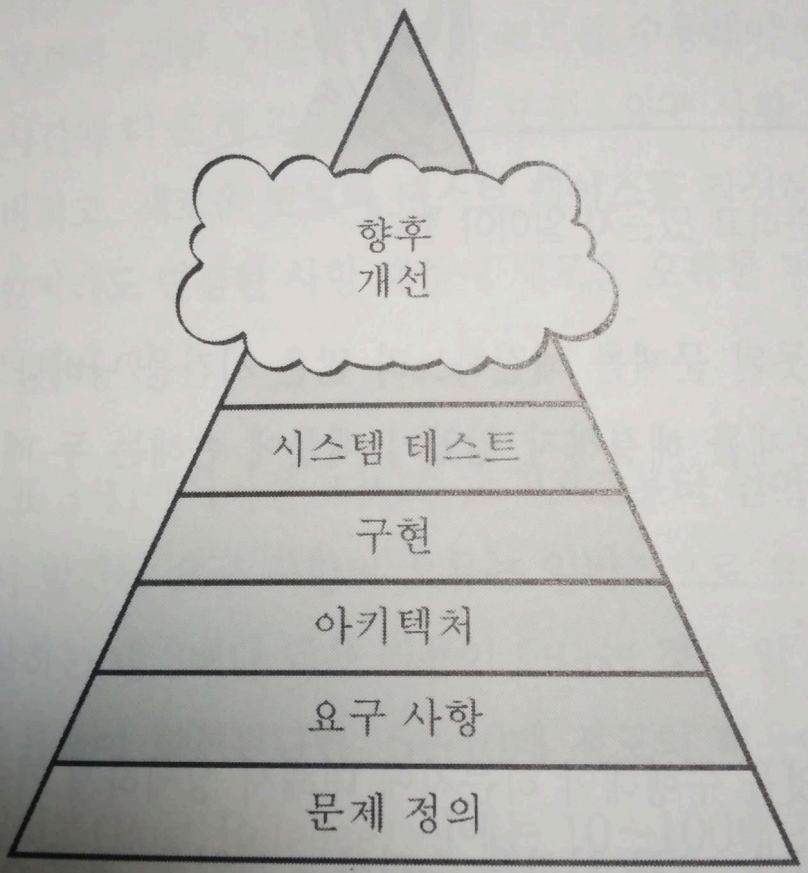
[그림 3-4]에서와 같이 문제 정의는 문제를 깊이 있게 살펴
선행된다.



[그림 3-4] 문제 정의는 전체적인 프로그래밍 프로세스를 우

문제 정의는 반드시 사용자의 언어로 작성되어야 하며,
일반적으로, 기술적인 컴퓨터 용어로 작성되어서는 안

아키텍처는 현재의 시스템 구조에 대한 논리적 근거를 제공하고 각각의 클래스가 신중하게 고려되었음을 보여준다. 설계에 대한 논리적인 근거는 설계 자체만큼이나 유지보수에 중요하다

아키텍처는 고려했던 다른 클래스 설계에 대해서 기술하고, 왜 지금과 같은 구조를 선택하게 되었는지에 대한 이유를 설명해야한다.

아키텍쳐의 할 일에 대해서 보려면 코드 컴플리트 챕터 3을 보자

추상화는 객체를 높은 수준에서 볼 수 있도록하고 캡슐화는 다른 수준에서는 객체를 볼 수 없도록한다

복잡성을 줄이기 위한 상속이어야한다

하나의 자식 클래스만 있는 부모클래스는 상속을 의심

잦은 스위치케이스는 상속을 사용할지 체크

변수 가시 범위를 줄이는 것은 머리 속에 좀 더 적은 내용만 기억해도 되게하는 지적 관리성을 높여준다.

성공적인 프로그래밍은 복잡성을 최소화하는 데 있다.

변수는 하나의 목적으로만 사용하라.

변수 이름의 길이는 8~20 이 디버깅하기가 쉬웠다.

totalRevenue 보다는 revenueTotal 이 일관성있는 변수들을 만들기 쉬우며, 중요한 뜻이 앞에 온다

부동소수점(float, double)을 사용할 때는 항상 정밀도를 생각해야 한다. 0.1을 10번 더할 때 1.0 이 되지 않을 수 있음을 생각하라. IsNearlyZero 같은게 필요하다. 작은 수를 먼저 더한 후에 큰 수를 더하면 그나마 조금 더 낫다.

변경은 오류를 유발하기 쉽기 때문에, 변경이 적다는 것은 그만큼 오류가 적다는 것을 의미한다.

개념적으로 모든 포인터는 두 가지 부분으로 구성된다. 즉, 메모리 상에서의 위치와 해당 위치의 내용을 어떻게 해석하는 지에 대한 내용이다.

몇 백 줄의 코드보다 큰 프로그램을 만들 때, 핵심적인 사항은 복잡성을 관리하는 것이다. 지능적으로 큰 프로그램을 관리할 수 있는 유일한 방법은 한 번에 한 부분만 생각할 수 있도록 코드를 여러 조각으로 나누는 것이다. 만약 전역 데이터를 사용한다면, 한 번에 한 루틴에 집중할 수 있을까? 그렇지 않다. 한 루틴과 동일한 전역 데이터를 사용하는 다른 모든 루틴들도 살펴봐야 한다.

정상적인 경우를 else 가 아니라 if 문 다음에 입력한다.

근접성 원칙에 따르면 연관된 명령문들을 함께 두어야 한다. 만약 연관된 명령문들이 여기저기에 퍼져 있다면, 수정할 때 지나치기 쉽고 정확하게 수정하지 못하기 쉽다.

for 문이 while 문보다 좋은 점은 loop 에 관련된 내용이 한 줄에 다 보일 수 있다는 것이다.

loop 가 하나의 기능만 수행하도록 한다. 만약 loop 하나만으로도 충분한 코드에 두 개의 루프를 사용하는 것이 비효율적인 것처럼 보인다면, 두 개의 루프로 코드를 작성한 후, 효율성을 위해서 결합될 수 있다는 것을 주석으로 작성하여, 두 개의 루프를 하나로 통합하기 전에 벤치마크 결과가 해당 코드 섹션에서 성능 문제가 발생한다는 것을 보고할 때까지 기다리도록 한다.

루프가 종료되는지 마음 속으로 루프를 실행해 본다.

loop 안의 인덱스를 참조해서(recordCount == MAX) 어떤 결과를 도출해내기 보다, 추가 변수(found = true)를 사용해서 명확한 코드를 작성하자. 흔히 있는 일이지만, 추가 불린 변수가 사용되면 작성된 코드가 명확해진다.

무척 중요하며, 무한 루프에 빠지지 않기 위해서 안전한 카운터를 사용한다. if (safetyCounter >= SAFETY_LIMIT) break; 안전한 카운터는 복잡도를 증가시키고 추가적인 오류를 만들 수 있다.

