라이브러리의 모듈에 이미지를 저장할 배열이 필요한 경우에는 정적 배열을 선언해야 한다. 아무도 512 x 512 크기 이상의 이미지를 사용하지 않을 것이므로 크기가 고정된 배열도 좋다. 정확성을 높일 수 있도록 double 배열을 사용하는 것도 바람직하다. 이렇게 하면, 2 메가 크기의 정적 배열을 효과적으로 숨길 수 있다. 클라이언트는 우리가 만든 루틴을 한번도 수행하지 않았음에도 미친 것처럼 허우적대고 프로그램은 결국 클라이언트의 메모리를 초과할 것이다.

더미 인터페이스

"WrittenByMe"와 같은 빈 인터페이스를 만들고 모든 클래스에서 "WrittenByMe" 인터페이스를 구현하도록 하자. 그리고 우리가 사용하는 자바의 내장 클래스 래퍼 클래스를 만들자. 핵심은 우리 프로그램의 모든 오브젝트가 위 인터페이스를 구현하게 하는 것이다. 마지막으로 모든 메소드를 직접 구현하므로 매개변수와 반환 형식을 WrittenByMe가 구현하는 것이 목적이다. 이 기법을 사용하면 특정 메소드가 어떤 작업을 수행하는지 알아내기 어렵게 할 수 있고 아주 다양한 종류의 형변환 이라는 즐거움을 만끽할 수 있다. 이를 좀 더 활용해서 각 팀 멤버에게 자신만의 개인 인터페이스(예를 들어, WrittenByJoe)를 만들어줄 수 있다. 그리고 각자가 작업한 클래스는 자신만의 인터페이스를 구현하게 한다. 자 이제 의미 없는 수많은 인터페이스 중에 아무 인터페이스를 골라잡아 오브젝트 참조에 사용할 수 있다.

거대 리스너

각 컴포넌트에 리스너를 개별적으로 만들지 않는다. 우리 프로젝트의 모든 버튼의 이벤트를 처리할 리스너를 단 한 개로 통일하고 수많은 if…else문으로 각 버튼 동작을 처리하는 것이 바람직하다.

좋은 것™ 은 남용하라

캡슐화와 oo를 남용하라. 예를 들면,

```
myPanel.add( getMyButton() );
private JButton getMyButton()
      {
      return myButton;
      }
```

위 코드에 특별히 흥미로운 부분은 없어 보인다. 걱정할 필요 없다. 언젠가는 재미있는 일이 일어날 것이다.

우호적인 친구

C++에서는 가능한 한 자주 friend-선언을 사용한다. 생성된 클래스의 클래스 생성 포인터 처리와 결합하는 것도 좋은 방법이다. 그럼 인터페이스를 생각하는데 시간을 낭비할 필요성이 없어진다. 또한 private와 protected 키워드를 사용해 클래스를 캡슐화 할 수 있다.

삼차원 배열을 사용하라

삼차원 배열을 적극 사용하자. arrayA의 행 데이터를 arrayB의 열에 채우기와 같이 배열간의 데이터를 이동할 때는 복잡한 방법을 사용할수록 좋다. 특별한 이유 없이 오프셋을 0이 아닌 1로 사용한다면 유지보수 프로그래머를 불안하게 만들 수 있다.

혼합과 매치

가능하면 accessor 메소드와 public 변수를 함께 사용한다. 이 방식을 사용하면 accessor를 호출하지 않고도 오브젝트 변수를 변경할 수 있다. 그러나 여전히 우리 클래스는 "자바 빈"이라고 말할 수 있다. 이 방법은 누가 변수 값을 변경하는지 알아내려고 로깅 기능을 추가한 유지보수 프로그래머를 좌절시킬 수 있다는 장점도 제공한다.

감싸고, 감싸고, 감싸라

우리가 구현하지 않은 코드를 우리 메소드에 사용해야 할 때에는 다른 더러운 코드에 우리 코드가 오염되지 않도록 적어도 한 번 이상 래퍼 레이어를 사용해야 한다. 어쩌면 다른 부분의 저자도 언젠가 모든 메소드의 이름을 자기 마음대로 바꾸어 버릴지 모를 일이다. 그럼 우리는 어떻게 대처해야 할까? 이러한 경우 래퍼를 만들어 우리 코드를 보호하거나 VAJ가 전체적인 이름 변경을 처리하게 할 수 있다. 한편 이는 간접적으로 래퍼 레이어를 통해 어떤 멍청한 일을 저지르기 전에 그를 제거할 구실을 제공하는 기회로 삼을 수 있다. 자바의 주요 문제점 가운데 하나는 별다른 작업을 수행하지 않고 다른 메소드의 같은 이름 또는 밀접히 연관된 이름을 호출하는 더미 래퍼 메소드 없이는 간단한 문제도 해결하기 힘들다는 점이다. 즉, 우리는 아무 작업도 하지 않는 4단계의 래퍼를 쥐도 새도 모르게 만들 수 있다. 소스 코드 혼잡성을 극대화하려면 각 단계에서 메소드 이름을 변경하고, 랜덤으로 유의어 사전에서 동의어를 선택할 수 있다. 이와 같은 방법을 이용해 마치 뭔가가 일어나고 있다는 환상을 심어줄 수 있다. 나중에 이름을 변경하면서 프로젝트 용어의 일관성을 깨뜨릴 가능성도 커진다. 코드에서 래퍼의 각 단계를 건너뛰도록 해둠으로써 혹시라도 우리가 만든 여러 단계를 제거하려는 시도를 방지할 수 있다.

감싸고 감싸고 감싸고 더 감싸라

모든 API 함수를 적어도 6~8번은 감싸야 하고 다른 소스 파일에서 함수를 정의해야 한다. #define으로 이들 함수를 연결하는 것도 좋은 방법이다.

비밀은 없다!

언젠가 모두가 사용할 수 있도록 모든 메소드와 변수를 public으로 선언하라. 메소드를 public으로 선언한 이후에는 메소드 기능을 축소하기가 어렵다. 즉, 내부 동작을 수정하기가 매우 어려워진다. 이 기법을 사용하면 클래스의 목적이 무엇인지를 알기 어렵게 하는 부수적 효과를 얻을 수 있다. 상사가 우리에게 미친 것 아니냐고 얘기한다면, 우리는 그저 투명한 인터페이스라는 고전 원칙을 따르고 있을 뿐이라고 변명하면 된다.

카마수트라

이 기술로는 유지보수 프로그래머 뿐만 아니라 사용자와 문서화 담당자의 집중을 방해하는 효과를 발휘할 수 있다. 같은 메소드에 수십 개 이상의 오버로드를 이용한 변형을 생성해 약간만 다른 기능을 하게 만든다. 소설가 오스카 와일드는 카마수

트라의 47과 115번 자세에서 115번의 여자의 중지와 인지 손가락을 포갠 것을 제외하면 같은 자세라는 사실을 발견한 것 같다. 패키지 사용자가 여러 변형 버전 가운데 어떤 것을 사용해야 할지 선택하려면, 먼저 길고 긴 메소드 목록을 정독해야 한다. 동시에 문서화해야 할 양도 크게 늘어나기 때문에 문서를 최신으로 유지하기 힘들게 된다. 상사가 우리에게 왜 이런 짓을 하는 것이냐고 묻는다면, 사용자의 편의성을 개선하기 위해서라고 설명하자. 공통 로직을 복제한 다음 복사본의 내용이 더 이상 동기화되지 않을 때까지 기다리면 효과가 더욱 극대화 된다.

치환으로 당황시키기

drawRectangle(height, width)라는 메소드가 있다면 다른 부분은 건드리지 말고 파라미터 순서만 drawRectangle(width, height)처럼 역순으로 바꿔보자. 그리고 몇 번의 릴리즈가 일어난 다음에 원상 복귀시킨다. 유지보수 프로그래머는 그런 변경이 일어났는지 쉽게 구별하기가 어려울 것이다. 일반화 문제는 독자 여러분에게 숙제로 남겨둔다.

테마와 변형

하나의 메소드에 파라미터를 사용하기 보다는 되도록이면 여러 메소드를 만드는 것이 좋다. 예를 들어 왼쪽, 오른쪽, 가운데 정렬을 의미하는 상수를 파라미터로 넘겨줄 수 있는 setAlignment(int alignment)보다는 setLeftAlignment, setRightAlignment, setCenterAlignment와 같이 세 개의 메소드를 정의하는 것이 바람직하다. 마찬가지로 공통 로직을 복제해서 동기화를 어렵게 만들면 효과가 커진다.

Static이 좋다

가능하다면 변수를 static으로 만들자. 우리 프로그램에서 클래스 인스턴스가 한 개 이상 필요하지 않으면 다른 이들도 마찬가지일 것이다. 프로젝트의 다른 코더가 이에 대해 불평한다면 이 방법 덕분에 실행속도가 빨라졌음을 알려주자.

카길사의 진태양난

카길사의 진퇴양난을 이용하자. "적절한 수준의 부정 수단을 이용한다면 모든 디자인 문제를 해결할 수 있다". 프로그램 상태를 갱신하는 메소드를 찾을 수 없을 때까지 OO 프로그램을 분해하자. 모든 결과물을 전체 시스템 내에서 사용하는 모든 함수 포인터를 포함하는 포인터 포레스트를 탐색한 결과에 대한 콜백으로 활성화 할 수 있게 정렬하면 더욱 좋다. 포레스트 탐색 정렬을 활성화하면 깊은 복사(실제로는 그렇게 깊지 않지만)해서 만들어진 오브젝트 레퍼런스를 해제하는 과정에서 부작용을 일으키게 할 수 있다.

잡동사니 수집

사용하지 않고 오래된 메소드나 변수라 할지라도 모두 코드에 모아준다. 1976년에 한번 사용했던 적이 있는 코드라도 언제 어떻게 사용해야 할지 누가 알겠는가? 항상 프로그램의 변경 사항을 관리하므로 "바퀴를 다시 발명하는 일은 피해"(상사는 이런 말을 좋아한다)야 한다. 메소드와 변수 주석을 수수께끼처럼 남겨둔다면, 그 코드를 유지보수해야 할 누군가는 코드를 보고 겁부터 집어먹을 것이다.

