

클린 코드

By Robert C. Martin (Uncle Bob)

깨끗한 코드

나쁜 코드의 경험

■ 코드란?

- 기계가 실행할 수 있게 상세한 요구사항을 명시하는 작업의 결과
- 궁극적으로 요구사항을 표현하는 언어



■ 나쁜 코드의 경험

- 프로그래머라면 누구든 엉킨 코드 더미들과 숨겨진 함정으로 가득한 나쁜 코드의 늪을 힘들게 헤쳐나간 경험이 있다. 단서를 찾으려 애써도 소용이 없던 기억도...
- 왜 나쁜 코드를 짰는가?
 - 출시에 바빠서? 코드를 다듬느라 시간을 허비할 수 없어서? 밀린 다음 업무로 빨리 넘어가려고?...
 - 대충 짠 코드가 돌아간다는 사실에만 안주하고, 나중에 다시 정리하리라 다짐하지만,,,,,

나중은 결코 오지 않는다.

깨끗한 코드가 비용을 절감한다.

■ 나쁜 코드가 쌓일수록 개발 생산성은 떨어진다.

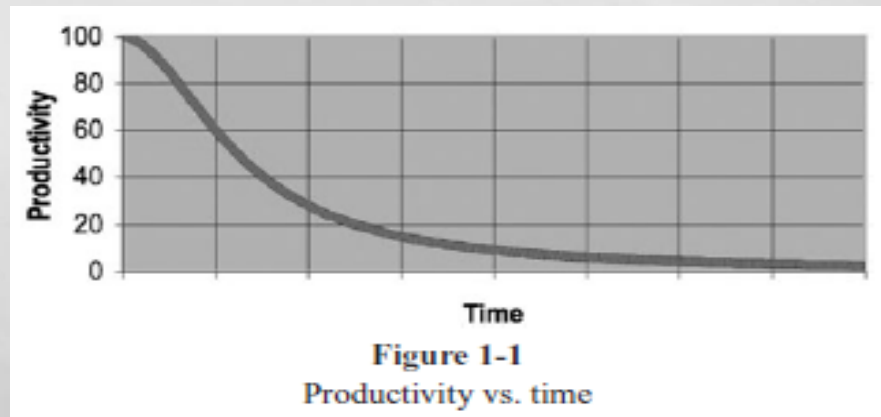
- 새로운 코드를 짜면서 끊임없이 기존 코드를 읽는다 (10 : 1 = 읽기 : 쓰기)
- 나중에 코드를 유지 보수할 사람이 코드 작성자 만큼이나 문제를 깊이 이해할 가능성은 희박
- 시스템이 복잡할수록 유지보수 개발자가 시스템을 이해하느라 보내는 시간이 길어지고

동시에 코드를 오해할 가능성도 커진다

- 나쁜 코드는 고칠 때마다 엉뚱한 곳에서 문제를 발생시킨다. 엉킨 코드를 해독해서 더 얽힌 코드를 덧붙인다. 시간이 지날수록 쓰레기 코드의 산은 점점 더 높아지기만 한다.

■ 새로운 시스템의 재설계, 개발?

- 새로운 시스템이 기존 시스템 기능들을 100% 제공하면서 추가되는 변경과 기능을 제공하지 않는다면, 기존 시스템을 대체하진 않는다.



깨끗한 코드를 사수하려는 태도

- 나쁜 코드의 양산하게 되는 이유

: 설계를 뒤집는 요구사항의 변경, 촉박한 일정, 무능한 관리자, 조급한 고객

- 나쁜 코드의 위험을 알지만, 관리자의 말을 따르는 행동은 전문가답지 못하다.

- 나쁜 코드로라도 기한을 맞추려다, 오히려 엉망진창이 되어 기한을 맞추지 못한다.

- 빨리 가는 유일한 방법은, 언제나 코드를 최대한 깨끗하게 유지하는 습관이다.

깨끗한 코드를 작성하려면?

- 깨끗한 코드를 작성하는 프로그래머는 빈 캔버스를 우아한 작품으로 바뀌가는 화가와 같다.
- 깨끗한 코드를 작성하려면,
 - 다양한 기법과 절차들을 알고 적용하는 절제와 규율이 필요하다.
 - 좋은 코드와 나쁜 코드를 구분할 줄 알아야 한다.
 - 나쁜 코드를 좋은 코드로 바꾸는 연습을 하면, 깨끗한 코드를 만드는 능력을 가질 수 있다.
 - 나쁜 코드를 좋은 코드로 바꾸는 전략도 파악한다.

깨끗한 코드 - 전문가 의견

비야네 스트루스트룹(Bjarne Stroustrup) : C++ 창시자

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.

깨끗한 코드는 한 가지를 제대로 한다.

- 우아한 코드

: 보기에 즐거운 코드, CPU 자원을 낭비하지 않는 코드

- 나쁜 코드는 나쁜 코드를 유혹한다. : **깨진 유리창의 법칙**(by 데이브 토마스와 앤디 헌트)

- 나쁜 코드를 고치려다 더 나쁜 코드를 만든다.

- 철저한 오류 처리

- 세세한 사항까지 꼼꼼하게 처리하는 코드가 깨끗한 코드이다.

- **깨끗한 코드는 한 가지를 제대로 한다. <= 소프트웨어 설계의 원칙**

- 각 함수와 클래스와 모듈은 너무 많은 일을 하려 하지 말고, 한 가지에 집중한다.

깨끗한 코드 – 전문가 의견

그래디 부치(Grady Booch) : UML 개발자

깨끗한 코드는 단순하고 직접적이다. 깨끗한 코드는 잘 쓴 문장처럼 읽힌다.

깨끗한 코드는 결코 설계자의 의도를 숨기지 않는다.

오히려 명쾌한 추상화와 단순한 제어문으로 가득하다.

■ 가독성 : 잘 쓴 문장처럼 읽힌다.

: 해결할 문제를 명확히 드러내고, 명백한 해법을 제시하여 풀어야 한다.