루프 내에 수많은 break 문이 산재되는 것을 주의한다. (진욱:여튼 복잡해 보이는 코드는 뭔가 문제를 일으킴 ㅋㅋ)

break 를 사용하면 루프를 블랙박스로 취급할 수 없게 된다. 루프의 종료 조건을 제어하기 위해서 오직 한 명령문만 사용하도록 제한하는 것이 루프를 단순화시킬 수 있는 강력한 방법이다. break 를 사용하면 여러분이 작성한 코드를 읽는 사람이 루프의 제어 구조를 이해하기 위해서 루프의 내부를 봐야 한다. 결국 루프를 더욱 더 이해하기 어렵게 만든다.

머리에서 돌려보고 손으로 계산해 봄으로써 많은 이득을 얻을 수 있다. 머리에서 돌려보는 습관을 가지면 초기 코드 작성 시 오류가 적게 발생하며 디버깅 시 오류를 더 빨리 발견할 수 있다. 그리고 전체적인 프로그램의 이해도도 좋아진다. 머리에서 돌려본다는 것은, 여러분이 코드가 어떻게 작동하는지를 추측하지 않고 이해하고 있다는 것을 의미한다.

중첩 루프의 인덱스는 의미 있는 이름을 가져라. i,j,k 이런거 안 좋음

재귀 호출이 중단되는지 확인하라. 무한 재귀 호출을 막기 위해서 안전한 카운터를 사용하라. A->B, B->A 와 같은 순환적인 재귀 호출은 발견하기가 어렵기 때문에 위험하다. 한 루틴으로 재귀 호출을 제한하라. 꼭 순환적인 재귀 호출을 해야한다면, 문제점을 예방하는 차원에서 안전한 카운터를 사용해라. 스택의 크기를 고려하고, 이를 위해 힙에 메모리를 할당해라. 팩토리얼을 사용하기 위해서 재귀 호출을 하지마라!!!! ㅋㅋㅋ for 문이 더 이해하기 쉽다. 재귀 호출은 스택과 반복문으로도 할 수 있다. 이 두 가지 모두를 고려해라

꼭 필요한 리소스 해제 루틴이 있다면 try~finally 구문을 고려하자. 물론 언어가 지원해줘야 한다.

설계의 핵심은 여러 가지 중에서 상황에 맞는 것을 선택하는 것이다. (진욱:여러 가지를 알아야 그 중에 선택을 할 수 있다)

코드가 간단해지면 오류가 발생할 확률도 낮아진다.

부정문은 조금 더 읽기가 불편하다. 하지만 부정문을 쓰는게 더 나은 경우가 있으므로 양쪽을 고려해보고 이 상황에서는 어느쪽이 더 나은지 판단하자.

if 문 안에서는 숫자의 크기 순서대로 수치 표현식 작성하기. MIN_ELEMENTS <= i && i <= MAX_ELEMENTS 가 i >= MIN_ELEMENTS && i <= MAX_ELEMENTS 보다 좋다.
시각적으로 의미를 줄 수 있기 때문이다.

0을 비교할 때는, boolean 은 ! 을 사용하고, 숫자는 a != 0 처럼 직접 비교, null terminated char 는 char[i] != '\0' 처럼 명시적으로, 포인터는 p != nullptr 로 해라

3수준을 넘어가는 중첩된 if 문을 이해할 수 있는 사람은 거의 없다.

코드가 복잡하다는 것은 코드를 단순하게 작성할 수 있을 만큼 프로그램을 충분히 이해하고 있지 않다는 것을 나타낸다.

깊은 중첩 구조는 루틴을 분리하거나 복잡한 코드의 일부분을 재설계해야 할 필요가 있다는 경고 신호이다. 반드시 해당 루틴을 수정해야 한다는 것은 아니지만, 그렇게 하지 않는 경우엔 충분한 이유가 있어야 한다.

구조적 프로그래밍의 핵심은 오직 하나의 입구와 출구만이 있는 제어 구조(단일 진입점과 단일 탈출점 제어 구조라고도 불린다)를 사용해야 한다는 간단한 개념이다. 하나의 입구와 출구가 있는 제어 구조는 시작할 수 있는 곳과 끝날 수 있는 곳이 하나 뿐인 코드 블록이다. 다른 진입점이나 탈출점은 없다. 구조적 프로그래밍은 구조적인 하향식 설계와 동일하지 않으며, 상세한 코드 작성 단계에만 적용한다. 구조적 프로그램 이론의 핵심은 모든 제어 구조를 (순서, 선택, 반복)으로부터 만들어낼 수 있다는 것이다.

제어 구조에 이렇게 많은 주의를 기울여야 하는 이유 중 하나는 제어 구조가 프로그램의 전체 복잡도에 지대한 영향을 미치기 때문이다. 제어 구조를 잘못 사용하면 복잡도가 증가하고, 잘 사용하면 복잡도가 감소한다.

"프로그래밍 복잡도"를 측정하는 한 가지 방법은 프로그램을 이해하기 위해서 한꺼번에 기억해야 하는 머릿속 객체의 수이다. 이러한 정신적인 요술 행동(juggling act)은 프로그래밍을 할 때 가장 어려운 부분 중 하나이며, 동시에 프로그래밍이 다른 행동보다 더 많은 집중력이 요구되는 이유이기도 하다. 그리고 프로그래머들이 "방해"를 받을 때 화가 나는 이유이기도 하다. 여기서의 "방해"라는 말은 곡예사에게 세 개의 공을 공중에 던져놓고 그와 동시에 다른 물건도 집어보라고 요구하는 것과 같다.

더 이상 단순해질 수 없을 때까지 단순하게 만들어라 - 아인슈타인

한 루틴 내에 있는 변수의 수가 복잡도와 연관이 있다.

복잡도가 대단히 높은 코드를 절대로 다룰 수 없기 때문에 가능한 한 복잡도를 줄이기 위한 단계를 따라야 한다.

Tom McCabe 의 'A Complexity Measure' 에 따르면, 결정점(decision points)의 수를 세어 복잡도를 측정한다. 분기를 일으키는 keyword(if, case, for등) 가 나올 때마다 +1을 한다. 이 점수가 0~5 는 괜찮고, 6~10은 단순화가 필요하고, 10+ 은 하위 루틴으로 나누자.

소프트웨어 품질을 유지하는 데 있어서 한 가지 큰 걸림돌은 통제되지 않은 변경이다.

여러 결함 발견에 따라서 결함 수정 비용은 다르다. 코드 조사의 경우에는 바로 원인을 찾아낸 셈이지만, 테스트로 찾아낸 결함은 직접적인 원인을 찾는 비용이 더 든다.
(진욱:버그가 발생하면 재현 테스트도 좋지만 관련 코드의 검토를 먼저 하는 것도 좋을듯)

대부분의 프로젝트에서 가장 큰 단일 활동은 정상적으로 작동하지 않는 코드를 디버깅하고 수정하는 것이다. 디버깅과 리팩토링, 그리고 다른 수정 작업이 전형적인 소프트웨어 개발 주기에서 약 50% 정도의 시간을 차지한다. 오류를 예방하여 디버깅을 줄이면 생산성이 향상된다. 따라서 개발 일정을 줄이는 가장 확실한 방법은 제품의 품질을 향상시키고 디버깅과 소프트웨어의 수정 작업으로 낭비되는 시간을 줄이는 것이다.