Final이 주는 즐거움

모든 최종 클래스를 final로 정의하라. 이렇게 프로젝트를 마무리할 수 있다. 즉 아무도 우리가 만든 클래스를 확장해 작업을 개선시킬 수 없다. 누군가가 우리 클래스를 확장하면서 발생하는 보안 취약성도 방지할 수 있다. 마찬가지로 java.lang.String이 final인 이유와 같은 맥락이다. 프로젝트의 다른 코드가 불평한다면 언제나처럼 이 기법으로 인해 얻고 있는 속도 향상에 대해 얘기해주면 된다.

인터페이스를 피하라

자바에서 인터페이스는 무시해야 할 존재다. 관리자가 이에 대해 불평한다면 자바의 인터페이스는 우리로 하여금 같은 인터페이스를 구현하는 다른 클래스에서 "잘라내고 붙이기"를 반복하게 만드는 존재이며 이는 당신도 알다시피 유지보수를 어렵게 하는 것이라고 말하자. 인터페이스 대신 자바 AWT 설계자의 방식을 따를 수 있다. AWT 설계자는 특정 클래스와 해당 클래스를 상속하는 클래스에서만 사용할 수 있는 많은 기능을 추가하고 각 메소드에서는 "instanceof"를 수시로 활용했다. 누군가가 우리 코드를 재사용하고 싶어한다면 우리 클래스를 상속받도록 할 수 있다. 혹시 두 개의 다른 클래스를 모두 재사용하고 싶어하는 이가 있다면, 불행히도 그렇게 하는 것은 불가능하다! 어쩔 수 없이 인터페이스가 필요한 상황이라면 "ImplementableIface"와 같은 하나의 인터페이스를 만들어 다용도로 사용할 수 있게 해야 한다. 인터페이스를 구현하는 클래스명에 "Impl"을 붙이는 것은 학계에서 떠도는 유행 중 하나다. Runnable을 구현하는 클래스 등에서 유용성이 커진다.

레이아웃을 피하라

절대 레이아웃을 이용하지 말라. 그러면 유지보수 프로그래머가 필드 하나를 추가하려고 화면에 나타나는 모든 다른 콤포넌트의 좌표를 일일이 수정하게 만들 수 있다. 상사가 레이아웃을 사용하도록 강요한다면, 거대한 단일 GridBagLayout을 만들어서 절대 그리드 좌표로 하드 코딩하는 방법으로 우회하자.

환경 변수

다른 프로그래머가 사용할 클래스를 만들어야 한다면 환경 변수를 확인하는 코드(C++에서는 getenv(), 자바에서는 System.getProperty())를 클래스의 정적 초기화를 담당하는 메소드(이름없는)에 추가하자. 이러한 방법으로 모든 매개변수를 생성자를 거치지 않고 클래스에 전달할 수 있다. 초기화 담당 메소드를 사용하면 프로그램 바이너리를 로드하는 순간 호출된다는 장점이 있다. 즉, 프로그램의 main()이 호출되기도 전에 이와 같은 작업이 완료된다. 따라서 우리의 클래스를 직접 확인하기 전까지는 프로그램의 다른 부분에서는 파라미터 값을 고칠 방법이 없다. 오히려 사용자가 우리의 방식에 맞춰 자신의 환경 변수를 모두 설정하는 편이 편할 것이다.

테이블 기반 로직

테이블 기반 로직은 피해야 한다. 테이블 기반 로직은 너무 훤히 들여다보여서 최종 사용자는 바로 교정을 시작할 수 있고 곧 몸서리 칠 수 있다. 심지어는 직접 테이블을 수정할 수도 있다.

엄마의 필드를 수정하라

모든 기본형 파라미터는 값으로 전달되므로 자바에서는 이들을 읽기-전용으로 전달한다. 피호출자에서 파라미터 값을 수정할 수는 있지만, 호출자의 변수에는 영향을 미치지 않는다. 반대로 넘겨지는 모든 오브젝트는 읽고 쓸 수 있다. 레퍼런스는 값으로, 즉 오브젝트 자체는 레퍼런스로 전달된다. 피호출자는 오브젝트에 원하는 모든 동작을 수행할 수 있다. 메소드에서 넘겨진 파라미터의 각 필드를 수정하는지 여부를 절대 문서화하지 말자. 메소드에서 각 필드를 수정하는 경우에는 메소드 이름을 마치 보기만 할 것처럼 보이는 이름으로 위장해야 한다.

전역 변수의 마술

예외를 이용해 예외를 처리하는 것보다는 자신만의 에러 메시지 루틴 집합을 전역 변수로 갖는 것이 좋다. 시스템에서 오랫동안 수행되는 모든 루프에다 전역 플래그를 검사하고 문제발생시 종료하도록 코드를 추가하자. 사용자가 'reset' 버튼을 눌렀을 때 이를 알릴 수 있는 또 다른 전역 변수도 추가한다. 마찬가지로 시스템의 모든 루프에 이 두 번째 전역 변수를 확인하는 코드를 넣어야 한다. 이 때 요청시에도 종료하지 않는 루프 몇 개를 숨겨두는 것을 잊지 말자.

아무리 강조해도 지나치지 않을 그대 이름은 전역!

우리가 지역 변수를 사용하는 것을 원치 않았다면 신은 전역 변수라는 것을 창조하지 않았을 것이다. 따라서 가능한 한 많은 수의 지역 변수를 사용하므로 신을 실망시키지 말아야 한다. 특별한 이유가 없더라도 각 함수에서는 최소한 두 개 이상의 전역 변수를 사용해야 한다. 좋은 유지보수 프로그래머라면 이것이 일종의 검출 연습이라는 사실을 금방 깨닫게 될 것이다. 그리고 그는 진심을 다하는 유지보수 프로그래머와 취미로 하는 유지보수 프로그래머를 구별해주는 이 테스트를 즐길 것이다.

여러분! 전역에 대해 한 번 더 살펴봅시다

전역 변수를 사용하면 함수에서 매개변수를 지정하는 일을 생략할 수 있다. 이를 적극 활용하자. 전역 변수 중 몇 개를 선택해서 어떤 프로세스가 작업을 수행할지 지정하자. 유지보수 프로그래머는 바보처럼 C 함수에서 부작용이 발생하진 않을 것이라 생각할 수 있다. 이들에게 깜짝 놀랄 결과를 보여주자. 물론 내부 상태 정보는 지역 변수로 사라진다.

부작용

C에서 함수는 값이 변하지 않는 것(부작용 없이)으로 알려져있다. 이 정도면 충분한 힌트가 되겠는가?

철회

루프의 바디에서는 루프가 성공적으로 수행되고 있으며 모든 포인터 변수 값이 즉시 갱신되고 있다고 가정하라. 루프 동작 중에 예외가 발생한 경우에는 루프 바디 다음 부분에서 조건문을 이용해 포인터 값을 철회하도록 한다.

지역 변수

쓰고 싶은 마음이 굴뚝같이 들더라도 절대 지역 변수를 사용하지 말라. 자유롭게 클래스의 다른 메소드에서 공유할 수 있도록 하는 것보다는 인스턴스 변수나 static 변수로 만드는 것이 바람직하다. 이렇게 함으로 다음에 비슷한 정의를 갖는 다른 메소드를 작업할 때에는 시간을 절약할 수 있을 것이다.

줄이고, 재사용하고, 재활용하라

콜백(callback)에 사용할 데이터를 저장할 구조체를 정의해야 한다면, 그 구조체를 PRIVDATA라고 부르자. 모든 모듈은 자신만의 PRIVDATA를 정의할 수 있다. 이 구조체로 VC++의 디버거를 교란시킬 수 있다. 변수 watch 윈도우에 PRIVDATA 변수가 있는 상태에서 해당 변수를 펼치려고 하면 디버거는 어느 PRIVDATA를 의미하는 것인지 결정할 수 없어 아무것이나 선택한다.

설정 파일

설정 파일에는 보통 키워드=값 형태의 데이터를 저장한다. 프로그램을 로드할 때 이 값을 자바 변수로 읽어온다. 키워드와 자바 변수의 이름을 살짝 다르게 하는 방법으로 혼란을 일으킬 수 있다. 심지어 실행 중에 값이 변경되지 않는 상수도 설정 파일을 이용하자. 파라미터 파일 변수를 유지보수 하려면 간단한 변수를 유지보수 할 때보다 다섯 배나 많은 코드가 필요하다.

퉁퉁 부은 클래스

중요하지 않고 잘 알려지지 않은 메소드와 속성을 모든 클래스에 포함시킴으로써 외부에 둔감한 클래스를 만들 수 있다. 일례로, 천체 기하학적 궤도를 정의하는 클래스에 해수면 조류 스케쥴과 크레인(Crane) 날씨 모델을 구성하는 속성을 포함할 수 있다. 이와 같은 기법으로 클래스에 수많은 기능을 정의할 수 있을 뿐만 아니라 시스템에서 필요한 부분을 검색하는 작업을 서울에서 김서방 찾기처럼 힘들게 만들 수 있다.

자식 클래스에게 양보하는 미덕

객체 지향 프로그래밍은 유지보수 할 수 없는 코드를 작성하도록 하늘이 준 선물이다. 클래스에 10개 프로퍼티(멤버/메소드)를 갖고 있다 가정하자. 베이스 클래스는 하나의 프로퍼티를 갖고 있고, 베이스 클래스를 상속하는 클래스에서 한 개씩만 속성을 추가하는 방식으로 9 단계를 상속받도록 클래스 계층을 구성할 수 있다. 가장 하위 클래스로 10개 프로퍼티 모두를 사용할 수 있다. 가능하다면 각 클래스 선언을 다른 파일에 정의하는 것이 좋다. 이렇게 함으로써 INCLUDE나 USES 구문을 팽창시키는 부수효과를 얻을 수 있고 유지보수 프로그래머는 자신의 편집기에 더 많은 파일을 열어보아야 한다. 물론 적어도 각 서브클래스의 인스턴스를 한 개 이상은 만들어야 함을 잊지 말자.

코드 혼잡화

끈기를 가지고 혼잡성, 수퍼 복잡성, 장황함을 피하라.

읽을 수 없는 C

인터넷의 읽을 수 없는 C 경연대회에 참석하고 스승님의 책상다리 근처 앉아 경청하라.

방법을 찾거나 APL 권위자를 찾으라

이 세계에서는 코드는 간결하고 동작을 괴이할수록 더 많은 존경이 따라올 것이다.

나는 수십 개를 이용할 것이다

아무 문제없이 두 개 혹은 세 개를 이용할 수 있는 상황에서 문제를 처리하는 한 개의 변수만을 사용하지 말라.