■ 명쾌한 추상화

- 코드는 사실에 기반해야 한다.
- 반드시 필요한 내용만 담아야 한다.

깨끗한 코드 – 전문가 의견

데이브 토마스(Dave Thomas) : OTI 창립자, 이클립스 전략의 대부

깨끗한 코드는 작성자가 아닌 사람도 읽기 쉽고 고치기 쉽다.

단위 테스트 케이스와 인수 테스트 케이스가 존재한다. 깨끗한 코드에는 의미 있는 이름이 붙는다.

특정 목적을 달성하는 방법은 (여러 가지가 아니라) 하나만 제공한다.

의존성은 최소이며 각 의존성을 명확히 정의한다. API 는 명확하며 최소로 줄였다.

언어에 따라 필요한 모든 정보를 코드만으로 명확히 표현할 수 없기에 코드는 문학적으로 표현해야 마땅하다.

- 깨끗한 코드는 다른 사람이 고치기 쉽다.
- 테스트 케이스가 없는 코드는 깨끗한 코드가 아니다.
- "최소" : 작은 코드에 가치를 둔다.

깨끗한 코드 – 전문가 의견

마이클 페더스(Michael Feathers) : 레거시 코드 활용 전략 저자

깨끗한 코드의 특징은 많지만 그 중에서도 모두를 아우르는 특징이 하나 있다.

깨끗한 코드는 언제나 누군가 주의 깊게 봤다는 느낌을 준다.

고치려고 살펴봐도 딱히 손 댈 곳이 없다. 작성자가 이미 모든 사항을 고려했으므로, 고칠 궁리를 하다 보면 언제나 제자리로 돌아온다.

그리고는 누군가 남겨준 코드, 누군가 주의 깊게 짜놓은 작품에 감사를 느낀다.

■ 깨끗한 코드는 주의를 기울여 작성한 코드다.

- 시간을 들여 깔끔하게 정리한 코드
- 세세한 사항까지 꼼꼼하게 신경 쓴 코드

깨끗한 코드 – 전문가 의견

론 제프리스(Ron Jeffries) : Extreme Programming Installer 저자

최근 들어 나는 켄트 벡이 제안한 단순한 코드 규칙으로 구현을 시작한다.(그리고 같은 규칙으로 구현을 거의 끝낸다.) 중요한 순으로 나열하자면 간단한 코드는

- 모든 테스트를 통과한다.*
- 중복이 없다.*
- 시스템 내 모든 설계 아이디어를 표현한다.*
- 클래스, 메서드, 함수 등을 최대한 줄인다.*

... 중략...

중복 줄이기, 표현력 높이기, 초반부터 간단한 추상화 고려하기, 내게는 이 세 가지가 깨끗한 코드를 만드는 비결이다.

깨끗한 코드 – 전문가 의견

워드 커닝햄(Ward Cunningham)

: 위키 창시자, 익스트림 프로그래밍 창시자.,,,

코드를 읽으면서 짐작했던 기능을 각 루틴이 그대로 수행한다면 깨끗한 코드라 불러도 되겠다.

코드가 그 문제를 풀기 위한 언어처럼 보인다면 아름다운 코드라 불러도 되겠다.

- 깨끗한 코드는 읽으면서 놀랄 일이 없어야 한다.
 - 코드를 독해하느라 머리를 쥐어짤 필요가 없어야 한다.
 - 읽으면서 짐작한 대로 돌아가는 코드가 깨끗한 코드다.
 - 언어를 단순하게 보이도록 만드는 열쇠는 프로그래머다.

깨끗한 코드의 정의

깨끗한 코드에 대한 절대적 정의는 없다.

개발자 수만큼이나 다양한 시각이 존재

=> 다른 사람의 의견을 이해/존중하되 자신만의 생각을 정리해야 한다.

깨끗한 코드의 중요 원칙

■ 지속적인 개선은 전문가 정신의 본질이다.

- 보이с카우트 원칙
: 캠프장은 처음 왔을 때보다 더 깨끗하게 해 놓고 떠나라.
- 체크 아웃할 때보다 좀 더 깨끗한 코드를 체크인 한다면 코드는 절대 나빠지지 않는다.
- 변수 이름 하나를 개선하고, 조금 긴 함수 하나를 분할하고, 약간의 중복을 제거하고, 복잡한 if 문 하나를 정리하면 충분하다



출처 : [빼꼼 시즌 2] 9화 보이с카우트

네이밍

의도가 드러나는 이름

■ 변수, 함수, 클래스 명명

- 변수, 함수, 클래스의 이름은 존재 이유, 수행 기능, 사용 방법이 드러나게 지어야 한다.
- 주석이 필요하다면 의도를 드러내지 못했음을 의미

```
unsigned int num;  
bool flag;  
std::vector<Customer> list;  
Product data;
```



```
unsigned int numberOfArticles;  
bool isChanged;  
std::vector<Customer> customers;  
Product orderedProduct;
```


의도가 드러나는 이름

- 변수, 함수, 클래스 명명
 - 맥락이 분명하거나 멤버 변수라면 가능한 짧게

```
unsigned int totalNumberOfCustomerEntriesWithMangledAddressInformation;  
  
totalNumberOfCustomerEntriesWithMangledAddressInformation =  
    amountOfCustomerEntriesWithIncompleteOrMissingZipCode +  
    amountOfCustomerEntriesWithoutCityInformation +  
    amountOfCustomerEntriesWithoutStreetInformation;
```



```
class CustomerRepository {  
private:  
    unsigned int numberOfMangledEntries;  
    // ...  
};
```

잘못된 정보를 담은 이름

- 널리 쓰이는 의미가 있는 단어를 다른 의미로 사용해선 안된다.(예: hp, aix, sco,...)
- 특수한 의미를 갖는 단어의 사용
 - accountList : List는 자료형으로 실제 타입을 의미하는 것인가???
 - 어떤 집합을 의미하는 것이라면 accountGroup, bunchOfAccounts, Accounts 가 더 좋은 선택
- 비슷한 이름은 사용하지 않는다.