여러분의 지식이 모두 경험으로부터 온 것이라면, 계속 읽어라. 다른 사람들은 각자 다른 경험을 갖고 있으므로, 여러분은 새로운 개념들을 발견하게 될 것이다.

검토는 경험이 많고 적은 프로그래머들이 기술적인 문제에 대해서 대화할 수 있는 장을 마련해 준다. 형식적인 정밀 검사를 사용한 한 팀은 정밀 검사를 통해서 모든 개발자들이 빠르게 가장 훌륭한 개발자의 수준까지 도달할 수 있었다고 보고했다.

여러 사람이 코드를 보고 코드를 다루면 코드의 품질이 좋아진다.

짝 프로그래밍을 할 때, 한 프로그래머는 키보드로 코드를 입력하고 다른 프로그래머는 실수를 감시하면서, 코드가 정확하게 작성되고 있는지 그리고 올바른 코드가 작성되고 있는지에 대해서 전략적으로 생각한다. 짝 프로그래밍을 할 때는 코딩 컨벤션이 정해져 있어야 한다. 코딩 컨벤션으로 시간 낭비하지 말자.

짝 프로그래밍을 모든 곳에 적용하려고 강요하지 말라. 혼자하는게 더 나은 것도 있다. 정기적으로 짝과 작업을 교대하라.

짝 프로그래밍은 혼자서 개발할 때보다 압박을 더 잘 견딘다. 짝은 코드를 빠르고 엉망으로 작성하도록 만드는 압력이 있을 때에도 코드의 품질을 높게 유지할 수 있도록 서로를 격려한다.

정밀 검사 자체의 핵심은 설계나 코드에 있는 결함을 발견하는 것이다. 다른 대안을 찾거나, 누가 옳고 누가 그른지에 대해서 논쟁하는 것이 아니다. 요점은 설계나 코드를 작성한 사람을 비판해서는 안 된다는 것이다. 정밀 검사는 작성자로 하여금 회의에 참석하는 것이 프로그램을 개선할 수 있다는 확신을 줄 수 있어야 하고 참여한 모든 사람들에게 무언가를 배울 수 있는 경험이 되는 긍정적인 효과를 가져와야 한다.

정밀 검사 체크 리스트가 있으면 좋다. 그 리스트를 중점으로 관리하면 리뷰가 좀 더 집중적이 될 수 있고, 많이 발견되는 오류들이 중점적으로 체크될 수 있다. 단, 한 페이지가 넘지 않도록 관리를 하자. 더 이상 자주 발견되지 않는 항목은 빼자. 체크 리스트를 사용하면 정밀 검사가 체계적이 된다. 그리고 미리 형식적인 체크를 하기 때문에 스스로 최적화가 된다.

정밀 검사는 수정보다는 결함의 발견에 초점을 맞추어야 한다.

회귀 테스트는 이전에 통과했던 테스트 집합을 가지고 소프트웨어에 있는 결함을 찾기 위해서 이전에 실행했던 테스트 케이스를 반복하는 것이다.

시스템 테스트는 다른 소프트웨어와 하드웨어 시스템과의 통합을 포함한 최종 환경에서 소프트웨어를 실행시키는 것이다. 이 테스트는 보안, 성능, 자원 손실, 시간 문제, 그리고 저수준 통합에서는 테스트될 수 없는 다른 문제들을 테스트한다.

테스트는 오류를 발견하기 위한 방법이며, 디버깅은 이미 발견된 오류의 원인을 진단하고 수정하는 방법이다.

테스트 자체는 소프트웨어의 품질을 향상시키지 않는다. ...중략... 만약 소프트웨어를 향상시키고 싶다면, 테스트를 더 많이 하지 말고 더 잘 개발하도록 한다.

테스트를 통해서 발견한 결함의 기록은 가장 자주 발생하는 오류의 종류를 찾는 데 도움을 준다. 적절한 교육 과정을 선택하고, 이후의 기술적인 검토 작업을 지시하고, 이후의 테스트 케이스를 설계하는 데 이 정보를 이용할 수 있다.

테스트 케이스를 먼저 작성하면 코드를 작성하기 전에 최소한 요구 사항과 설계에 대해서 좀 더 생각하게 되며, 이는 더 좋은 코드를 만든다.

테스트 케이스를 먼저 작성하면 코드가 작성되기 전에 요구 사항에 있는 문제를 조기에 노출시킨다. 왜냐하면 요구 사항이 잘못되어 있으면 테스트 케이스를 작성하기가 어렵기 때문이다.

대체로, 테스트 우선(test-first) 프로그래밍은 지난 수십 년 동안 나타난 소프트웨어 기법 중에서 가장 유용한 것 중 하나라고 생각한다.

개발자 테스트는 테스트 커버리지를 낙관적으로 바라보는 경향이 있다. 일반적인 프로그래머들이 스스로 95%의 테스트 커버리지를 달성하고 있다고 믿지만, 전형적으로 최고 80%에서 최소 30% 정도, 그리고 평균적으로 50~60% 정도의 테스트 커버리지를 달성한다.

현실적으로 말하자면 완전한 테스트는 불가능하기 때문에, 가장 오류를 잘 발견할 것 같은 테스트 케이스를 선택하는 것이 바로 테스트의 기술이다.

수동으로 점검하기에 편리한 테스트 케이스를 사용하라. 사실 경계 숫자가 아니라면 5235235와 20000 은 같은 종류의 테스트 데이터이다. 2000을 쓰자.

오류는 균등하게 배포되어 있지 않고, 몇몇 루틴에 집중되어 있다. 많은 오류를 포함하기 쉬운 루틴들은 지나치게 복잡한 루틴들이다. 오류를 유발할 가능성이 있는 코드를 찾아서 재설계한 다음, 재작성하도록 한다.

설계를 잘못 이해하는 것은 오류에 대한 연구에서 계속해서 나타나는 주제이다. ...중략... 따라서 설계를 완전하게 이해하기 위해서 충분한 시간을 가질 필요가 있다.

디버깅은 품질을 올리는 방법이 아니라, 결함을 진단하는 방법이다. 소프트웨어 품질은 처음부터 확립되어야 한다. 고급 제품을 만드는 가장 좋은 방법은 요구 사항을 주의 깊게 계발하고, 설계를 잘하고, 고급 코드 작성 방법을 사용하는 것이다. 디버깅은 최후의 수단이다.