알려지지 않은 자

같은 작업을 수행하더라도 가장 알려지지 않은 방식을 사용하라. 예를 들어, 배열을 이용해 정수를 문자열로 변경하는 대신 다음과 같은 코드를 이용할 수 있다.

```
char *p;
switch (n)
{
case 1:
    p = “one”;
    if (0)
case 2:
    p = “two”;
    if (0)
case 3:
    p = “three”;
    printf(“%s”, p);
    break;
}
```

일관성에 대한 고집은 아량이 없는 말썽쟁이 요정이나 하는 짓이다

‘ ‘, 32, 0x20, 040와 같이 하나의 문자 상수를 다양한 방법으로 표현할 수 있다. C나 자바에서는 10과 010을 동일시 하지 않는다는 사실을 자유롭게 활용하길 바란다.

형변환

모든 데이터는 void *로 넘겨서 적절한 구조체로 형변환해서 사용한다. 구조체로 형변환하지 않고 바이트 오프셋을 이용한 데이터 접근하는 것도 색다른 재미가 있다.

중첩된 Switch

중첩된 switch 구문(switch문 내의 switch)은 사람을 가장 혼란스럽게 하는 중첩 가운데 하나다.

암시적 변환을 악용하라

프로그래밍 언어에서 제공하는 모든 미묘한 암시적 변환 규칙을 기억한 다음 그들을 최대한 활용하자. 절대 picture(코볼이나 PL/I에서) 변수를 사용하지 말고 일반적인 변환 루틴(C의 sprintf와 같은)도 사용하지 말자. 소수점 변수를 배열 인덱스로, 문자를 루프 카운터로, 문자열 함수를 숫자에 사용해보자. 잘 정의한 함수 덕분에 우리 코드는 한결 단순해진다. 우리가 만든 코드를 유지보수 할 프로그래머는 암시적 데이터 형 변환 전체 장을 읽어야만 하는 기회를 제공해 준데 대해 우리에게 감사할 것이다. 물론 대부분의 프로그래머가 이전에도 암시적 데이터 형 변환을 살펴봤을 테지만, 아마도 다시 공부해야 할 필요성을 절실히 느꼈을 것이다.

정수(int) 그대로

ComboBox를 사용하면서 switch 구문이 필요할 때에는 이름을 붙인 상수를 사용하는 것보다는 정수를 그대로 사용하는 것이 바람직하다.

세미콜론!

문법적으로 허용된 곳이라면 어디에든 세미콜론을 사용하라. 아래 예를 살펴보자.

```
if(a);
else;
      {
      int d;
      d = c;
      }
      ;
```

8진법을 활용하라

아래 예에서 볼 수 있듯이 10진수 리스트에 8진수를 몰래 추가하는 것이 핵심이다.

```
array = new int []
      {
      111,
      120,
      013,
      121,
      };
```

간접적으로 변환하라

자바는 변환이 필요할 때마다 코드를 혼잡하게 만들 기회를 제공한다. 간단한 예로 double을 String으로 변환해야 한다면 Double.toString(d)보다는 우회적인 방법을 사용해서 new Double(d).toString()와 같이 하는 것이 좋다. 물론 예제에서 설명한 것보다 더 우회적인 방법을 사용해도 된다! 변환 대필자가 제안하는 변환 기법은 되도록 피해야 한다. 변환하면서 생긴 임시 오브젝트로 (heap) 메모리를 낭비하므로 추가 점수를 획득할 수 있음을 잊지 말자.

중첩

가능한 한 깊이 중첩하라. 훌륭한 독자는 한 라인에 사용한 10개의 ()를 이해할 수 있고, 하나의 메소드에 사용한 20개의 {}도 이해할 수 있다. C++의 경우 전처리기 중첩을 이용할 수 있다. 전처리기 중첩은 코드의 중첩 구조와는 독립적이므로 보다 강력한 옵션이다. 인쇄된 코드에서 블록의 시작과 끝이 같은 페이지에 나오지 않게 할 수 있다면 추가 점수를 획득할 수 있다. 가능하다면 중첩된 if 문을 중첩된 [A?B…] 구문으로 바꾸는 것이 바람직하다. 게다가 한 줄 이상의 코드로 확장할 수 있다면 금상첨화다.

숫자 기호

100개의 요소를 갖는 배열이 있다면 최대한 100이라는 기호를 프로그램에서 가능한 많이 사용해야 한다. 절대 static final을 붙인 상수로 100을 대체하거나 myArray.length와 같은 방법을 사용하지 말아야 한다. 상수를 이해하고 대체하기 어렵게 하려면 100/2보다는 50, 100-1보다는 99와 같이 표기하는 것이 좋다. a 〉 100 보다는 a == 101을 사용하고 a 〉= 100 보다는 a 〉 99을 사용하므로 100이라는 숫자가 사용되지 않은 것처럼 위장할 수 있다.

헤더에 x개, 바디에 y개, 푸터에 z개 라인을 사용한 페이지 크기가 있다고 가정해보자. 이 때 난독화를 각각에 적용할 수 있으며 부분합 혹은 전체합에도 적용할 수 있다.

이 유서 깊은 기법은 특히 100개의 요소를 갖는 연관성이 없는 두 배열을 갖는 프로그램에 효과적이다. 유지보수 프로그래머가 둘 중 하나의 배열 크기를 변경하려면 프로그램에서 사용된 모든 100이라는 기호를 해독해서 어떤 배열에 해당하는 것인지 이해해야 한다. 이러한 과정에서 그가 적어도 한 번 이상 실수를 저지를 것이라고 장담할 수 있다. 운이 좋다면 그 에러가 몇 년이 지난 후에야 발견될 수도 있을 것이다.

더 사악한 변형 기법도 있다. 매우 가끔 “우발적으로” 100을 사용하므로 유지보수 프로그래머를 안심시킬 수 있다. 가장 사악한 방법은 100이나 이름을 붙인 상수를 쓰지 않고 산발적으로 우연히 100 값을 갖는 상수를 사용하는 것이다. 당연히 배열의 크기를 연상시키는 어떠한 이름도 사용하지 말아야 한다.

C의 기이한 배열 접근법

C 컴파일러는 myArray[i]를 *(myArray + i)로 변환하는데 이는 *(i + myArray)과 같고 이는 i[myArray]과 같다. 전문가들은 이와 같은 사실을 어떻게 활용해야 할지 알고 있다. 인덱스를 생성하는 함수를 이용하므로 이 사실을 숨길 수 있다.

```
int myfunc(int q, int p) { return p%q; }
...
myfunc(6291, 8)[Array];
```

불행히도 이 기법은 네이티브 C 클래스에서만 사용할 수 있으며 자바에서는 사용할 수 없다.

긴 줄

가능하면 많은 내용을 한 줄에 담으려 노력하라. 이 기법은 임시 변수를 줄이고, 줄 바꿈 문자나 공백 문자를 제거함으로써 소스 파일 크기를 줄이는 효과를 제공한다. 좋은 프로그래머는 몇몇 편집기의 한 줄에 들어갈 수 있는 문자 크기의 한계인 255 문자까지 활용하기도 한다. 글자 크기를 6으로 맞추었을 때 글자를 읽지 못한다면 스크롤해서 읽도록 길게 코딩하는 것이 정석이다.

예외

거의 알려지지 않은 코딩 비밀을 공유하려고 한다. 예외의 이면에는 고통이 숨겨져 있다. 제대로 구현한 코드는 실패할 일이 없으므로 예외 자체가 필요 없게 된다. 따라서 예외에 시간낭비 할 필요가 없다. 예외를 상속하는 것은 자신의 코드에 문제가 있음을 인정하는 무능력자가 하는 짓이다. 일반적으로는 전체 응용 프로그램을 감싸는 하나의 try/catch를 사용(main에서)하고 문제가 생겼을 때 System.exit()를 호출함으로써 프로그램을 단순하게 만들 수 있다. 예외를 발생시킬 수 있는지 여부와 관계없이 모든 메소드 헤더에 표준 throws 구분을 추가하자.

언제 예외를 사용해야 하는가?

예외가 발생하지 않는 상황에서만 예외를 사용하라. 주기적으로 루프를 종료하면서 ArrayIndexOutOfBoundsException를 발생시킬 수 있다. 그리고 아무 일도 없는 것처럼 예외가 발생한 메소드에서 표준 결과값을 넘긴다.

스레드에 떠넘기라

제목 그대로 행동하라.

변호사 코드

뉴스 그룹에서 까다로운 코드가 어떻게 수행돼야 하는지를 두고 논쟁을 하는 경우가 있다(예를 들어, a=a++;, f(a++,a++);). 이와 같이 논쟁이 벌어지는 코드를 우리 프로그램에 자유롭게 사용하자. 아래와 같은 선/후 감쇠 코드를 어떻게 처리해야 하는지는 언어 스펙에서 정의하지 않는다.

```
*++b ? (*++b + *(b-1)) 0
```

따라서 컴파일러마다 각자 임의로 위 수식을 평가한다. 비슷한 방법으로 모든 공백을 없애서 C와 자바의 복잡한 토큰화 규칙을 이용할 수도 있다.

중간 반환문

Goto를 사용하지 않고, 중간에 반환하지 않으며, label을 사용하지 않는다는 규칙을 엄격히 지켜야 한다. 그러나 우리가 최소한 5 단계 이상으로 중첩하는 if/else문을 만들 수 있다면 이 규칙을 꼭 지키지 않아도 된다.

Avoid {}

문법적으로 어쩔 수 없는 경우가 아니라면 절대로 if/else 블록을 감싸는 괄호 { }를 사용하지 말자. 들여쓰기 없이 if/else와 블록을 여러 단계로 중첩한다면 전문 유지보수 프로그래머라도 가볍게 해치울 수 있다. 펄(Perl)을 사용하면 이 기법의 효과가 극대화된다. 일반 코드 뒤에 if문을 양념으로 추가하는 것도 좋은 방법이다.

지옥에서 온 탭

탭을 과소평가하지 마라. 탭이 얼마의 공간을 나타내야 하는지에 대한 표준이 없는 회사에서는 공백 대신에 탭을 사용하므로 큰 혼란을 야기할 수 있다. 스트링 문자에 탭을 추가하거나 공백을 탭으로 변환해주는 툴을 돌리는 것도 좋은 방법이다.