같은 모듈에서 XYZControllerForEfficientHandlingOfStrings, XYZControllerForEfficientStorageOfStrings 를 사용한다면?
- 유사한 개념은 유사한 표기법을 사용
 - 일관성 있는 표기법도 정보가 됨
 - 코드 자동 완성 기능에서 핫키 사용이 수월해짐
(후보 목록에 유사한 개념 알파벳 순으로 나오고, 각 개념 차이가 명백히 드러난다면...)
- 소문자 L(l), 대문자 O(0) 변수

차이를 알 수 없는 이름

■ 연속적인 숫자를 덧붙인 이름(a1, a2, ...)

- 아무런 정보도 의도도 알 수 없는 이름

```
template <typename T, size_t N>
std::enable_if_t<!std::is_trivially_copyable<T>::value>
copy(T(&a2)[N], T(&a1)[N])
{
    for (size_t i = 0; i < N; ++i)
        a1[i] = a2[i];
}
```

=> a1, a2 를 source, destination 으로 변경

■ 개념의 구분 없이 이름만 달리한 경우

- 의미가 불분명한 불용어
: info, data, a, an, the, variable, table
- Product vs ProductInfo vs ProductData
- NameString vs Name
- Customer vs CustomerObject
- getActiveAccount();
getActiveAccounts();
getActiveAccountInfo();

발음하기 쉬운 이름

- 발음하기 어려운 이름은 토론하기 어렵다.

```
class DtaRcrd102 {  
private:  
    time_t genymdhms;  
    time_t modymdhms;  
    std::string pszqint;  
    ....  
}
```



```
class Customer {  
private:  
    time_t generationTimestamp;  
    time_t modificationTimestamp;  
    std::string customerId;  
    ....  
}
```

검색하기 쉬운 이름

- 한 글자 변수명과 상수는 코드 상에서 찾아내기 어렵다.
- 숫자 대신 constant로 정의해서 상수를 사용하는 것이 좋음
 - MAX_CLASSES_PER_STUDENT는 grep으로 찾기 쉽지만 7은 검색 어려움
- 이름 길이는 범위 크기에 비례해야 한다.

```
for (int j=0; j<3; j++) {  
    s += (t[j] * 4) / 5;  
}
```

// 4,5 를 찾아야 한다면???

```
int realDaysPerIdealDay = 4;  
const int WORK_DAYS_PER_WEEK = 5;  
const int NUMBER_OF_TASKS = 3  
int sum = 0;  
for (int j=0; j<NUMBER_OF_TASKS; j++) {  
    int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;  
    int reaktTaskWeeks = (realTaskDays / WORK_DAYS_PER_WEEK);  
    sum += reaktTaskWeeks;  
}
```

인코딩된 이름

- 유형, 범위 정보까지 인코딩된 이름은 해독하기 어렵다.
- 헝가리식 표기법 : 데이터 타입을 변수명 앞에 붙이는 표기법
 - weakly typed language like C 에서, 단순한 에디터에서는 유용할 수도 => not IDEs that have a feature like "IntelliSense." (요즘 컴파일러는 타입을 기억하고 강제한다.)
 - 다형성을 지원하는 object-oriented languages 에서 어떤 prefix를 사용할 것 인가.
 - 클래스와 함수가 작아지는 추세이므로, 변수 선언 위치와 사용 위치간의 거리가 가깝다.
 - 타입을 인코딩한 명명법은 변수, 함수, 클래스 이름이나 타입 변경을 어렵게 한다.
- 멤버변수 접두어
 - 멤버 변수에 m_ 접두어를 붙일 필요 없다.
 - IDE 가 멤버 변수를 눈에 띄게 보여준다.
- 인터페이스 클래스와 구현 클래스
 - 도형을 생성하는 추상 클래스 와 구현 클래스
IShapeFactory, CShapeFactory .VS. ShapeFactory, ShapeFactoryImpl

인코딩된 이름

■ 축약대신 full words를 사용

```
std::size_t idx; // Bad!
std::size_t index; // Good; might be sufficient in some cases
std::size_t customerIndex; // To be preferred, especially in situations where
// several objects are indexed
Car ctw; // Bad!
Car carToWash; // Good
Polygon ply1; // Bad!
Polygon firstPolygon; // Good
unsigned int nBottles; // Bad!
unsigned int bottleAmount; // Better
unsigned int bottlesPerHour; // Excellent!
const double GOE = 9.80665; // Bad!
const double gravityOfEarth = 9.80665; // More expressive, but misleading. The constant is
// not a gravitation, which would be a force in physics.
const double gravitationalAccelerationOnEarth = 9.80665; // Good.
constexpr Acceleration gravitationalAccelerationOnEarth = 9.80665_ms2; // Wow!
```

불필요한 중복 이름

- namespace와 type이 명확한 경우 => Title로 충분
 - 클래스의 이름을 attribute에 반복

```
class Movie {  
    // ...  
    private:  
        std::string movieTitle;  
    // ...  
};
```

- Attribute type을 attribute에 반복

```
class Movie {  
    // ...  
    private:  
        std::string title;  
};
```


기억력을 자랑하지 마라

- 변수 이름을 자신이 아는 이름으로 변환해야 한다면 바람직하지 않은 이름이다.
- 문자 하나만 사용하는 변수 이름은 문제가 있다.
 - Loop에서 반복횟수를 세는 변수: i, j, k는 허용. L은 절대 안됨.
 - a, b를 이미 사용했으므로 c를 사용한다는 것은 최악임.
- 똑똑한 프로그래머와 전문가 프로그래머
 - 똑똑한 프로그래머: r 이라는 변수가 호스트와 프로토콜을 제외한 소문자 URL 임을 기억하고 있다.
 - 전문가 프로그래머: 명료함이 최고이며 자신의 능력을 남들이 이해하는 코드를 작성하는 좋은 방향으로 사용

클래스 이름

■ 클래스 이름

- 대문자로 시작, 명사, 명사구가 적합, 동사는 사용하지 않는다.
- 좋은 예 : Customer, WikiPage, Account, AddressParser
- 나쁜 예 : Manager, Processor, Data, Info
- 여러 단어의 첫 글자만 따 약자나 max, min 처럼 널리 통용되는 줄임말을 제외하고는 줄여 쓴 이름은 쓰지 않는다.