프로그램이 무엇을 하고 있는지 정확하게 이해하지 못했다면, 시행착오를 통해서 프로그래밍을 하고 있는 것이다. 그리고 만약 시행착오를 통해서 프로그래밍을 하고 있다면, 결함이 반드시 생기게 된다.

디버깅 문제를 해결하기 위한 접근 방법에 확신이 드는가? 접근 방법이 작동하는가? 결함을 빨리 찾았는가? 또는 디버깅을 하기 위한 접근 방법이 취약한가? 괴로움이나 좌절감을 느끼는가? 임의대로 추측하는가? 개선이 필요한가? ...중략... 디버깅하는 방법을 분석하고 변경하는 시간을 갖는 것은 프로그램을 개발할 때 드는 전체 시간을 줄이는 가장 빠른 방법일 것이다.

문제가 아니라 증상을 변경하는 goto 미봉책이나 특별한 case를 적용하여 가장 쉬운 방법으로 결함을 수정하고 있는가? 또는 문제의 핵심을 정확하게 진단하고 처방하여 체계적으로 수정하고 있는가?

아래와 같이 가장 명백한 수정으로 오류를 수정하고 있지 않은가? Compute() 를 살펴볼 필요가 없다고 생각하지 않나?

```
x = Compute(y)
if ( y == 17 ) {
  x = 25
}
```

디버깅은 결함을 발견하고 수정하는 것으로 구성된다. 일반적으로 결함을 찾고 결함을 이해하는 것이 전체 작업의 90% 정도를 차지한다.

다음은 결함을 찾는 효율적인 접근 방법이다.

1. 오류를 안정화시킨다.
2. 오류의 원인을 찾아낸다.
    - 2.1. 결함을 만들어내는 데이터를 수집한다.
    - 2.2. 수집된 데이터를 분석하고 결함에 대한 가설을 세운다.
    - 2.3. 프로그램을 테스트하거나 코드를 살펴봄으로써 그러한 가설을 증명하거나 반증할 방법을 결정한다.
    - 2.4. 2.3 에서 규명한 절차를 사용하여 가설들을 증명하거나 반증한다.
3. 결함을 수정한다.
4. 수정한 내용을 테스트한다.

5. 유사한 오류를 찾는다.

예측 가능하게 발생하지 않는 오류는 일반적으로 초기화 오류, 시간 문제, 또는 허상 포인터(dangling-pointer) 문제이다.

이전에 결함이 있었던 클래스와 루틴들을 의심하라. 이전에 결함을 갖고 있었던 클래스들은 계속해서 결함을 갖기가 쉽다. 과거에 문제가 있었던 클래스들은 결함이 없던 클래스보다 새로운 결함을 포함할 가능성이 높다.

문제를 수정하기 전에 문제를 충분히 이해해라. 가능하다면 프로그램의 전체적인 작동을 이해해라.

문제를 성급하게 해결하지 말아라

코드를 임의로 변경하지 않는다. 그것은 주술적인 프로그래밍이다. 여러분이 코드를 이해하지 않고 더 많은 방법으로 코드를 변경할수록, 코드가 정확하게 작동할 것이라는 자신감은 점점 떨어진다. 변경하기 전에, 작동할 것이라는 것을 확신해야 한다. 변경을 한 결과가 틀리면 반드시 경악해야 한다. 자신을 의심하고, 자신을 재평가하고, 깊은 반성을 해야 한다.

한 번에 한 가지만 변경한다. 변경은 한 번에 하나만 수행되어야 할 정도로 신중을 요한다. 한번에 두가지를 변경하면, 원본 오류와 같은 미묘한 오류가 발생할 수 있다. 그러면 자신이 오류를 수정했는지, 오류를 수정했지만 비슷한 새로운 오류를 만들었는지, 또는 오류를 수정하지도 않고 비슷한 새로운 오류를 만들었는지를 알지 못하게 되는 곤란한 상황에 빠져버린다. 변경은 단순하게, 한 번에 한 가지만 변경한다.

결함을 만났을 때에는 그 결함이 다시 발생하지 않도록 오류를 노출시키는 테스트 케이스를 추가한다.

결함 하나를 찾았을 때, 그와 유사한 다른 결함들을 찾아본다. 결함은 그룹으로 발생하는 경향이 있기 때문에, 결함의 종류에 주의를 기울이면 그와 같은 종류의 모든 결함들을 수정할 수 있다. 다만, 유사한 결함을 찾기 위해서는 문제를 완전하게 이해해야 하고 경고 신호에 주의해야 한다. 만약 유사한 결함을 찾는 방법을 알 수 없다면, 여러분이 문제를 완벽하게 이해하지 못했다는 신호이다.

디버깅을 할 때의 심리적인 상태를 주의하자. 특히 비슷한 변수명을 섞어쓰는 것은 위험할 수 있으므로 미리 방지하거나, 주의를 기울이자. 이 책에서는 변수명의 '심리적인 거리' 라고 한다.

컴파일러 경고가 실제로 무엇을 의미하는지 이해하기 위해서 노력하라.

실행 프로파일러를 사용하면 숨겨진 결함들을 발견할 수 있다. 자신의 예상과 다른 프로파일링 결과를 얻었을 때 왜 그런지를 의심해보자.

마틴 파울러는 리팩토링을 "소프트웨어를 보다 쉽게 이해할 수 있고, 적은 비용으로 수정할 수 있도록 겉으로 보이는 동작의 변화 없이 내부 구조를 변경한 것"으로 정의하였다.

시스템을 개선하기 위한 한 가지 방법은 모듈화를 증가시키는 것이다. 즉, 한 가지 기능만을 잘 처리하며 잘 정의되고, 이름이 좋은 루틴들의 수를 증가시키는 것이다.

병렬 수정의 필요성 제거하기 - 변경을 할 때 여러 개의 클래스를 동시에 수정해야 한다면, 리팩토링의 대상일 수 있다. 이와 비슷하게, 하나의 서브클래스를 만들 때 다른 서브클래스도 항상 같이 만들거나, case 문을 변경할 때 다른 곳의 case 문도 같이 변경되어야 한다면 리팩토링의 대상이다.

어떤 모듈들은 다른 모듈보다 오류를 유발할 가능성이 높고 불안정하다. 여러분과 여러분의 팀에 있는 다른 모든 사람들이 두려워하는 코드가 있는가? 아마도 그 모듈은 오류를 유발할 가능성이 높을 것이다.

오래된 제품 시스템을 보수하기 위한 효율적인 전략은 혼란스러운 실세계에 있는 코드와 이상적인 새로운 세계에 있는 코드, 그리고 두 세계를 연결해 주는 인터페이스에 있는 코드들을 분류하는 것이다.

코드 최적화 때문에 직관적인 코드를 다루도록 설계된 훨씬 강력한 컴파일러가 최적화하지 못할 수도 있다는 점이다.