마법 같은 행렬 위치

특정 행렬 값을 플래그로 사용해보자. 변환 행렬에서 동종 좌표 시스템을 이용한 [3][0]에 위치한 요소 등은 좋은 예이다.

마법 같은 행렬 슬롯의 개선

주어진 형식의 여러 변수가 필요할 때에는 변수를 배열로 정의하고 숫자로 각 변수를 접근할 수 있다. 이 때 숫자 규칙은 자신만이 아는 것으로 정하고 문서화하지 않는다. 또한 고생스럽게 #define 상수로 인덱스를 정의할 필요도 없다. 전역 변수 widget[15]는 취소 버튼이라는 사실은 누구나 당연히 알고 있어야 한다. 이는 어셈블러 코드에서 절대적 숫자 주소를 접근할 때 최근 이용하는 방식이다.

아름다움을 멀리하라

절대 자동 코드 정렬 기능으로 코드를 정렬하지 말자. PVCS/CVS(버전 관리를 이력을 추적하는)에서 가짜 정보를 만드는 이와 같은 도구를 사용하지 않도록 회사 현장에서 로비를 하자. 또는 모든 프로그래머는 자신이 만드는 모듈에 영원히 신성불가침한 자신만의 들여쓰기 방식을 가져야 한다고 주장하자. 다른 프로그래머에게는 이러한 방식을 따르더라도 각자가 만든 모듈에서만 이와 같은 특유의 규칙이 나타나므로 별로 신경 쓸 일이 아니라고 설득하자. 아름다움을 멀리하는 방법은 간단하다. 아름다움을 멀리하므로 수동 정렬로 인한 수백만 번의 키 입력을 아낄 수 있고, 정렬이 엉망인 코드를 오역하는 바람에 며칠을 낭비하는 일도 사라진다. 공통 저장소에 코드를 저장할 때 뿐 아니라 편집할 때에도 모두가 같은 정렬 형식을 사용해야 한다고 주장하자. RWAR과 상사를 먼저 설득하자. 아마도 평화를 위해 자동 정렬 기능은 사용 금지될 것이다. 자동 정렬이 금지되었으면 루프나 if문의 바디가 실제보다 길어 보이거나 짧아 보이게 정렬할 수 있다. 혹은 else 문을 다른 if와 연관되는 것처럼 보이게 할 수도 있다. 물론 모든 일은 우발적으로 일어나는 것일 뿐이다.

```
if(a)
    if(b) x=y;
else x=z;
```

매크로 전처리기

매크로 전처리기는 코드 난독화를 할 좋은 기회를 제공한다. 핵심은 매크로 확장을 여러 단계에 거쳐 확장하고 여러 *.hpp 파일을 이용해야 뜻을 파악할 수 있게 만드는 것이다. 실행할 코드를 매크로에 넣은 다음 모든 *.cpp 파일에서 이 매크로를 가져다 사용하므로(매크로를 쓰지 않더라도 이렇게 하는 것이 좋다) 코드가 변경될 때마다 다시 컴파일 해야 할 코드 양을 극대화 할 수 있다.

일관성 부족을 악용하라

자바에서 배열 선언은 정말 어지럽다. 예전 C 형식으로 할 수 있고, String x[]와 같은 방식도 가능하며(배열 표시 위치가 앞-뒤로 혼합된 형태), String[] x(배열 표시가 앞쪽에 오는 형태)와 같이 선언할 수 있다. 사람들에게 혼란을 주려면 다음과 같이 여러 표기법을 혼합한다.

```
byte[ ] rowvector, colvector , matrix[ ];
```

위 코드는 다음과 같다.

```
byte[ ] rowvector;
byte[ ] colvector;
byte[ ][] matrix;
```

에러 복구 코드를 숨겨라

중첩을 특정 함수 호출의 에러를 복구하는 함수를 가능한 한 멀리 배치할 수 있다. 아래 간단한 예제를 좀 더 수정하면 10단계나 12단계 중첩으로 확장할 수 있다.

```
if ( function_A() == OK )
    {
    if ( function_B() == OK )
        {
        /* 일반 종료의 경우에 처리 코드 */
        }
    else
        {
        /* Function_B에 대한 에러 복구 코드 */
        }
    }
else
    {
    /* Function_A에 대한 에러 복구 코드 */
    }
```

가짜(pseudo) C

#define 의 본래 목적은 다른 프로그래밍 언어에 익숙한 사람이 C로 쉽게 변환할 수 있도록 하는 것이다. 따라서 #define begin { “ 또는 “ #define end }과 같은 선언문을 이용하면 아주 신선한 코드를 쉽게 만들 수 있다.

혼란을 안겨주는 import

패키지에서 우리가 어떤 메소드를 사용하는 것인지 유지보수 프로그래머에게 직접 알려주지 말고 추측하게 할 수 있다. 즉 아래와 같이 명시하지 말고,

```
import MyPackage.Read;
import MyPackage.Write;
```

이렇게 사용한다.

```
import Mypackage. *;
```

절대 클래스나 메소드에 패키지명을 포함한 전체 이름을 사용하지 않는다. 유지보수 프로그래머가 우리 코드를 보면서 각각이 어떤 패키지/클래스에 속한 것인지 추측하게 만들자. 물론 경우에 따라 전체 패키지 이름을 사용하거나 import 구문 사용방식을 바꾸어 주면 더욱 효과적이다.

변기 배관

어떤 경우에도 함수나 프로시저가 한 화면에 모두 보이도록 하지 말아야 한다. 간단한 루틴에는 아래 설명하는 기법을 사용하자.

일반적으로 코드의 논리적인 블록을 구분하는데 빈 줄을 사용한다. 우리 코드의 모든 라인은 각각이 논리적인 블록이다. 따라서 모든 줄에 빈 줄을 추가한다.

주석을 코드 끝에 추가하지 말고 윗부분에 추가하라. 어쩔 수 없는 강요에 의해 코드의 뒷부분에 추가해야만 하는 상황이라면 전체 파일에서 가장 긴 줄을 찾아 10개의 공백을 넣고 모든 end-of-line 주석을 왼쪽 정렬한다.

프로시저의 윗부분의 주석에 사용할 템플릿은 적어도 15줄 이상이어야 하고 빈 줄도 자유롭게 추가한다. 아래는 간단한 예제 템플릿이다.

```
/*
/* 프로시저 이름:
/*
/* 원래 프로시저 이름:
/*
/* 저자:
/*
/* 생성일:
/*
/* 수정일:
/*
/* 수정한 사람:
/*
/* 원래 파일명:
/*
/* 목적:
/*
/* 의도:
/*
/* 직함:
/*
/* 사용한 클래스:
/*
/* 상수:
/*
/* 지역변수:
/*
/* 파라미터:
/*
/* 만든 날짜:
/*
/* 목적:
*/
```

문서에 여러 번 중복 정보를 넣는 기법을 사용하면 해당 정보를 최신으로 유지하기 어려워 진다. 이를 순진하게 믿는 유지보수 프로그래머는 금방 혼란에 빠질 것이다.

테스트

나는 에러를 수정하는 모뎀이 있으므로 따로 프로그램을 테스트할 필요가 없다.
– Om I. 바우드(Om I. Baud)

프로그램에 버그를 남겨둠으로써 유지보수 프로그래머에게도 재미있는 일거리를 제공해야 한다. 잘 만든 버그라면 어디서 어떻게 발생했는지에 관한 단서를 남기지 않는다. 버그를 남겨두는 가장 게으른 방법으로는 우리 코드를 절대 테스트 하지 않는 방법도 있다.

절대 테스트하지 마라

에러나, 기기 크래쉬, OS 결함을 처리하는 코드는 절대 테스트 하지 않는다. OS가 반환하는 코드도 검사하지 않는다. OS가 반환하는 코드는 실행에 아무 도움이 되지 않으며 우리 테스트 시간만 오래 걸리게 한다. 게다가 우리 코드가 디스크 에러, 파일 읽기 에러, OS 크래쉬와 같은 모든 경우를 적절하게 처리하는지 어떻게 일일이 테스트 할 수 있겠는가? 도대체 왜 컴퓨터 시스템을 신뢰할 수 없는 것처럼 생각하고 교수대 같은 것이 제대로 동작하지 않는지 테스트해야 하는지 이해할 수가 없다. 최신 하드웨어에서는 에러가 발생하지 않는다. 그러나 아직도 누군가는 테스트 전용 코드를 구현하는가? 정말 지치는 일이 아닐 수 없다. 사용자가 우리 프로그램의 문제에 대해 불평한다면 사용자가 잘 알 수 없는 OS나 하드웨어 탓으로 떠넘기자.

세상이 무너져도 성능 테스트를 하지 않는다

프로그램이 좀 느리다고? 고객에게 더 빠른 컴퓨터를 사라고 말하자. 성능 테스트를 수행했다면, 문제가 일어나는 지점을 찾았을 것이다. 아마 문제를 해결하려면 알고리즘을 변경해야 할 것이고 제품 전체를 완전히 다시 설계해야 하는 경우도 생길 수 있다. 이런 일을 누가 하고 싶어 하겠는가? 게다가 고객사에 성능 문제가 불쑥 나타난다는 것은 이국적인 곳으로 공짜 여행을 할 수 있는 기회일 수 있다. 정말로 우리가 해야 할 일은 여권을 항상 가까이에 소지하면서 상황을 예의주시하는 것이다.

절대 테스트 케이스를 만들지 않는다

코드 커버리지나 패스 커버리지 테스트를 절대 수행하지 말자. 자동화 테스트는 겁쟁이나 하는 짓이다. 우리 루틴의 90%에 해당하는 기능을 찾고, 전체 테스트의 90%를 이러한 기능 확인에 사용한다. 결과적으로 이 기법으로는 우리 코드의 약 60% 정도만 검증을 한 셈이 되고 40% 정도의 노력을 절약할 수 있다. 절약한 시간으로 프로젝트의 백엔드 계획을 수립할 수 있다. 멋진 "마케팅 기능" 이 동작하지 않는다는 사실을 알아차리기 전에 회사를 조용히 떠날 수 있다면 정말 짜릿한 경험이 될 것이다. 크고 유명한 소프트웨어 회사는 이런 방식으로 코드를 테스트한다. 따라서 위 방법을 사용할 것을 추천한다. 어떤 이유에서인지 모르겠지만 혹시 위 계획을 실천하지 못하고 아직 회사에 남아있어야 한다면 계속해서 이 문서를 읽기 바란다.