■ 상수 필드(const, constexpr 필드)

- 변수와 동일하게 사용

■ 지역 변수

- 약어를 써도 좋으나, 변수가 사용되는 문맥에서 의미를 쉽게 유추할 수 있어야 한다.

메서드 이름

■ 메서드 이름

- 소문자로 시작, 동사, 목적어를 포함한 동사구가 적합
 - `postPayment`, `deletePage`, `save`
- boolean 값을 반환하는 메서드는 `is`, `has` 로 시작한다.
 - `isDigit`, `isEmpty`, `hasSiblings`
- 속성을 반환하는 메서드의 이름은 `get` 으로 시작.
 - `getSize`, `getTime`, `getHashCode`
- 속성을 변경하는 메서드의 이름은 `set` 으로 시작.
 - `setAttribute`, `setName`
- 객체의 타입을 바꿔서 다른 객체를 반환하는 메서드의 이름은 `toType` 형태로 짓는다.
 - `toString`, `toArray`
- 객체 내용을 다른 뷰로 보여주는 메서드는 `asType` 형태의 이름
 - `asList`
- 팩토리 메서드 이름
 - `Complex.fromRealNumber(23.0)` vs `new Complex(23.0)`

명료하지 못한 이름

- 독자가 코드를 읽으면서 자신이 아는 다른 이름으로 변환해야 한다면 바람직하지 않은 이름이다. 명료함이 최고다.
- 기발하거나 재미있는 이름보다 명료한 이름을 선택하라.
- 추상적인 개념 하나에 단어 하나를 고수하라.
 - 메서드 이름은 독자적이고 일관성을 유지해야 한다.
 - 똑같은 메서드를 fetch, retrieve, get 을 혼용할 경우
 - controller, manager, driver 혼용
 - DeviceManager, ProtocolController 는 근본적으로 다른가? 왜 controller를 사용하지 않은 이유는?
- 한 단어를 두 가지 목적으로 사용하지 마라. 다른 개념에 같은 단어를 사용하지 마라.
 - Add 메서드를 두 수를 더하는 메서드로 구현하다가, 집합에 값을 하나 추가하는 데 add 라고 이름 붙인다?
 - 추가하는 것은 insert, append 메서드로 이름 붙이는 게 나은 선택이다.

해법 영역, 문제 영역의 이름

- 전산 용어, 알고리즘 이름, 패턴 이름, 수학 용어 등의 사용은 무방하다.
- 기술 개념에는 기술 이름이 가장 적합하다.
 - Adaptor 패턴을 사용한다면 : AccountAdaptor
 - 큐 : JobQueue
- 문제 영역의 개념과 관련이 깊은 코드라면 문제 영역에서 이름을 가져와야 한다.
 - 통신장비에 대한 개발 : ethernet, zigbee

맥락이 있는 이름

- 클래스, 함수, 이름에 맥락을 부여한다. 모든 방법이 실패하면 마지막 수단으로 접두어를 붙인다.
 - firstName, lastName, street, houseNumber, city, state, zipcode 라는 변수가 있는데, 어떤 메소드에서 state 변수 하나만 쓴다면 state 가 주소 일부이라는 걸 알 수 있을까?
 - addr 접두어를 추가하여 addrFirstName, addrLastName, addrState 라 쓰면 맥락이 명확해진다.
- 이름에 불필요한 맥락을 추가하지 않도록 주의한다.
 - GSD(Gas Station Deluxe) 애플리케이션을 개발한다고 했을 때, 모든 클래스 이름에 GSD로 시작하는 건 불필요하다.
 - accountAddress, customerAddress 는 Address 클래스 인스턴스 이름으로 적절하나 클래스 이름으론 부적합하다.
 - 포트 주소, MAC 주소, 웹 주소를 구분해야 하면, PostalAddress, MAC, URI 이름이 의미가 더 선명하다.

■ 의미 있는 이름을 사용하면 가독성이 좋아지고 유지보수성도 보장됨

- 이름을 바꾸면 다른 개발자가 반대할까 두렵지만, 개선하려는 노력을 중단하면 안됨
- 소개한 규칙을 적용해 코드 가독성이 높아지는지 살펴보고, "자연스럽게 읽히는 코드"를 짜는데 노력해보자



주석

“나쁜 코드에 주석을 달지 마라. 새로 짜라”
- 브라이언 W. 커니핸, P.J.플라우거

■ 코드만으로 의도를 표현할 수 있어야 한다

- 주석은 나쁜 코드를 보완하지 못한다
→ 주석을 추가하는 일반적인 이유는 코드 품질이 나쁘기 때문, 주석이 필요한 코드는 새로 작성
- 코드는 변화하고 진화한다. 그러나 주석은 코드를 따라가지 못한다.
- 부정확한 주석은 현혹하고 오도한다.
- 좋은 이름, 좋은 구조로도 코드를 명확히 설명할 수 없을 때 주석을 사용
- 코드만이 정확한 정보를 제공한다.

“자신이 저지른 난장판을 주석으로 설명하려 애쓰지 말고 치우는데 시간을 보내라!”