너무 이른 최적화는 모든 악의 근원이다. 프로그램이 완벽하게 작동하기 전까지는 성능 병목을 규명하기가 거의 불가능하다. 그렇기 때문에 코드를 작성하면서 최적화에 매달리는 것은 평균적으로 최적화될 필요가 없는 코드를 최적화하고 있는 것일 확률이 높다.

만약 프로그램이 완성되기 전에 최적화를 해야 한다면, 목표를 명시한 다음 작업을 하면서 그 목표를 달성하기 위해서 최적화를 하는 것이다. 명세서에 그러한 목표를 설정하는 것은 개별적인 나무가 얼마나 큰지를 이해하는 동안에도 한쪽 눈으로 계속해서 숲을 바라보는 방법이다.

고급 설계를 사용하라. 프로그램을 올바로 만들어라. 나중에 작업하기 쉽도록 모듈화하고 변경이 쉽도록 만들어라. 정확하게 완성되었을 때, 성능을 검사하라.

늘 그런 것처럼, 하루만 작업하면 코드에 있는 분명한 병목들을 규명할 수 있다.

컴파일러는 교묘한 코드를 최적화하는 것보다 직관적인 코드를 최적화하는 데 능숙하다.

4K 페이지를 사용한다면, 해당 연산 혹은 루프에서 4K 페이지를 계속 벗어나면서 동작하고 있지 않는지 확인하자. table[row][col] 인데 가까운 루프가 row++ 하고 있지 않은지 체크하자. 이렇게 되면 계속 캐시를 비웠다가 올렸다가 하게 된다.

시스템 루틴에 대한 호출은 종종 비싸다. 시스템 루틴들은 디스크, 키보드, 스크린, 프린터, 또는 다른 장치에 대한 입력/출력 연산과 메모리 관리 루틴들, 그리고 특정한 유틸리티 루틴들을 포함한다.

연산들의 상대적인 성능은 매우 많이 변한다. 따라서 만약 성능에 대해서 10년 묵은 개념으로 코드 최적화를 접근하고 있다면, 아마도 자신의 사고를 갱신해야 할 필요가 있을 것이다.

경험은 최적화에 있어서 큰 도움을 주지 못한다. 개인의 경험은 오래된 기계, 언어, 또는 컴파일러로부터 얻은 것이기 때문에, 그러한 것들이 변하게 되면, 경험도 필요 없게 된다. 여러분은 그 효과를 측정하기 전까지는 최적화의 효과에 대해서 절대로 장담할 수 없다.

한 가지 최적화 기법으로 10배가 향상되는 경우는 거의 없을 것이다. 하지만 꾸준히 시도해 보도록 한다. 여러 최적화 기법들이 결합되면 의미있는 결과를 만들어 낸다. 코드 최적화를 통해서 원하는 만큼의 성능을 개선하기 위해서는 일반적으로 여러 번 반복해야 한다.

초기 코드 작성 시 성능 작업을 준비하는 가장 좋은 방법은, 이해하고 수정하기 쉬운 분명한 코드를 작성하는 것이다.

코드 최적화 변경은 성능을 향상시키는 대신 내부 구조를 손상시킨다. ... 만약 변경 사항이 내부 구조를 손상시키지 않는다면, 우리는 그것을 최적화로 생각하지 않을 것이다. 그러한 기법은 기본적으로 사용하는 것이며, 그러한 기법을 코드 작성 방법의 표준으로 생각할 것이다.

특정 최적화 방법이 모든 언어에서, 특히 모든 컴파일러에서 좋은 효과를 얻는다고 생각하지 말아라. 꼭 측정을 해보자.

특정한 최적화를 수행하면 성능에 별다른 효과도 거두지 못한 채, 코드의 유지 관리성은 더 떨어지는 결과를 가져올 수 있다.

감시 값 사용 : 찾기를 원하는 값을 배열의 마지막 다음(!!) 위치에 넣은 후, while, if 의 조건 체크에서 감시값과 같은지만 체크하는 방법. 꽤 성능을 올릴 수 있다. 다만, 감시값을 잘 설정해야 한다.

가장 빈번한 루프를 안쪽에 작성한다. 루프의 시작도 부하라는 것을 잊지 말자.

강도 감소 : 일반적으로 곱셈보다 덧셈이 빠르다. 나눗셈보다 곱셈이 빠르다. 부동 소수점 연산보다 정수 연산이 더 빠르다.

기본 제공되는 함수보다 정확도가 덜 필요하다면, 직접 시스템 함수를 만들어 쓰는 것도 좋다. 예를 들면 log(x) 의 반환값으로 int 만 필요하다면 좀 더 싸게 log(x) 를 만들 수도 있을 것이다.

프로그램의 크기가 구현에 미치는 영향

프로젝트의 크기가 증가함에 따라서, 구현은 선형 증가하지만 그 외의 아키텍쳐나, 통합, 시스템 테스트 등이 더 빠르게 증가한다.

다른 조건들이 동일하다면, 큰 프로젝트의 생산성은 작은 프로젝트보다 낮을 것이다.

다른 조건들이 동일하다면, 큰 프로젝트는 작은 프로젝트보다 1,000줄당 오류의 수가 더 많을 것이다.

경량 방법론을 확대하는 것은 중량 방법론을 축소하는 것보다 더 잘 작동하는 경향이 있다. 가장 효율적인 접근 방법은 “올바른” 접근 방법을 사용하는 것이다.

구현 관리

코드가 구현의 주요 산출물이기 때문에, 구현을 관리하는 데 있어서 핵심적인 사항은 “좋은 코드 작성 습관을 어떻게 장려할 것인가?”이다. 일반적으로, 관리자가 엄격한 기술적인 표준을 정하는 것은 좋은 생각이 아니다. ...중략... 만약 프로그램 표준이 있어야 한다면, 프로그래머가 정할 필요가 있다.

프로젝트의 모든 부분에 두 사람을 할당한다.

모든 코드를 검토한다. 코드 검토에는 전형적으로 한 명의 프로그래머와 적어도 두 명의 검토자가 필요하며, 이는 적어도 세 명의 사람들이 모든 코드를 읽어본다는 것을 의미한다. 동료에 의한 검토를 “동료의 압박”이라고도 한다.

검토를 위해서 좋은 코드 예제를 돌려본다.

만약 큰 프로젝트를 예측하고 있다면, 예측을 하나의 작은 프로젝트로 취급하여, 예측을 잘 할 수 있도록 예측을 위한 계획을 수립하는 시간을 갖도록 한다.

저수준의 세부 사항을 예측하라.

여러 가지 다양한 예측 기법을 사용하고 그 결과를 비교하라.

주기적으로 재예측하라.

Brooks의 설명에 따르면...중략... 새로운 사람은 무언가를 생산할 수 있기 전에 프로젝트에 익숙해지기 위한 시간이 필요하기 때문이다. 그들의 훈련은 이미 훈련을 받은 사람들의 시간을 빼앗게 된다. ...중략... 하지만 만약 프로젝트의 작업이 나뉠 수 있다면, 프로젝트의 일정이 늦어졌다고 하더라도 프로젝트를 나누어 서로 다른 사람들에게 일을 할당할 수 있다.