겁쟁이 전용 테스트

용감한 코더는 이 단계를 건너뛴다. 상사를 무서워하고, 직장을 잃을까 두려워하고, 고객의 불평 메일을 받거나 고소당하는 일을 걱정하는 프로그래머가 너무 많다. 이러한 두려움 때문에 행동에 제약을 받고, 생산성이 떨어진다는 것은 누구나 아는 사실이다. 연구에 따르면 테스트 단계를 없애는 것이 관리자로 하여금 출시일을 미리 결정할 수 있게 하는 등 계획 단계부터 많은 도움이 된다. 두려움이 없어진다면 혁신과 실험정신이 창궐할 수 있다. 프로그래머는 코드를 생산하는데 주력할 수 있고, 헬프 데스크와 기존 유지보수 그룹이 협력해서 디버깅을 담당할 수 있다.

우리의 코딩 능력을 온전히 믿는다면 테스트 따위는 더 이상 불필요하다. 논리적으로 생각해보면 테스트라는 것은 감정적인 자신감 결여 문제일 뿐, 기술적으로 문제를 해결하지도 않는다는 점을 바보라도 눈치챌 수 있다. 자신감 결여 문제는 테스트

과정을 완전 제거하고 프로그래머를 자신감 회복 과정에 보냄으로 해결할 수 있다. 테스트를 하려면 프로그램을 변경할 때마다 테스트를 해야 한다. 프로그래머를 자신감 향상 프로그램에 보내는 것과 테스트를 하는 것 어떤 것이 바람직한 일인가? 비용 절감 효과는 상상을 초월할 것이다.

디버그 모드에서만 동작하는 코드

TESTING을 1로 정의했다면,

```
#define TESTING 1
```

아래와 같이 TESTING이 1인 경우에만 수행되는 별도의 코드 섹션을 가질 수 있다.

```
#if TESTING==1
#endif
```

별도의 코드 섹션에 다음과 같은 꼭 필요한 코드를 넣고 빼는 것은 우리의 자유다.

```
x = rt_val;
```

누군가 TESTING 을 0으로 재설정하면 프로그램은 동작하지 않는다. 조금만 창의력을 발휘하면 이 기법으로 로직을 망가뜨리고 컴파일러 기능 장애를 초래할 수 있다.

언어 선택

철학이란 언어를 사용해 우리의 지성이라는 마력에 대항하는 싸움이다.
– 루트비히 비트겐슈타인(Ludwig Wittgenstein)

컴퓨터 언어는 바보 같은 동작을 방지하는 방향으로 점차 변화하고 있다. 최첨단 언어에 의존하는 것은 남자답지 못한 행동이다. 우리가 사용할 수 있는 가장 오래된 언어 사용을 고집하자. 가능하다면 8진법 기계어(필자는 Hans와 Frans처럼 계집애 같은 남자가 아니다. 나는 남성미 넘치는 사람으로 IBM의 레코드 장비(펀치 카드)의 플러그보드에 직접 케이블을 꽂아가며 코딩을 하거나 종이 테이프에 펀치를 이용해 구멍을 뚫어 코딩을 했던 사람이다)를, 안 된다면 어셈블러, 안 되면 포트란이나 코볼이라도, 안 되면 C, 안되면 베이직, 안되면 C++ 을로 시도하자.

포트란(FORTRAN)

포트란으로 코드를 작성하라. 상사가 그 이유를 묻는다면 포트란에는 유용한 라이브러리가 많아서 시간을 절약할 수 있다고 대답하자. 포트란으로 작성한 코드를 유지보수 할 수 있는 확률은 0이다. 따라서 유지보수 할 수 없는 코드 가이드라인을 지키기가 아주 쉬워진다.

Ada를 피하라

여기서 설명한 기법 중 약 20%는 Ada에 사용할 수 없다. Ada 사용은 적극 피해야 한다. 관리자가 우리를 압박하면 다른 사람 모두 Ada를 사용할 수 없다고 주장하고, lint나 plummer 같은 우리의 커다란 도구 스위트와 맞지 않는다는 점을 부각하자.

ASM 사용

모든 공통 유틸리티 함수를 asm으로 변환하자.

QBASIC 사용

모든 중요 라이브러리 함수는 QBASIC으로 남겨둔 다음, 큰 메모리 –〉 중간 메모리 모델로 매핑을 처리하는 asm 래퍼를 만들자.

인라인 어셈블러

인라인 어셈블러를 우리 코드 여기저기에 흩어놓는 것도 재미있다. 어셈블러를 이해하는 사람은 이제 거의 없다. 몇 라인의 어셈블러 코드로도 유지보수 프로그래머를 얼려버릴 수 있다.

C를 호출하는 MASM

C에서 호출하는 어셈블러 모듈이 있다면 특별한 이유가 없더라도 가능한 한 자주 어셈블러에서 C로 다시 호출하게 하자. 어셈블러만의 코드 난독화의 매력을 줄 수 있는 goto, bcc 등을 충분히 활용하는 것도 잊지 말자.

유지보수 도구를 피하라

풍부한 코딩을 피하고 다른 언어에 조잡하게 만들어진 인터페이스를 사용하지 말아야 한다. 이들은 철저히 유지보수 프로그래머의 직업을 쉽게 도와줄 목적으로 설계되었다. 마찬가지로 제품이 생산되기도 전에 버그를 잡을 수 있게 설계한 Eiffel나 Ada도 피해야 할 대상이다.

발상의 전환

지옥은 다른 사람이다.
– 진 폴 사르트르(Jean-Paul Sartre), 출구가 없다(No Exit), 1934년

지금까지 어떻게 유지보수 프로그래머를 좌절시키고 당황시킬 수 있는지에 대한 팁을 살펴봤다. 유지보수 할 수 없는 코드를 작성하는 일을 방해하려는 직장 상사의 노력을 어떻게 저지하는지 또는 저장소의 코드가 어떤 형식를 유지할 것인지에 대한 선동을 할 수 있는지 살펴봤다.

직장 상사는 무엇이 좋은 것인지 안다

직장 상사가 자신의 20년 포트란 경력이 현재 프로그래밍에 훌륭한 도움이 될 수 있다고 생각한다면 그대로 그 사람의 제안을 받아들이자. 결과적으로 상사는 우리를 신뢰할 것이고 이는 우리의 경력에도 도움이 된다. 결과적으로 프로그램 코드를 난독화하는 새로운 기술을 얻을 수 있을 것이다.

헬프 데스크 조직을 와해시키라

코드에 버그가 창궐할 수 있게 하는 확실한 방법 중 하나는 유지보수 프로그래머에게 이 소식이 전해지지 않게 하는 것이다. 그러려면 헬프 데스크를 와해시켜야 한다. “전화 주셔서 감사합니다. 직원 연결은 “1”번을 음성 메일은 삐 소리 후 남겨주세요”라는 멘트가 나오는 자동응답기를 이용하고 직접 전화에 응답하지 말자. 이메일로 도움을 요청한 경우에도 문제 추적 번호를 할당하기 전에 무시되게 해야 한다. “당신의 계정에 문제가 있습니다. 현재 담당자가 없어 처리가 곤란합니다”와 같은 답변은 모든 문제에 거의 공통적으로 적용할 수 있는 문구다.

그 입 닫으라

Y2K와 같이 걱정스런 문제를 바짝 경계할 필요는 없다. 심각한 일이 벌어질 날이 다가오는 걸 알았더라도 실제 문제가 발생할 때까지는 공개적 토론을 삼가야 한다. 친구나, 직장동료, 우리의 발견을 가로챌 경쟁자에게 절대 말하지 말자. 어떤 경우에라도 이 사건에 힌트를 제공하거나 사건을 기회로 이용하지 말아야 한다. 전문 용어로 암호화한 메모를 위 관리자에게 전달해 둠으로써 면책할 구멍을 마련할 수 있다. 가능하면 추신으로 본문과 전혀 관련이 없는 사업 현황을 걱정하는 평문 메모를 덧붙이는 것도 좋은 방법이다. 이제 우리가 해야 할 일은 모두 했으므로 밤에는 두 다리 뻗고 잘 수 있을 것이다. 그리고 어느날 우리가 퇴직한 이후에 어마어마한 급여를 제시하며 우리를 애타게 찾을 그날을 기다리자.

헛소리로 당황시켜라

때로는 대형 망치가 다른 도구보다 미묘할 수 있는 것처럼, 미묘함이란 생각하면 생각할수록 놀라운 것이다. FooFactory라는 같은 클래스를 만들고 실제 이 클래스는 오브젝트 생성과는 전혀 관련 없는 클래스지만 주석에 GoF의 생성 관련 패턴(가짜 UML 디자인 문서를 가리키는 http 링크 등을 사용하면 효과적이다)을 추가하자. 이렇게 함으로 유지보수 프로그래머가 망상에 빠지게 할 수 있다. 좀 더 미묘하게 싱글톤을 반환하는 것이 아니라 새로운 인스턴스를 반환하는 protect 생성자와 메소드 Foo f = Foo.newInstance()를 만들 수도 있다. 이로 발생할 수 있는 부작용은 상상을 초월한다.

월간 서적 클럽

월간 서적 클럽에 가입하라. 저자 중에 책을 쓰느라 너무 바쁜 나머지 코드를 직접 작성하지 못하는 저자를 찾아보자. 다이어그램만 잔뜩 있고 예제 코드가 없는 책이 있는지 동네 서점에서 찾아보자. 이런 책을 훑어보면서 잘 알려지지 않은 유식한 단어를 알아둔 다음 건방진 애송이가 들어오면 이들 단어로 한 방 먹여줄 수 있다. 물론 우리의 코드로도 인상을 줄 수 있어

야 한다. 우리의 어휘를 이해하지 못하는 사람은 아마도 그것은 우리가 너무 똘똘하고 알고리즘이 심오하기 때문이라고 생각할 것이다. 알고리즘을 쉽게 유추할 수 있게 설명하는 일은 피해야 한다.

직접 만들어라

우리는 늘 시스템 수준 코드를 직접 만들어보길 원한다. 지금이 바로 시스템 수준 코드를 작성해 볼 기회다. 표준 라이브러리는 무시하고 자신만의 라이브러리를 만들어보자. 이것이 우리의 이력서를 빛낼 것이다.