- 회사가 정립한 구현 표준에 맞춰 법적인 이유로 사용되는 주석
- 소스 파일 첫머리에 위치
- 저작권 정보, 소유권 정보 등

```
// Copyright (C) 2003,2004,2005 by Object Mentor, Inc. All rights reserved.  
// Released under the terms of the GNU General Public License version 2 or later.
```

```
/*  
 * Copyright (c) 2017 Samsung Electronics Co., Ltd All Rights Reserved  
 * PROPRIETARY/CONFIDENTIAL This software is the confidential and proprietary  
 * information of SAMSUNG ELECTRONICS ("Confidential Information"). You shall  
 * not disclose such Confidential Information and shall use it only in  
 * accordance with the terms of the license agreement you entered into with  
 * SAMSUNG ELECTRONICS. SAMSUNG make no representations or warranties about the  
 * suitability of the software, either express or implied, including but not  
 * limited to the implied warranties of merchantability, fitness for a  
 * particular purpose, or non-infringement. SAMSUNG shall not be liable for any  
 * damages suffered by licensee as a result of using, modifying or distributing  
 * this software or its derivatives.  
*/
```

정보와 구현 의도를 알려주는 주석

35

■ 정보를 제공하는 주석

- 때로는 기본적인 정보를 주석으로 제공하면 편리함

```
// format matched kk:mm:ss EEE, MMM dd, yyyy  
Pattern timeMatcher = Pattern  
    .compile("WWd*:WWd*:WWd* WWW*", WWW* WWd*, WWd*");
```

주석은 정규표현식이 시각과 날짜를 뜻한다고 설명함.
출력 형식에 대한 정보 제공

■ 구현 의도를 설명하는 주석

- 주석은 구현을 이해하게 도와주는 선을 넘어 결정에 깔린 의도까지 설명함

```
Public void testConcurrentAddWidgets() throws Exception {  
    ...  
    // 스레드를 대량 생성하는 방법으로 어떻게든 경쟁 조건을 만들려 시도한다.  
    for (int i = 0; i < 25000; i++) {  
        WidgetBuilder Thread widgetBuilder Thread  
            = new WidgetBuilderThread(widgetBuilder, text, parent, failFlag);  
        Thread thread = new Thread(widgetBUilderThread);  
        thread.start();  
    }  
}
```

문제를 해결한 방식
& 의도를 설명함

의미를 명료하게 하는 주석

36

- 인수나 반환 값이 표준 라이브러리가 변경하지 못하는 코드에 속한다면 의미를 명료하게 밝히는 주석이 유용

```
public void testCompareTo() throws Exception {  
    WikiPagePath a = PathParser.parse("PageA");  
    WikiPagePath ab = PathParser.parse("PageA.PageB");  
    WikiPagePath b = PathParser.parse("PageB");  
    WikiPagePath aa = PathParser.parse("PageA.PageA");  
    WikiPagePath bb = PathParser.parse("PageB.PageB");  
    WikiPagePath ba = PathParser.parse("PageB.PageA");
```

```
    assertTrue(a.compareTo(a) == 0); // a == a  
    assertTrue(a.compareTo(b) != 0); // a != b  
    assertTrue(ab.compareTo(ab) == 0); // ab == ab  
    assertTrue(a.compareTo(b) == -1); // a < b  
    assertTrue(aa.compareTo(ab) == -1); // aa < ab  
    assertTrue(ba.compareTo(bb) == -1); // ba < bb  
    assertTrue(b.compareTo(a) == 1); // b > a  
    assertTrue(ab.compareTo(aa) == 1); // ab > aa  
}
```

Parse() 반환값 의미를 이해하기 쉽게
인수로 표현한 경우, 이런 경우 주석 필요 없음

라이브러리 사용으로 가독성이 떨어지는 경우,
CompareTo() 반환값 의미를 주석으로 표시

※ 주석이 올바른지 검증하기 쉽지 않음,
의미를 명료히 밝히는 주석이 필요한 이유지만, 만약 주석이 그릇된 정보를 준다면 주석이 위험한 이유이기도 함

- 다른 개발자에게 결과를 경고할 목적으로 사용

테스트 실행 시간이 오래 걸린다고 경고

// 여유 시간이 충분하지 않다면 실행하지 마십시오.

```
public void _testWithReallyBigFile() {  
    writeLinesToFile(100000000);  
    response.setBody(testFile);  
    response.readyToSend(this);  
    String responseString = output.toString();  
    assertSubString("Content-Length: 1000000000", responseString);  
    assertTrue(bytesSent > 1000000000);  
}
```

- 특정 테스트 케이스를 꺼야 하는 이유 설명하는 경우

```
public static SimpleDateFormat makeStandardHttpDateFormat() {  
    // SimpleDateFormat은 스레드에 안전하지 못하다.  
    // 따라서 각 인스턴스를 독립적으로 생성해야 한다.  
    SimpleDateFormat df = new SimpleDateFormat("EEE, dd MMM yyyy HH:mm:ss z");  
    df.setTimeZone(TimeZone.getTimeZone("GMT"));  
    return df;  
}
```

프로그램의 효율을 높이기 위해서
정적 초기화 함수를 만들려는 개발자에게 경고를 줌

- '앞으로 할 일' 또는 당장 구현하기 어려운 업무, 요청 사항 등을 기술
- 대다수 IDE는 TODO 주석을 전부 찾아 보여주는 기능을 제공하므로 주기적으로 점검 필요

```
// TODO-MdM 현재 필요하지 않다.  
// 체크아웃 모델을 도입하면 함수가 필요 없다.  
protected VersionInfo makeVersion() throws Exception {  
    return null;  
}
```

'앞으로 할 일'을 //TODO 주석으로 표현
현재 구현하지 않은 이유와 미래의 모습을 설명

■ 중요성을 강조하는 주석

- 자칫 대수롭지 않다고 여겨질 뭔가의 중요성을 강조하기 위한 경우

```
String listItemContent = match.group(3).trim();  
// 여기서 trim은 정말 중요하다. trim 함수는 문자열에서 시작 공백을 제거한다.  
// 문자열에 시작 공백이 있으면 다른 문자열로 인식되기 때문이다.  
new ListItemWidget(this, listItemContent, this.level + 1);  
return buildList(text.substring(match.end()));
```

나쁜 주석 - 주절거리는 주석

■ 주절거리는 주석

- 주석을 달기로 결정했으면 충분한 시간을 들여 최고의 주석을 달도록 노력한다.

```
public void loadProperties() {  
    try {  
        String propertiesPath = propertiesLocation + "/" + PROPERTIES_FILE;  
        FileInputStream propertiesStream = new FileInputStream(propertiesPath);  
        loadedProperties.load(propertiesStream);  
    } catch (IOException e) {  
        // 속성 파일이 없다면 기본 값을 모두 메모리로 읽어 들였다는 의미다.  
    }  
}
```

읽는 사람 입장에서는 기본값을 어디서 누가 읽어 들인다는 것
인지 이해가 어려움
→ 이해가 안되어 다른 모듈까지 뒤져야 하는 주석
→ 명확한 주석 작성 필요

나쁜 주석 - 중복되는 주석, 정확하지 않은 주석

41

- 코드랑 내용이 중복되는 주석, 오해할 여지가 있는 주석
 - 코드의 내용을 그대로 중복한 주석
 - 코드만 지저분해지게 하고 기록의 역할을 제대로 이행하지 못한다.
- 정확하지 않은 주석
 - 잘못된 정보로 인해 오해의 여지를 만든다.