만약 일정에 늦어지게 되면, “선택적인 것”과 “가지면 좋은 것”에 대한 우선순위를 매겨서 가장 덜 중요한 것을 뺀다.

어떤 프로젝트든지 간에, 측정을 하는 것이 측정을 하지 않는 것보다는 낫다.

통합

한 번에 여러 모듈들을 통합할 수도 있지만, 하나의 뼈대 모듈을 만든 후 점진적으로 하나씩 붙여나가자.

점증적 통합 전략에는 하향식 통합, 상향식 통합, 샌드위치 통합, 위험 지향적인 통합, 기능 지향적인 통합, T자형 통합이 있다.

소프트웨어 설계 접근 방법처럼 이러한 접근 방법들은 알고리즘이라기보다는 경험적이기 때문에, 어느 방법을 독단적으로 따르기보다는 여러분이 진행하는 프로젝트에 맞는 고유한 전략을 만들어야 한다.

프로그램은 프로그램을 실행했을 때 "연기를 내뿜는지" 확인하기 위한 간단한 "스모크 테스트"를 거친다.

스모크 테스트 없는 일별 빌드는 앙코 없는 찐빵과 같다. 스모크 테스트는 제품의 품질을 떨어뜨리고 통합에서 발생하는 문제점을 서서히 악화시키지 못하도록 보호하는 파수꾼이다.

스모크 테스트는 시스템과 함께 발전해야 한다. ...중략... 만약 스모크 테스트가 최신으로 유지되지 않는다면, 일별 빌드는 자칫 테스트 케이스가 제품의 품질에 대한 잘못된 자신감을 불러일으켜 자기 기만에 빠질 수 있는 작업이 될 수 있다.

아침에 빌드를 배포한다. 빌드를 오후에 배포하면 테스터는 퇴근하기 전에 자동화된 테스트를 실행하고 싶을 것이다. 빌드가 지연되면(종종 있는 일이다), 테스터는 테스트를 시작하기 위해서 늦게까지 대기해야 한다. 테스터들이 늦게까지 대기하는 것은 그들의 잘못이 아니기 때문에, 빌드 프로세스는 사기를 떨어뜨리게 된다.

프로그래밍 도구들

...딱히 새로운 내용이 없었음...

배치와 방식

프로그래머가 본인이 작성한 코드를 몇 달이 지난 후에 코드를 수정할 때의 난이도는 다른 사람들이 코드를 읽고 이해하고 수정하는 난이도와 비슷하다.

좋은 코드를 좋아 보이게 만들고 나쁜 코드를 나쁘게 보이도록 만드는 기법이 모든 코드를 좋아 보이게 만드는 기법보다 더 유용하다.

...중략... 하지만 아무리 보기 좋아 보일지라도 6개의 공백을 사용한 들여쓰기는 가독성이 떨어진다는 것으로 판명되었다. 이는 미적인 호감과 가독성이 일치하지 않음을 보여주는 예이다.

결과는 컴파일러마다 다르지만 일반적으로 성능 이득을 측정할 때까지는 성능보다 명료함과 정확성을 위해서 노력하는 것이 좋다.

비록 부수적인 효과를 갖고 있는 명령문을 쉽게 읽을 수 있다고 하더라도, 여러분이 작성한 코드를 읽게 될 사람을 불쌍히 여겨라.

(이런 방식의) 가장 큰 문제점은 유지 보수하는 데 손이 많이 가게 된다는 점이고, 이처럼 유지 보수하는 데 어려운 방식은 결국 유지 보수가 되지 않는다는 점이다.

스스로를 설명하는 코드

만약 코드가 설명이 필요할 정도로 복잡하다면, 주석을 추가하는 것보다는 코드를 향상시키는 것이 거의 항상 좋은 방법이다.

...중략... 프로그램이 무엇을 하고 있는지 설명하기 위한 단어들이 쉽게 떠오르지 않기 때문에 주석을 작성하는 일이 어려운 경우가 있다. 이 경우는 여러분이 프로그램이 하는 일을 이해하고 있지 못하다는 신호이다.

유지 보수하기 어려운 주석은 제대로 유지 보수되지 않는다. 불필요하게 멋져 보이는 주석은 그 멋짐을 유지하기 위해서 유지 보수하기 어려운 경향이 있다. 예를 들면, 전체를 * 로 감싸는 주석은, 줄 길이가 변화되면 전체 * 의 위치를 수정해야 할지도 모른다.

만약 설계가 코드를 작성하기에는 너무 어렵다면, 주석이나 코드에 대해서 걱정하기 전에 설계를 단순화시켜라.

특별한 두 경우를 제외하면, 줄 끝 주석은 개념적인 문제점들을 갖고 있고 코드를 복잡하게 만들기 쉽다. 또한, 코드의 포맷을 맞추고 관리하기도 어렵다. 결론적으로, 피하는 게 상책이다.

코드를 보면 알 수 있는 주석이 아니라, 코드를 봐서는 알 수 없는, 코드의 의미를 담고 있는 주석을 작성하자. (inputString 에서 '$' 를 찾는다 => 명령어 종결자($)를 찾는다.)

혹은, 주석의 의미를 가지도록 함수 이름을 정하자. (명령어 종결자($)를 찾는다 주석을 제거하고, 해당 코드를 함수로 빼내고 함수 이름을 FindCommandWordTerminator() 로 변경)

코드를 읽는 독자에게 다음에 오는 내용을 알려주기 위하여 주석을 사용한다. 훌륭한 주석은 코드를 보는 사람에게 무엇이 올 것인지 말해준다. 독자는 주석만 읽고도 코드가 무엇을 수행하는지, 그리고 특정한 기능을 어디서 찾아야 하는지에 대한 아이디어를 얻을 수 있어야 한다. 이러한 규칙이 성립하기 위해서는 주석이 설명하는 코드보다 항상 앞에 위치해야 한다.

좋은 프로그래밍 방식을 어긴 것에 대해서 이유를 설명하라. 그렇게 해야 좋은 의도를 가진 프로그래머가 그 코드를 더 좋은 방식으로 변환하여 코드를 망가뜨리지 못하게 할 것이다.

나쁜 코드를 문서화하지 말라. 그런 코드는 다시 작성하라.

여러분은 스스로에게 "이 코드가 교묘한가?" 라고 물어봐야 한다. 여러분은 언제나 교묘하지 않은 방법을 찾을 수 있기 때문에, 코드를 다시 작성해라. 주석이 필요 없을 정도로 코드를 작성한 다음, 코드를 더욱 향상시키기 위해서 주석을 작성하라.