자신만의 BNF 만들기

항상 자신만의 명령어 문법을 독창적이고 알려지지 않은 BNF 표기법으로 문서화하자. 학습 열의를 떨어뜨릴 수 있으므로, 유효한 커맨드와 유효하지 않은 커맨드와 같은 샘플을 추가하면서까지 문법을 설명하는 것을 피해야 한다. 대충 그린 철로-다이어그램 정도면 충분하다. 각각의 심볼(문법에서 한 구문을 가리키는)이 명확히 무엇을 나타내는지 알 수 없게 해야 한다. 활자체, 컬러, 대문자 등과 같이 두 사물을 구분하는데 도움이 되는 어떤 시각효과도 사용하지 않는 것이 바람직하다. 우리의 명령어 자체의 BNF 표기에서는 항상 같은 구두 문자를 사용해야 한다. (...), [...], {...} , “...” 와 같은 경우 독자는 구두점이 명령어의 일부인지, 꼭 있어야 하는 것인지, 옵션 사항인지 구별하기가 힘들어진다. 결국 우리가 만든 BNF를 이해하지 못할 정도로 멍청한 자들은 우리 프로그램을 사용할 자격을 상실하는 것이다.

자신만의 할당자를 만들어라

동적 저장소 디버깅이 복잡하고 시간이 많이 걸린다는 것은 모두가 아는 사실이다. 각 클래스에서 저장소를 낭비하지 않는다는 사실을 일일이 확인하는 것보다는 큰 저장소의 공간을 할당해주는 자신만의 저장소 할당자를 만드는 것이 바람직하다. 저장소를 직접 해제하는 것보다는 주기적으로 시스템을 리셋해서 힙(heap)을 정리하게 만들어야 한다. 리셋을 이용하면 시스템에서 처리해야 할 일(모든 저장소 누수를 확인하지 않아도 되므로)이 크게 줄어든다. 따라서 사용자가 시스템을 주기적으로 리셋하는 것을 까먹지 않는 한 힙 메모리 공간을 모두 사용하는 일이 없다. 프로그램을 배포한 다음에 이와 같은 동작을 바꾸려고 낑낑대는 이들의 모습이 떠오르지 않는가?

색다른 언어를 이용한 트릭

베이직 프로그래밍은 뇌를 손상시킨다.
– 에스거 비버 딕스트라(Edsger Wybe Dijkstra)

SQL 별칭

테이블 이름 별칭을 하나 혹은 두 개 문자로 정하자. 물론 관련이 없는 기존 테이블 이름을 별칭으로 사용하는 것도 바람직하다.

SQL 외부 조인

다양한 외부 조인 문법을 사용하므로 모두를 긴장시키자.

자바스크립트 범위

자바스크립트 코드의 함수는 호출자의 범위에 있는 모든 지역 변수에 접속할 수 있다는 사실을 "최대한 활용" 하자.

비주얼 베이직 선언

다음과 같은 선언보다는

```
dim Count_num as string
dim Color_var as string
dim counter as integer
```

아래와 같이 선언하는 것이 바람직하다

```
Dim Count_num$, Color_var$, counter%
```

사람을 미치게 하는 비주얼 베이직

텍스트 파일에서 데이터를 읽을 때 실제 필요한 데이터 보다 15개 문자를 더 읽은 다음 실제 텍스트는 아래와 같이 삽입할 수 있다.

```
ReadChars = .ReadChars (29,0)
ReadChar = trim(left(mid(ReadChar,len(ReadChar)-15,len(ReadChar)-5),7))
If ReadChars = "alongsentancewithoutanyspaces"
Mid,14,24 = "withoutanys"
and left,5 = "without"
```

델파이/파스칼 전용 기법

함수나 프로시저를 사용하지 말자. 대신 label/goto 문으로 코드 여기저기를 점프할 수 있다. 이 기법은 유지보수 프로그래

머를 정신을 빼 놓을 수 있다. 처음에는 이와 같은 기법에 적응할 시간을 좀 주고 갑작스럽게 마구잡이로 점프를 시작하는 것도 좋은 방법이다.

펄

If문과 unless문을 계속 이어 나간다. 특히 아주 긴 줄일 경우에 효과적이다.

Lisp

LISP는 유지보수 할 수 없는 코드를 작성하는 우리에게는 환상적인 언어다. 아래와 같은 환상적인 코드를 감상해보라!

```
(lambda (*<8-]= *<8-[= ) (or *<8-]= *<8-[= ))

(defun :-] (<) (= < 2))

(defun !(!)(if(and(funcall(lambda(!)(if(and '(< 0)(< ! 2))1 nil))(1+ !))
(not(null '(lambda(!)(if(< 1 !)t nil)))))1(* !(!(1- !)))))
```

비주얼 폭스프로(Foxpro)

이 팁은 비주얼 폭스프로에서만 적용할 수 있는 내용이다. 변수의 상태가 undefined인 경우에는 값을 할당해야 변수를 사용할 수 있다. 이와 같은 상황은 변수의 형식을 확인할 때 발생한다.

```
lcx = TYPE('somevariable')
```

위 코드의 수행 결과 lcx는 'U' 또는 undefined가 된다. 그러나 변수에 범위를 할당하면 변수를 정의하는 효과가 나타나므로 논리적인 FALSE값을 갖게 할 수 있다. 간단하지 않은가?

```
LOCAL lcx
lcx = TYPE('somevariable')
```

그럼 lcx 값은 'L' 혹은 논리값(결국 FALSE)을 갖는다. 유지보수 할 수 없는 코드를 작성하는데 이러한 특징이 얼마나 도움이 될지 상상해보라.

```
LOCAL lc_one, lc_two, lc_three... , lc_n

IF lc_one
DO some_incredibly_complex_operation_that_will_neverbe_executed WITH
make_sure_to_pass_parameters
ENDIF

IF lc_two
DO some_incredibly_complex_operation_that_will_neverbe_executed WITH
make_sure_to_pass_parameters
ENDIF

PROCEDURE some_incredibly_complex_oper....
```

```
* 여기에 추가하는 많은 코드는 절대 실행되지 않는다.
* 우리의 메인 프로시저 코드를 여기에 잘라내서 붙인다면 재미있지 않겠는가!
ENDIF
```

잡다한 기법

누군가에게 프로그램을 주면 하루 만에 그를 좌절시킬 수 있다. 그러나 프로그램을 어떻게 사용하는지를 알려줌으로써 그를 평생 좌절시킬 수 있다.
– 아무개씨

재컴파일 하지 마라

가장 사악한 기법부터 살펴보자. 컴파일에 성공해서 실행파일을 만들었다면, 각 모듈에서 소스코드 몇 개를 수정하자. 그러나 이들 파일을 애써 재컴파일할 필요까진 없다. 나중에 디버깅 할 시간이 남았을 때 재컴파일 하면 된다. 운이 지지리 없는 유지보수 프로그래머가 몇 년 후에 코드를 수정하지만 프로그램이 제대로 동작할 리 없다. 아마도 그녀는 자신이 수정한 뭔가에 문제가 있다고 생각할 것이다. 이와 같은 기법으로 우리는 그녀를 몇 주 동안 바쁘게 만들 수 있다.

디버거 차단

줄을 길게 만들어서 디버거를 이용해서 한 줄씩 우리 코드를 이해하려는 사람을 좌절시킬 수 있다. 특히 브레이크 포인트를 잡기 어렵게 if와 then을 한 줄에 모두 사용하면 더욱 효과적이다. 그러면 분기문이 어느 문장을 수행하는 것인지 구별하기가 어려워진다.

S.I. vs 미국식 단위

엔지니어링 작업을 할 때 두 가지 방법으로 코드를 만들 수 있다. 하나는 모든 입력을 S.I.(미터법) 단위를 사용하고 나중에 결과를 돌려줄 때에는 사람들이 이용하는 단위로 변환할 수 있다. 다른 방법은 여러 단위를 다양하게 시스템 전체에서 혼합해 사용하는 것이다. 물론 항상 두 번째 방법을 이용하는 것이 좋다. 그게 미국식이다!

CANI

상수 그리고 절대 끝나지 않는 개선작업(Constant And Never-ending Improvement). 우리가 코드를 "개선" 하면 사용자도 업그레이드해야 한다. 결국 오래된 버전을 원하는 사람은 아무도 없기 때문이다. 프로그램 자체로도 그렇게 좋아한다면, 문제를 "수정" 한 버전을 내놓았을 때에는 얼마나 기뻐할지 생각해보라! 누군가가 묻지 않는 한 절대 버전 변경의 차이를 말하지 마라. 알려주지 않았으면 알아차리지도 못했을 예전 버그를 왜 알려줘야 하는가?

프로그램 정보

프로그램 정보란에는 프로그램 이름, 코더 이름, 난해한 법률용어를 포함하는 저작권 등을 포함한다. 이 화면에서 재미있는 애니메이션을 보여주는 코드를 실행하는 링크가 있다면 금상첨화다. 그러나 프로그램 정보란에 프로그램의 목적, 마이너 버전, 최신 코드 리비전 날짜, 업데이트를 얻을 수 있는 웹사이트, 저자의 이메일 주소 등은 절대 명시하지 말아야 한다. 이렇게 시간이 흐르면 많은 이들이 각자 서로 다른 버전을 사용하게 할 수 있다. 때로는 N+1 버전을 설치하지도 않고 N+2 버전을 설치하려 하는 이도 있을 것이다.

변화, 변화, 변화하라

버전마다 더 많은 변화를 줄수록 좋다. 사용자가 예전 API나 오래된 사용자 인터페이스를 사용하면서 지루해 하는 것을 원하는 사람은 아무도 없다. 그러나 사용자가 알지 못하는 변경을 하는 것도 나쁘진 않다. 결국 이 모든 변화는 사용자를 긴장

시키고, 현실에 안주하지 않게 채찍질 할 것이다.

개별 파일에 C 프로토타입을 추가하라

공통 헤더를 사용하지 않고 개별 파일에 C 프로토타입을 추가하므로 파라미터 데이터 형식을 모든 파일에서 유지하게 하고 컴파일러나 링크가 형식 불일치를 검출하지 못하게 하는 일석 이조 효과를 얻을 수 있다. 이는 32비트에서 64비트 플랫폼으로 포팅할 때 특히 유용하다.

특별한 기술은 필요없다

큰 기술이 있어야 유지보수 할 수 없는 코드를 작성할 수 있는 것은 아니다. 그냥 잡히는 대로 코딩을 시작해보자. 대부분의 관리자는 우리가 만든 코드 줄 수로 생산성을 판단하는 경향이 있다. 따라서 나중에 삭제할 쓸모 없는 코드라도 일단 넣어보자.