// closed가 true일 때 반환되는 유틸리티 메서드다
// 타임아웃에 도달하면 예외를 던진다.

```
public synchronized void waitForClose(final long timeoutMillis) throws Exception {  
    if (!closed) {  
        wait(timeoutMillis);  
        if (!closed)  
            throw new Exception("ResponseSender could not be closed");  
    }  
}
```

[중복] 코드만 읽어봐도 충분히 주석에 있는 내용을 알 수 있음.
[오해] closed가 true가 되면 함수를 바로 반환하는 것이 아니라,
timeout시간 동안 기다린 후 그래도 true면 반환함

나쁜 주석 – 의무적으로 다는 있으나 마나 한 주석

■ 의무적으로 다는 주석

- 모든 함수에 Javadocs를 달거나 모든 변수에 주석을 작성하는 경우
- 비공개 코드에 대한 Javadocs

■ 있으나 마나 한 주석

- 너무 당연한 사실을 언급하며 새로운 정보를 제공하지 못하는 주석

```
/**
 *
 * @param title CD 제목
 * @param author CD 저자
 * @param tracks CD 트랙 숫자
 * @param durationInMinutes CD 길이(단위: 분)
 */
public void addCD(String title, String author, int tracks,
                  int durationInMinutes) {
    CD cd = new CD();
    cd.title = title;
    cd.author = author;
    cd.tracks = tracks;
    cd.duration = duration;
    cdList.add(cd);
}
```

```
/** 기본 생성자 */
protected AnnualDateRule() {
}

/** 월 중 일자 */
private int dayOfMonth;

/**
 * 월 중 일자를 반환한다.
 *
 * @return 월 중 일자
 */
public int getDayOfMonth() {
    return dayOfMonth;
}
```

■ 닫는 괄호에 다는 주석

- 함수의 길이가 길 경우 닫는 괄호에 주석을 다는 경우가 많음
- 함수의 길이를 짧게 Refactoring하면 닫는 괄호에 주석이 필요 없음

```
while ((line = in.readLine()) != null) {  
    ...  
    String words[] = line.split("WWW");  
    wordCount += words.length;  
} //while  
System.out.println("wordCount = " + wordCount);  
} // try  
catch (IOException e) {  
    System.err.println("Error:" + e.getMessage());  
} //catch  
} //main
```


나쁜 주석 - 코드의 History 주석

44

■ 이력을 기록하는 주석, 저자를 표시하는 주석

- 코드의 History 정보는 형상 관리 시스템에서 제공하는 기능 활용

```
// #####  
// Change log:  
// 2016-06-14 (John Smith) Change method rebuildProductList to fix bug #275  
// 2015-11-07 (Bob Jones) Extracted four methods to new class ProductListSorter  
// 2015-09-23 (Ninja Dev) Fixed the most stupid bug ever in a very smart way  
// #####
```



나쁜 주석 - 함수나 변수로 표현할 수 있는 주석

45

■ 함수나 변수로 표현할 수 있는 주석

- 주석이 필요하지 않도록 코드를 개선하는 편이 더 좋음

```
static void WriteCircleMetrics(BinaryWriter writer,
float radius)
{
    // circle area
    writer.Write((float)(Pi * (decimal)Math.Pow(radius, 2)));

    // circle circumference
    writer.Write((float)(Pi * (decimal)(radius * 2)));
}
```

한줄로 작성된 코드가 이해하기 어려워
주석을 추가함.
그것보다 코드 개선으로 가독성을 높여
주석을 사용하지 않는게 더 좋음

```
public static float CalculateCircleArea(float radius)
{
    return (float)(Pi * (decimal)Math.Pow(radius, 2));
}
public static float CalculateCircleCircumference(float radius)
{
    return (float)(Pi * (decimal)(radius * 2));
}
static void WriteCircleMetrics(BinaryWriter writer,
float radius)
{
    writer.Write(CalculateCircleArea(radius));
    writer.Write(CalculateCircleCircumference(radius));
}
```

나쁜 주석 - 전역 정보를 제공하는 주석

46

■ 전역 정보를 제공하는 주석

- 주석을 달아야 한다면 근처에 있는 코드만 기술
- 시스템 전반적인 정보를 기술하지 마라.

```
/**
 * 적합성 테스트가 동작하는 포트: 기본값은 <b>8082</b>.
 *
 * @param fitnessPort
 */
public void setFitnessPort(int fitnessPort)
{
    this.fitnessPort = fitnessPort;
}
```

주석에서 기본 포트 정보를 제공

➔ 함수 자체는 포트 기본값을 전혀 통제 못함.
즉, 주석은 바로 아래 함수가 아닌 다른 함수를 설명

■ 주석으로 처리한 코드

- 주석으로 처리된 코드는 지우면 안 된다고 생각하여, 쓸모 없는 코드가 쌓이게 된다.
- 형상 관리 시스템을 이용하면 이전 코드를 잃어버릴 염려는 없음

```
// This function is no longer used (John Doe, 2013-10-25):  
/*  
double calcDisplacement(double t) {  
    const double goe = 9.81; // gravity of earth  
    double d = 0.5 * goe * pow(t, 2); // calculation of distance  
    return d;  
}  
*/
```