이러한 충고는 주로 여러분이 처음 작성하는 코드에 적용된다. 만약 프로그램을 유지 보수하고 있는데 잘못된 코드를 재작성할 수 있는 권한이 없다면, 교묘한 부분에 대해서 주석을 작성하는 것이 좋은 습관이다.

루틴 헤더(함수 앞의 주석)가 너무 무거울 때의 또 다른 문제점은, 코드에 대해서 리팩토링을 꺼리게 만든다는 점이다.

설명하고자 하는 코드와 가까운 곳에 주석을 작성한다.

루틴이 미치는 전역적인 효과를 문서화하라. (전역변수의 변경은 위험하므로...)

다른 프로그래머가 클래스의 구현을 살펴보지 않고 클래스 사용법을 이해할 수 있는가? 만약 그렇지 않다면, 클래스의 캡슐화에 심각한 문제가 있다.

클래스 인터페이스에 세부 구현 내용을 기술하지 않는다. 캡슐화의 기본적인 원칙은 알아야 하는 기본적인 정보만을 노출하는 것이다. 만약 정보가 노출되어야 할 필요가 있는지 혹은 없는 지에 대한 의문이 든다면, 기본적으로 감추도록 한다.

개인 성격

많은 프로그래밍 습관들의 목표는 두뇌 세포의 짐을 덜어주는 것이다.

루틴을 짧게 만들면 두뇌의 부담이 줄어든다.

경험적으로 자신의 오류 가능성을 보충하는 겸손한 프로그래머는 자신뿐만 아니라 다른 사람들도 이해하기 쉽고 오류가 적은 코드를 작성하는 것으로 밝혀졌다.

제대로 이해하지 못하는 큰 프로그램으로 기능을 살펴보는 것보다는 개념을 확인하기 위해서 간단한 프로그램을 작성하는 것이 더 좋다.

효율적인 프로그램을 위한 한 가지 핵심은 실수를 더 빨리 하고, 실수로부터 배우는 것이다. 실수를 하는 것은 죄악이 아니다. 실수로부터 배우지 못하는 것이 죄악이다.

문제 해결에 관한 책을 읽자. ...중략... 그들은 같은 전략을 다른 사람들로부터 쉽게 학습할 수 있었음에도 불구하고 항상 현명한 문제 해결 전략을 찾아내지 못한다는 점을 발견하였다. 이 말은 곧, 여러분이 바퀴를 다시 발명하려고 할지라도, 성공할지는 보장할 수 없다는 것을 의미한다. 아마도 바퀴 대신 네모진 것을 발명할지도 모른다.

행동을 취하기 전에 분석하고 계획을 세운다.

프로그래밍을 배우기 위한 특히 좋은 한 가지 방법은 훌륭한 프로그래머의 작업을 학습하는 것이다.

만약 훌륭한 프로그래밍 책을 두 달마다 읽는다면(대개 일주일에 35페이지 정도), 여러분은 곧 이 산업을 확실히 이해하게 되고 다른 사람과 차별화될 것이다.

훌륭한 프로그래머는 더 훌륭해지기 위한 방법을 지속적으로 찾는다.

사람들이 읽기 어려운 코드를 작성하는 가장 큰 이유는 코드에 대한 이해가 불충분하기 때문이다. 그들은 스스로에게 “내 코드는 나쁘기 때문에, 읽기 어렵게 만들거야”라고 말하지 않는다. 단지 읽기 쉽게 작성할 수 있을 만큼 코드를 충분히 이해하고 있지 못하기 때문이며, 그 때문에 그들은 낮은 단계에 머물러 있게 된다.

실수를 인정하지 않는 것은 매우 짜증나는 습관이다. 만약 샐리가 실수를 인정하지 않는다면, 명백히 그녀는 자신이 실수를 인정하지 않으면 다른 사람들이 자신이 실수한 것을 모를 것이라고 믿고 있을 것이다. 사실은 그 반대다. 모든 사람들이 그녀가 실수했다는 것을 알고 있을 것이다.

컴파일러 메시지를 이해하지 못하면서 이해하는 것처럼 행동하는 것도 또 다른 일반적인 맹점이다.

중요하지 않은 분야에서 규약을 만들어둠으로써 중요한 분야에서 창조력을 집중시킬 수 있다.

필자가 항상 바빠 보이는 프로그래머와 일했을 때, 필자는 그가 훌륭한 프로그래머가 아니라고 생각했다. 왜냐하면 그는 가장 유용한 도구인 그의 머리를 활용하고 있지 않기 때문이다.

만약 30분 내에 이 접근 방법을 사용하여 문제를 해결할 수 없다면, 10분 동안 다른 접근 방법을 생각해낸 후, 1시간 동안 최적의 해결책을 시도해 볼 것이다.

가장 중요한 성격들은 겸손과 호기심, 지적인 정직함, 창의성과 훈련, 현명한 게으름이다.

많은 프로그래머들이 능동적으로 새로운 정보와 기술을 찾지 않는다. 대신 그들은 일이나 우연적으로 새로운 정보를 얻는다. 만약 프로그래밍에 관한 책을 읽고 배우는 데 약간의 시간을 투자한다면, 몇 달 혹은 몇 년이 지난 후 다른 일반적인 프로그래머들과 확연하게 차별화된 자신의 모습을 발견할 수 있을 것이다.

훌륭한 성격은 주로 올바른 습관을 갖는 문제이다. 뛰어난 프로그래머가 되기 위해서는 올바른 습관을 개발하여 나머지 것들이 자연스럽게 따라올 수 있도록 해야 한다.

소프트웨어 장인(匠人)에 대한 주제

34.1 복잡성 정복

한번에 시스템의 작은 부분에 집중할 수 있도록 하나의 시스템을 아키텍처 수준에서 서브시스템으로 나눈다.

기능에 따라서 변수의 이름을 짓는 방식, 즉 구현 수준에서 “어떻게” 보다는 문제가 “무엇”인지를 나타내기 위하여 변수 이름을 지으면 추상화 수준을 증가시킨다. 만약 “내가 스택을 pop 한다는 것은 가장 최근에 입사한 사원을 가져오는 것을 의미한다”라고 말한다면, 추상화를 통해서 “나는 스택을 pop한다”를 생략할 수 있다. 간단히 “최근에 입사한 사원을 얻는다”라고 말하면 된다.

요약하자면, 소프트웨어 설계와 구현의 주요 목적은 복잡성을 정복하는 것이다. 많은 프로그래밍 습관의 내부적인 목적은 프로그램의 복잡성을 줄이는 것이며, 복잡성을 줄이는 일은 효율적인 프로그래머가 되기 위한 가장 중요한 핵심 요소이다.

34.2 자신에게 맞는 프로세스 선택

만약 무엇을 만들고 있는지 모른다면, 뛰어난 설계를 만들 수 없다.