한 개의 망치만 사용한다

자신이 잘 아는 도구를 사용해 가볍게 여행하자. 망치 하나만 있으면 모든 문제는 못에 불과하다.

표준을 무시하라

가능하면 프로젝트를 진행하는 언어와 환경에서 수 천명이 사용하는 코딩 표준을 무시해야 한다. 예를 들어 MFC 기반 응용 프로그램을 만들면서 STL 스타일의 코딩 표준을 따르겠다고 주장하자.

일반적인 True, False 규칙을 뒤집어라

일반적인 true, false에 대한 정의를 뒤집어라. 보기보다는 파급효과가 크다. 아래와 같은 정의를,

```
#define TRUE 0
#define FALSE 1
```

아무도 잘 찾아보지 않을 코드 깊은 곳에 숨겨야 한다. 그리고 프로그램에서는 아래와 같은 비교문을 사용할 수 있다.

```
if ( var == TRUE )
if ( var != FALSE )
```

누군가 위와 같은 중복 문제를 “수정” 해서, 다음과 같이 정상적인 방법으로 사용할 가능성도 있다.

```
if ( var )
```

완전한 사기처럼 보일 수 있겠지만 TRUE와 FALSE가 같은 값을 갖도록 하는 기법도 있다. 1과 2 또는 −1과 0과 같이 교묘하게 변경하는 방법도 바람직하다. 자바에서도 TRUE라는 이름의 정적 상수를 정의함으로써 이 기법을 사용할 수 있다. 자바에는 true라는 내장어가 이미 있기 때문에 우리의 의도를 의심하는 프로그래머가 등장할 수 있다.

서드 파티 라이브러리

프로젝트에 막강한 서드 파티 라이브러리를 포함하고는 사용하지 않는다. 추가하고 사용하진 않았지만, 우리의 이력서 “기타 도구” 부분에 사용하지 않았던 도구 이름을 추가할 수 있다.

라이브러리를 피하라

개발 도구에 포함된 라이브러리를 모른척해야 한다. 비주얼 C++을 사용한다면 MFC나 STL의 존재를 무시하고 문자열이나 배열을 손수 작정할 수 있다. 이렇게 하면서 자신도 모르게 포인터 기술이 좋아지고 동시에 코드를 확장하려는 시도를 좌절시킬 수 있다.

빌드 순서를 만들라

빌드 순서를 정교하게 만들어서 유지보수 프로그래머가 자신이 수정한 파일을 컴파일 하지 못하게 할 수 있다. 숨겨진 SmartJ를 이용해서 make 스크립트를 무용지물로 만들자. 비슷한 방식으로 컴파일러를 클래스로 사용할 수 있다는 사실도 비밀로 간직해야 한다. 죽는 한이 있더라도 파일을 찾고, 직접 컴파일 클래스 sun.tools.javac.Main를 호출하는 간단한 자바 프로그램을 만드는 일이 얼마나 쉬운지를 절대 발설하지 말자.

Make를 이용한 장난질

여러 디렉터리에서 소스를 복사하는 배치 파일을 Make파일로 생성한 다음 어떤 규칙으로 파일을 복사하는지는 문서화하지 않는다. 이 기술을 사용하면 멋진 소스 코드 관리 시스템 없이도 코드 가지치기(branching)를 할 수 있으며 후임자가 어떤 버전의 DoUsefulWork()을 수정해야 하는지 절대 찾을 수 없게 만들 수 있다.

코딩 표준을 수집하라

정사각 박스 제안(Square Box Suggestions)과 같은 유지보수 할 수 있는 코드를 작성하는 방법에 대한 팁을 모두 모아서 대놓고 그 팁을 위반하자.

내가 아니라, IDE!

모든 코드를 makefile로 만들어라. 후임자는 헤더파일을 생성하고 응용 프로그램을 빌드하는 배치파일을 생성하는 makefile을 만들었다는 사실에 감탄할 것이다. 그리고 이를 변경했을 때 어떤 일이 일어날 것인지 알기 어렵고 최신 IDE로 프로젝트를 옮기기도 어렵게 만들 수 있다. 그리고 이미 뇌사 상태에 들어간 NMAKE 버전을 종속성 개념 없이 사용하므로 효과를 극대화 할 수 있다.

회사 코딩 표준을 무시하라

몇몇 회사에서는 프로그램에 숫자 기호 사용을 금지하는 등과 같은 강력한 정책을 적용한다. 이 경우에는 어쩔 수 없이 상수에 이름을 붙여 사용해야 한다. 물론 정책을 와해시키는 것은 간단하다. 예를 들어, 똘똘한 C++ 프로그래머는 다음과 같이 상수를 정의했다.

```
#define K_ONE 1
#define K_TWO 2
#define K_THOUSAND 999
```

컴파일러 경고

컴파일러 경고를 모두 수정하지 말고 남겨두는 것이 좋다. Make 파일에 접두어 "-"를 사용하므로 컴파일 에러로 make가 실패하는걸 방지할 수 있다. 실수로 유지보수 프로그래머가 소스 코드에 에러를 저질렀더라도 make는 전체 패키지를 다시 빌드하려 할 것이다. 심지어 운이 좋으면 빌드에 성공한다! 우리 코드를 손수 컴파일 해본 프로그래머라면 컴파일에 실패할 것이고, 기존 코드나 헤더에 자신이 무엇을 잘못 추가했는지 확인하려 할 것이다. 하지만 결국 유지보수 프로그래머는 모

든 버그가 원래부터 있었던 것임을 알고는 이를 찾으려 애써야 했던 즐거웠던 경험에 대해 우리에게 감사해 할 것이다. 컴파일러의 몇몇 에러 확인 진단 옵션을 사용하면 우리 프로그램이 컴파일되지 않을 수 있다. 컴파일러는 경계 값 확인 등의 기능을 수행할 수 있지만, 실제 업무에서 이 기능을 사용하는 프로그래머는 거의 없으므로 우리도 이런 기능을 사용할 이유가 없다. 이런 미묘한 버그를 찾는 기쁨과 즐거움을 컴파일러에게 빼앗긴다니 말이 되는가?

버그 수정과 업그레이드를 혼합하라

절대 "버그만 수정한" 버전을 릴리즈 하지 말아라. 버그를 수정했으면 데이터베이스 형식도 바꾸고, 복잡한 사용자 인터페이스도 변경하고, 관리자 인터페이스도 다시 만들자. 이렇게 하면 사람들은 그냥 버그에 익숙해지려 하고 결국 버그를 기능이라 부르기 시작할 것이다. 단지 이런 "기능"이 다른 방식으로 동작하길 바라는 사용자만이 새로운 버전으로 업그레이드 할 필요성을 느낄 것이다. 이런 방식으로 유지보수 작업을 줄일 수 있고, 고객으로부터 얻는 수익도 늘어난다.

제품을 릴리즈 할 때마다 파일 형식을 변경하라

미래 호환성을 원하는 고객이 많으므로 원하는 대로 해주자. 그러나 기존 버전과의 호환성은 유지하지 않는 것이 핵심이다. 그러면 민감한 버그 수정 정책(윗글 참조)과 함께 나온 새로운 버전이 나왔을 때, 고객이 한번 새로운 버전을 사용하기 시작했다면 다시 이전버전으로 돌아갈 수 없게 된다. 결국 고객은 새로운 버전이 나와도 업그레이드를 선택하지 않을 것이다. 추가 보너스 팁! 새로운 버전에서 생성한 파일을 예전 버전에서 인식조차 못하게 하는 방법은 무엇이 있을까? 이런 방법을 이용해 심지어는 같은 응용 프로그램으로 만든 파일이 아니라고 거부할 수 있다. PC 워드 프로세서에서 이러한 복잡한 동작과 관련한 예제를 제공한다.

버그를 보상하라

코드에서 버그의 근본 원인을 찾아내는 것을 두려워하지 말고 고수준 루틴에 이를 보상할 수 있는 코드를 넣자. 이는 3D 체스와 같은 지능 활동의 산물이다. 이 덕분에 이후에 작업할 유지보수 프로그래머는 문제가 데이터를 생성하는 저수준 루틴에서 발생한 것인지 아니면 값을 변경하는 고수준 루틴에서 발생한 것인지를 찾는 즐거움에 빠져 수많은 시간을 보내야 할 것이다. 이 기법은 멀티 패스 프로그램인 컴파일러에 적합하다. 첫 번째 과정에서는 문제 수정을 회피함으로써 나중 과정을 더욱 복잡하게 만들 수 있다. 운이 좋으면 컴파일러의 프론트엔드 유지보수를 담당자에게 이 부분에 대해 얘기할 필요가 없는 경우도 있다. 프론트엔드에서 데이터를 정확하게 만든 경우 백엔드가 멈추게 하면 더욱 좋다.

스핀 락(Spin Lock)을 활용하라

스핀 락을 사용하고 기본 동기화 기능을 사용하지 말아라. 반복적으로 슬립 상태에 빠지면서 전역 변수(비휘발성의)를 통해 조건을 만족하는지 확인하자. 스핀 락은 시스템 오브젝트보다 더 "일반적"이고 "유연"하며 사용하기 쉽다.

동기화 코드를 마구 뿌려대라

꼭 필요하지 않은 곳이라 하더라도 시스템 동기화 코드를 추가해보자. 필자는 코드에서 두 번째 스레드가 실행할 가능성이 전혀 없는 크리티컬 섹션을 우연히 발견했다. 기존 개발자를 한번 시험해 봤고, 그는 해당 코드가 "비난받기(critical)!" 마땅하다고 시인했다.

우아한 타락

시스템에 NT 디바이스 드라이버를 사용한다면, 응용 프로그램에서 I/O에 필요한 버퍼를 요구하고 트랜잭션이 일어나는 동안 버퍼에 락을 걸도록 요청하자. 그리고 나중에 버퍼를 언락하고 해제한다. 응용 프로그램이 버퍼에 락을 건 상태로 비정상적으로 종료하면 NT 자체의 크래쉬를 발생시킬 수 있다. 클라이언트 사이트에서는 디바이스 드라이버를 바꿀 방법이 없으므로 선택의 여지는 없다.

커스텀 스크립트 언어

우리의 클라이언트/서버 응용 프로그램에서 실행 중에 바이트로 컴파일되는 스크립팅 명령어 언어를 포함해야 한다.