형상관리 시스템 사용으로 필요 X

■ 위치를 표시하는 배너 주석

- 배너는 반드시 필요할 때만, 아주 드물게 사용하는 게 좋다. 남용하면 독자가 무시한다.

```
// Actions //////////////////////////////////////
```

■ 코드와 모호한 관계의 주석

- 주석과 주석이 설명하는 코드는 둘 사이 관계가 명확해야 한다.
- 주석과 코드를 읽고 이해할 수 있어야 한다.

```
/*  
* 모든 픽셀을 담을 만큼 충분한 배열로 시작한다. (여기에 필터 바이트를 더한다)  
* 그리고 헤더 정보를 위해 200바이트를 더한다.  
*/
```

```
pngBytes = new byte[((width + 1) * height * 3) + 200];
```

필터바이트에 대한 설명 부재
→ 주석과 코드를 읽어보고
무슨 소린지 이해 가능 해야 함

형식 맞추기

■ 형식을 맞춰서 코드를 작성해야 하는 이유

- 코드 형식은 의사 소통의 일환이고 전문 개발자의 일차적인 의무이기 때문
- 처음 접은 구현 스타일과 가독성 수준은 앞으로 바뀔 코드의 품질에 지대한 영향을 미침
 - 온갖 스타일을 뒤섞어 소스코드를 필요 이상으로 복잡하게 만들 수 있음
- 형상 관리 도구에서 포맷으로 인한 Diff를 방지하기 위해서도 필요
- 코드 형식을 자동으로 맞춰주는 도구 활용 권장

➔ **팀은 한 가지 규칙에 합의하고 모든 팀원은 그 규칙을 따라 일관적인 스타일을 유지!**

■ 형식을 맞추는 목적

- 코드 형식은 의사소통의 일환이며, 의사소통은 전문 개발자의 일차적인 의무
- 오늘 구현한 코드의 스타일과 가독성 수준은 유지보수의 용이성과 확장성에 지속적인 영향을 미침

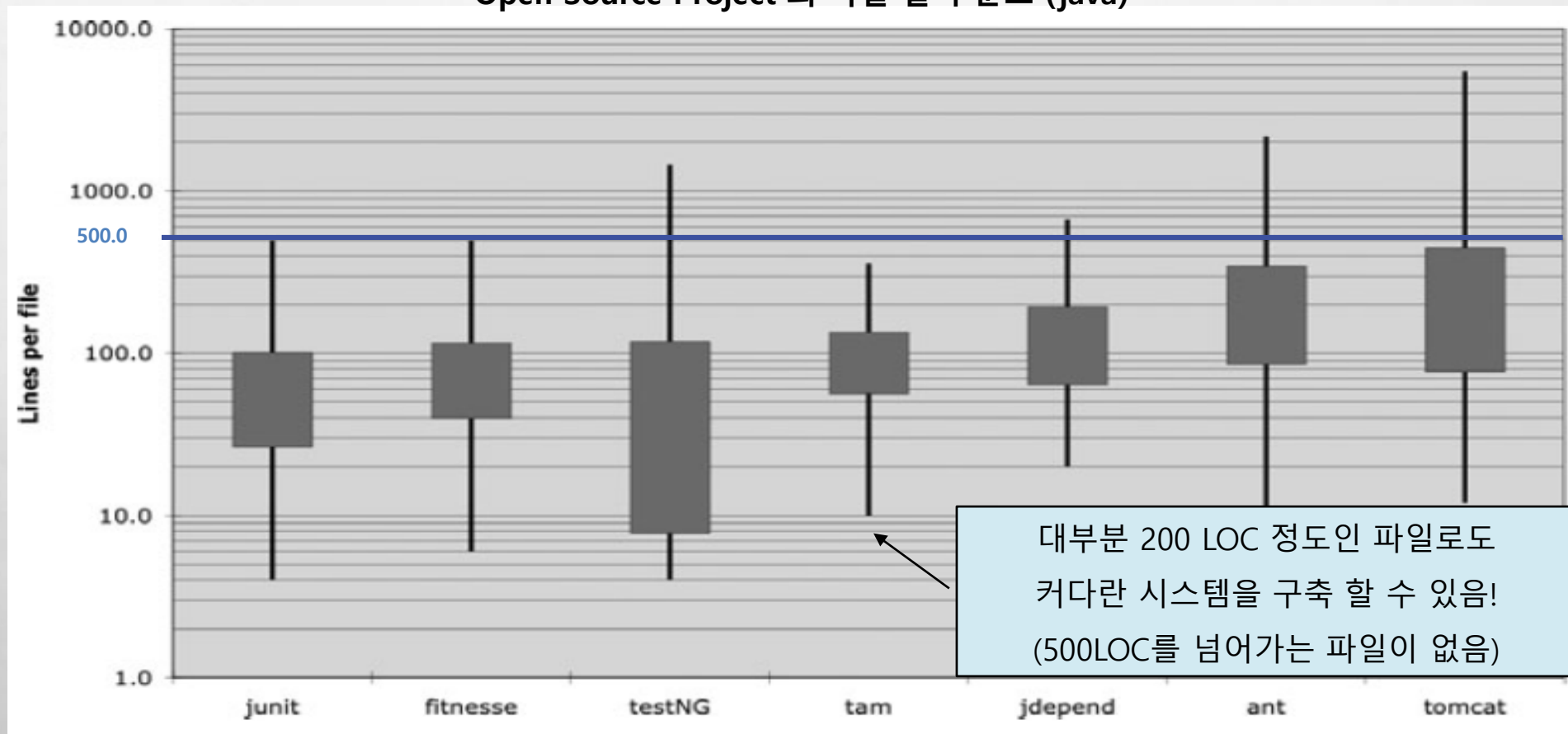
세로 형식 맞추기

51

■ 적절한 행 길이를 유지하라

- 일반적으로 큰 파일보다 작은 파일이 이해가 쉬움

Open Source Project 의 파일 길이 분포 (java)



대부분 200 LOC 정도인 파일로도
커다란 시스템을 구축 할 수 있음!
(500LOC를 넘어가는 파일이 없음)

※ 세로선 : 최대/최소 파일 길이, 박스 : 표준 편차

세로 형식 맞추기 - 신문 기사처럼 작성하라

■ High Level → Low Level 순서로 작성

- 이름은 간단하면서 설명 가능하게 지음
- 첫 부분은 고차원 개념, 알고리즘 설명
- 아래로 내려가며 의도를 세세하게 묘사
- 마지막은 가장 저차원 함수와 세부 내역

[신문 기사]

제목

표제
(기사 요약)

상세 내용
(이름, 발언,
세부 사항 등)

The sky's the limit

Franca Jamieson's company will try to marry convenient helicopter travel with 'enjoyable golf, fine dining and beautiful scenery'

By DON FRASER

In Niagara's burgeoning scene of high-end wine, golf and haute cuisine, the sky's the limit when it comes to luxury.