집을 짓기 전에는 반드시 기초 공사를 튼튼히 해야 한다. 만약 기초 공사가 끝나기도 전에 코드 작성에 바로 들어간다면, 시스템 아키텍처의 근본적인 사항들을 변경하기가 더욱 어려워질 것이다. 사람들은 구현하고자 하는 코드를 이미 작성했기 때문에 설계에 대해서는 감정적으로 투자할 것이다. 일단 집을 짓기 시작하면 잘못된 토대를 버리기가 힘들어진다.

34.3 컴퓨터보다 사람을 위한 프로그램을 작성하라

코드의 가독성을 강조하는 것이다.

(진욱 요약:코드는 한번 쓰여지지만 여러번 읽힌다. 며칠 후의 나도 다시 읽고, 디버깅할 때도 다시 읽고, 다른 사람이 리뷰할 때도 읽는다. 읽기 쉬운 코드를 작성해야 한다.)

읽을 때의 편리함보다는 작성할 때의 편리함을 추구하는 것은 잘못된 절약 방법이다.

개인적인 프로그램을 공용 프로그램으로 만드는 일 중 하나가 코드를 읽기 쉽도록 만드는 것이다.

34.4 언어에 얽매이지 말고 언어를 활용하는 프로그램을 작성하라

프로그래밍 사고를 언어가 기본적으로 제공하는 개념들에만 제한하지 않도록 한다.

언어에서 어설션을 지원하지 않는다면? assert() 루틴을 직접 작성하라.

34.5 규약을 이용하여 주의를 집중하라

규약은 (정답이 여러 개인) 프로그래머들이 동일한 질문에 답해야 하고 동일한 결정을 계속해서 해야 하는 수고를 덜어준다. 많은 프로그래머들이 참여하는 프로젝트에서는 규약을 사용하여 서로 다른 프로그래머들이 각기 임의의 결정을 내릴 때 발생하는 혼란을 막아준다.

규약은 언어의 약점을 보충할 수 있다. 규약을 이용하여 명명된 상수를 지원하지 않는 언어에서는 읽고 쓰기 위한 변수와 읽기 전용 상수를 나타내기 위한 변수들을 구분 지을 수 있다.

34.6 문제 중심의 프로그램

복잡성을 다루는 또 다른 방법은 가능한 가장 높은 추상화 단계에서 작업하는 것이다.

최상위 수준 노드는 파일과 스택, 큐, 배열, 그리고 l, j, k와 같은 문자에 대한 세부 사항들로 채워져서는 안 된다.

34.7 낙석을 주의하라**(**진욱**:**경고에 귀를 귀울여라**)**

여러분이나 다른 누군가가 "이것은 참으로 교묘한 코드이다"라고 말한다면, 이는 경고 표시이며 일반적으로 엉터리 코드이다.

평균 이상으로 많은 오류를 갖고 있는 클래스도 경고 표시이다. ...중략... 재작성하는 것이 좋다.

클래스에 있는 비정상적인 결함의 수가 클래스의 품질이 낮다는 것을 경고하는 것처럼, 프로그램에서의 비정상적인 결함의 수는 프로세스에 결함이 있음을 암시한다 좋은 프로세스는 오류를 내포하는 코드가 작성되지 않도록 한다. ...중략... 디버깅을 많이 해야 하는 프로젝트는 사람들이 현명하게 일하고 있지 않다는 경고 표시이다

프로그램에 깊이 빠져 있을 때, 프로그램 설계의 일부분이 코드를 작성하기에 충분할 정도로 잘 정의되어 있지 않다는 경고 표시에 주의를 기울인다. 주석을 작성하고 변수 이름을 짓고, 문제를 명확한 인터페이스를 갖는 응집된 클래스로 분리하는 데 어려움이 있다면, 이 것은 코드를 작성하기 전에 설계에 대해서 보다 열심히 생각해 보아야 한다는 것을 가리킨다. 우유부단한 이름이나 일련의 코드를 간결한 주석으로 기술하는 데 어려움을 겪고 있다는 점도 문제가 있다는 표시이다. 설계가 머릿속에서 명확할 때, 저수준에서의 세부 사항들이 쉽게 따라온다. 그리고 프로그램이 이해하기 어렵다는 증상에 민감해야 한다. 불편함이 그 단서이다. 만약 여러분이 이해하기 어렵다면 다른 프로그래머에게는 더욱 어려울 것이다. ...중략... 코드를 읽기보다는 이해하고 있다면, 너무 복잡한 것이다. 어려운 것은 잘못된 것이다. 단순하게 만들어야 한다.

34.8 반복**,** 반복**,** 또 반복

반복은 많은 소프트웨어 개발 행위에 적합하다.

소프트웨어는 검증되기보다는 확인되는 경향이 있다. 즉, 정확하게 답할 때까지 반복적으로 테스트하고 개발되어진다. 고수준과 저수준 설계 시도도 반복되어야 한다. 첫 번째 시도에서 작동하는 해결책을 만들 수 있겠지만, 최적의 해결책을 만들어내지는 않는다. 여러 번 반복되고 서로 다른 접근 방법을 취함으로써 한 가지 접근 방법만으로는 얻을 수 없는 문제에 대한 통찰력을 얻게 된다.

반복의 개념은 코드 최적화에서 다시 나타난다. ...중략... 최적화 기술의 효과에 대한 불확실성은 최적화, 측정, 그리고 재최적화를 필요로 한다.

버리기 위해서 만들어보라. 어떻게 해서든지 만들 수 있을 것이다.

소프트웨어 공학의 기교는 간단히 처리할 수 있는 부분들을 가능한 한 빨리, 그리고 적은 비용으로 만드는 것이다. 이것이 초기 단계에서의 반복의 요점이다.

각 개발 활동들을 반복하면 할수록, 제품은 더욱 더 좋아질 것이다.

34.9 소프트웨어와 신조를 떼어 놓아라**.**

최신 유행에 집착하기보다는, 여러 방법을 혼합하여 사용하라. 호기심을 자극하는 최신 방법을 경험하는 것도 좋지만, 오래되고 신뢰할 수 있는 방법에 의존하라.

어떤 것은 실패하고 어떤 것은 성공하겠지만, 어떤 것들이 그것을 시도해 보고 나서도 제대로 작동할 것인지 모른다. 여러분은 절충적이어야 한다.

소프트웨어 개발에 있어서 융통성 없는 접근 방법의 상당수는 실수에 대한 공포 때문이다. 실수를 피하기 위한 포괄적인 시도가 가장 큰 실수이다. 설계는 큰 실수를 피하기 위해서 작은 실수를 조심스럽게 계획하는 과정이다.

독단적인 방법론과 품질이 뛰어난 소프트웨어는 어울리지 않는다. 지적인 도구 상자를 프로그래밍 대안책들로 채우고, 일에 맞는 올바른 도구를 선택할 수 있는 기술을 향상 시켜라.

-- 이 책을 읽기 시작한 날짜 2014년 5월
-- 마지막 페이지를 읽은 날짜 2014-12-06
-- 2014-12-14 이 후기 페이지를 다시 한번 정독함