컴파일러 종속 코드

컴파일러나 인터프리터 버그를 발견했으면 이 버그를 이용해 우리 코드가 제대로 동작하게 만들자. 이제 우리 프로그램을 사용하는 모든 이는 다른 컴파일러를 사용할 수 없게 된다.

실생활 예제

스승님께서 작성하신 실생활 예제를 보여주겠다. 하나의 C 함수에 그가 사용한 여러 기법을 살펴보자.

```
void* Realocate(void*buf, int os, int ns)
{
    void*temp;
    temp = malloc(os);
    memcpy((void*)temp, (void*)buf, os);
    free(buf);
    buf = malloc(ns);
    memset(buf, 0, ns);
    memcpy((void*)buf, (void*)temp, ns);
    return buf;
}
```

1. 자세히 보면 Realocate의 철자가 부정확하다. 창의적인 철자법의 힘을 얕보지 말라.
2. 아무런 이유 없이 입력 버퍼를 임시 복사본을 만든다.
3. 이유 없이 형을 변환한다. Memcpy()는 (void*)형을 매개변수로 받으므로 이미 (void*)형을 사용하더라도 무조건 형변환한다.
4. temp를 해제할 필요는 없다. 이렇게 작은 메모리 누수의 효과는 미미하기 때문에 며칠 동안 프로그램을 돌려도 끄떡없는 경우가 많다.
5. 만약을 대비해서 버퍼에서 필요한 데이터 이상을 복사하라. 유닉스에서는 코어 덤프가 발생할 것이다.
6. 위 코드에서 os는 “old size”를 ns는 “new size”를 의미하는 듯 하다.
7. 표준 라이브러리에 이미 있는 간단한 함수는 다시 만든다.
8. buf를 할당하고 0으로 memset한다. 누군가 ANSI 스펙을 변경할 가능성이 있으므로 calloc()을 사용하는 것은 위험하다(일단 어차피 buf 크기와 동일한 데이터를 복사한다는 사실은 신경쓰지 말자).

사용하지 않은 변수 에러를 고치는 방법

컴파일러에서 “사용하지 않은 지역 변수” 경고를 발생한다고 해서 그 변수를 꼭 없앨 필요는 없다. 대신, 창의적으로 고민을 좀 해보자. 나같으면…

```
i = i;
```

중요한 것은 크기

함수가 크면 클수록 좋다는 것은 두말하면 잔소리인 진리다. 물론 jump와 GOTO도 많을수록 좋다. 이렇게 할 때 누군가 무

엇을 변경하게 되면, 검증해야 할 시나리오가 정말 다양해 진다. 이제 코드 스스로가 유지보수 프로그래머에게서 자신을 보호할 수 있게 된다. 함수를 거인 왕처럼 만들 수 있다면 함수는 고질라가 되어 유지보수 프로그래머가 무슨 일인지 정신을 차리기도 전에 공격하고 무자비하게 밟아버릴 것이다.

하나의 그림은 1000개의 단어를 대신하고, 하나의 함수는 1000개의 줄을 대신한다.

모든 메소드의 바디를 가능한 한 길게 만들어라. 깊게 중첩하는 것을 잊지 말고 절대 앞으로는 1000줄 이하의 메소드나 함수를 만드는 일이 없도록 하라.

사라진 한 개의 파일

중요한 파일이 적어도 한 개 이상이 없어지게 해야 한다. 이 기법은 include에 include를 혼합하므로 효과를 극대화할 수 있다. 예를 들어, 우리 모듈에서

```
#include <stdcode.h>
```

Stdcode.h는 존재한다. 그러나 stdcode.h에서 가리키는

```
#include "a:\\refcode.h"
```

Refcode.h 파일은 어디에서도 찾을 수 없어야 한다.

여러 곳에서 작성하고 어디에서도 읽을 수 없게 하라

적어도 한 변수를 모든 곳에서 설정하도록 하고 사용하는 곳이 없게 하라. 불행히도 현대 컴파일러는 아무데서도 기록하지 않고 모든 곳에서 읽는 동작을 허용하지 않는다. 하지만 C나 C++에서는 이러한 동작을 아직 수행할 수 있다.

철학

컴파일러나 시스템 클래스를 만드는 사람이 보통 언어를 설계한다. 당연한 일이지만, 그들은 자신의 작업을 용이하게 수학 계산을 간단히 할 수 있는 방식으로 언어를 설계한다. 그러나 한 컴파일러를 구현하는 사람 뒤에는 10,000 명의 유지보수 프로그래머가 필요하다. 정작 힘들게 일하는 유지보수 프로그래머는 언어 설계 단계에서 의견을 내지 못한다. 그러나 유지보수 프로그래머가 작성한 코드 양에 비하면 컴파일러 코드는 한없이 작을 뿐이다.

이런 엘리트 주의적인 생각은 JDBC 인터페이스에서 잘 나타난다. JDBC 인터페이스는 JDBC 구현자의 삶을 쉽게 만들어 준다. 그러나 유지 보수 프로그래머에게는 악몽을 가져다 주었다. JDBC 인터페이스는 삼십년 전에 나온 포트란의 SQL 인터페이스보다 못하다.

유지보수 프로그래머는 세부사항은 감추어서 나무로 이루어진 숲을 볼 수 있게 해달라고 요청한다. 그들은 다양한 종류의 단축(shortcut) 방법을 이용해서 많은 코드를 타이핑하지 않아도 되고, 따라서 프로그램을 화면으로 확인하기 쉽게 해달라고 요청한다. 유지보수 프로그래머는 컴파일러가 요구하는 많은 하찮은 작업으로부터 해방시켜 달라고 큰소리로 불평한다.

이와 같은 노력에 부흥하는 결과로 현재 작업과 관계없는 세부 사항은 감출 수 있는 NetRexx, Bali 그리고 비주얼 편집기(예를 들어 IBM의 비주얼 에이지를 시작으로) 등이 소개됐다.

구두 제조인은 신발이 없다

자신의 원장을 워드 프로세서로 관리해 달라고 주장하는 회계사 고객이 있다고 가정하자. 아마 우리는 데이터를 구조화 한다고 그를 설득할 것이다. 그는 상호간의 필드 검사와 유효성 검사도 요구한다. 우리는 동시 업데이트를 지원하는 데이터베이스에 데이터를 저장하는 것이 유리하다고 그를 설득할 것이다.

소프트웨어 개발자 고객의 경우를 살펴보자. 그는 모든 데이터(소스 코드를) 텍스트 편집기에 보존해야 한다고 주장한다. 그는 아직 워드 프로세서의 색상, 형식 크기, 폰트 등을 사용해본 적이 없다.

소스 코드를 구조화된 데이터로 저장하면 무슨 일이 일어날까? 하나의 소스코드를 자바, NextRex, 의사결정 테이블, 플로우 차트, 루프 구조 뼈대(상세 내용 없이), 상세 내용이나 주석을 제거한 자바, 현재 관심이 있는 변수와 메소드 호출을 하이라이트한 자바 코드, 매개변수 이름과 형식에 대한 주석을 포함하는 자바 코드 등의 다양한 방식으로 볼 수 있다. 복잡한 수식 표현을 TeX나 수학자들이 하듯이 2차원으로 확인할 수 있다. 괄호를 추가하거나 빼(자신의 우선순위 규칙 취향에 따라) 형태의 코드도 확인할 수 있다. 괄호는 다양한 크기와 색을 눈으로 구별하는데 유용하다. 선택적으로 제거하거나 적용할 수 있는 투명한 오버레이 집합을 사용해서 다른 나라에서 일하는 팀의 다른 프로그래머를 실시간으로 볼 수 있고 작업중인 코드 변경 사항도 확인할 수 있다.

색상을 이용해 미묘한 단서의 실마리로 사용할 수 있다. 예를 들어 각 패키지/클래스에 색상을 자동으로 지정해서 클래스의 메소드나 변수의 레퍼런스 백그라운드에 파스텔 그림자를 사용할 수 있다. 특정 식별자의 정의에 볼드체를 적용해서 두드러지게 할 수도 있다.

X 형식의 오브젝트를 생성하려면 어떤 메소드나 생성자를 사용해야 하는가? 오브젝트 X 형식을 파라미터로 받는 메소드는 어디 있는가? 코드에서 어떤 변수에 접근할 수 있는가?와 같은 질문을 할 수 있다. 메소드 호출이나 변수 레퍼런스를 클릭하면 관련 정의를 확인할 수 있고, 어떤 버전의 메소드가 실제 실행될 것인지 쉽게 확인할 수 있다. 지정한 메소드나 변수의 모든 레퍼런스를 확인하면서 체크 표시를 할 수도 있다. 가리키고 클릭하는 동작으로도 많은 일을 할 수 있으므로 코드를 직접 작성하는 일이 줄어든다.

위에서 살펴본 사항 중 몇 가지는 실현 불가능할 수도 있다. 어떤 것이 실생활에 꼭 필요한 것인지 알아보려면 부딪쳐 보는 수밖에 없다. 기본 도구를 얻었으면 유지보수 프로그래머의 생활을 개선할 수 있도록 수많은 아이디어를 실험해봐야 한다.

이에 대해서는 SCID 학생 프로젝트에서 더 설명하겠다.

이 기사의 초기 버전은 자바 개발자 저널(볼륨 2 문제 6)에 처음 실렸다. 1997년 11월 콜로라도 정상 회의(Colorado Summit Conference)에서도 이 주제에 대해 얘기한 적이 있다. 그 이후로 이 이야기를 발전시켜 왔다.

이 글은 농담일 뿐이다! 혹시라도 이 글의 내용을 있는 그대로 받아들인 이가 있다면 정중히 사과한다. 캐나다 사람은 농담에 :-)를 삽입하는 것을 세련되지 못한 행동으로 받아들인다. 내가 유지보수할 수 있는 코드를 작성하는 방법에 관해 지껄일 때면 사람들은 별로 관심을 가지지 않았다. 어느 날 일을 그르치는 얼간이 같은 행동을 얘기해야 사람들이 더 반응을 보인다는 사실을 알게 됐다. 유지보수 할 수 없는 디자인 패턴을 확인하므로 더 효과적으로 악의적인 혹은 부지불식간에 일어나는 나쁜 일을 예방할 수 있다.

원본 기사는 로에디 그린의 Mindproducts 사이트에서 확인 할 수 있다.