And Franca Jamieson literally wants to take people there. Jamieson is the owner of Niagara Heli-Golf Inc., which contracts its helicopter service through Niagara Helicopters Ltd.

She said her trademarked concept is to marry convenient helicopter travel with "enjoyable golf, fine dining and beautiful scenery."

Helicopter tour packages to wineries and other destinations are not unique in the region. Two-year-old Skyway Helicopters Inc. at the Niagara District Airport, for instance, also does tours to estate wineries such as Peller, Hillebrand, Hernder and, in the past, the Royal Niagara Golf Club.

But Jamieson says her Niagara Falls company, launched in June, offers its own twist to the genre.

"We're going to be providing optimum luxury services with these golf packages," she said. "And here, the wine and top cuisine is what we want to be known for."

Jamieson is completing her last year as a senior immigration officer with Citizenship and Immigration Canada and plans to devote her retirement energies to her new company.

It's a natural fit, says the 53-year-old Niagara Falls resident. "I've been dealing with people and every facet of a person's travelling for 35 years."

She has also been affiliated with the U.S.-based Club Corp of America — a large organization of international private golf clubs — and has played at a

number of them. She knows they spend when they're golfing."

She said the concept began about four years ago. Jamieson, a friend of Niagara Helicopters president Ruedi Hafen, approached Hafen with her business idea.

"I told him, 'This is the age of everything extreme and ultimate,'" she said. "And I believe he has the most reputable operation in southern Ontario."

"Golf seems to be very benign and sometime not even considered a sport so I thought, 'Well, let's do something extreme,'" said Jamieson.

"Let's take them up by helicopter and land them on the golf course — that would be really prestigious."

On the day of the interview, the heliport site itself was abuzz as one rainbow copter after another swooped over the asphalt. The five Bell 407s in Hafen's fleet and his pilots will be contracted out by the new company.

Niagara Helicopters is also in the midst of renovating a new facility is under construction that will house a restaurant, cafe, store, offices and tie-in booth.

Using the Niagara Helicopters heliport on Victoria Avenue as a base, Niagara Heli-Golf will offer tours from Toronto Island and Niagara to premier courses in the region and elsewhere in Ontario.

Right now, that roster includes the Royal Niagara Golf Club, Peninsula Lakes, Legends on the Niagara and Hunters Pointe Golf Course.

Arrangements have been made with the clubs to ensure clients will be the only ones arriving that day by helicopter.

It's a new venture she came to us with," said Kevin Poole, golf director at Welland's Hunters Pointe. "The concept is certainly an interesting one."

Heli-Golf, he said, seems designed to attract the kind of upscale market the club was seeking.

Jamieson's company will also co-ordinate flexible helicopter pickup spots and organize designer packages based on the desires of the client.

Right now, she's advertising in corporate and golf magazines and hopes to extend that to wine and food publications.

"We've had a lot of inquiries, I've been doing a lot of quotes," she said. "Her one-day and two-day packages are also affiliated with Vineland Estates and Peninsula Ridge wineries, where tour clients will dine. Helicopter golf tours to Muskoka are also being offered."

"Since it's an upscale tour, they're going to have the best," she said. "It's a complete dinner and they have their own private dining room."

For example, the one-day Niagara tour to the elite Glen Abbey Golf Course in Oakville begins with clients being picked up by limo at their hotels and taken to the heliport.



Niagara Heli-Golf Inc. will take groups on trips to high-end Niagara golf courses, restaurants and wineries. Franca Jamieson of Niagara Heli-Golf and Ruedi Hafen of Niagara Helicopters Ltd. check out Royal Niagara Golf Club.

MIFF PHOTO BY LEONARD LANGE

with," said Kevin Poole, golf director at Welland's Hunters Pointe. "The concept is certainly an interesting one."

Heli-Golf, he said, seems designed to attract the kind of upscale market the club was seeking.

Jamieson's company will also co-ordinate flexible helicopter pickup spots and organize designer packages based on the desires of the client.

Right now, she's advertising in corporate and golf magazines and hopes to extend that to wine and food publications.

"We've had a lot of inquiries, I've been doing a lot of quotes," she said. "Her one-day and two-day packages are also affiliated with Vineland Estates and Peninsula Ridge wineries, where tour clients will dine. Helicopter golf tours to Muskoka are also being offered."

"Since it's an upscale tour, they're going to have the best," she said. "It's a complete dinner and they have their own private dining room."

For example, the one-day Niagara tour to the elite Glen Abbey Golf Course in Oakville begins with clients being picked up by limo at their hotels and taken to the heliport.

From the heliport, they're flown to the golf course, where they use rented equipment. After an 18-hole round and lunch at the Glen Abbey Bistro Grill, they return to Niagara by limo, via a wine country route.

That's followed by a gourmet dinner at either Peninsula Ridge or Vineland Estates wineries, complete with brandy and cigars.

The packages for four-to-six people or two parties of 12 may not suit the budget di-tripper.

The "Glen Abbey Ultimate Day" tour costs \$1,000 US (\$1,300 Cdn) per person. Jamieson said she nets less than 15 per cent commission, given her cost per person are \$1,000.

Her clientele is currently a mix of mostly Canadians and Americans. "We're going to be global. We want to bring them to Canada's premier golf destinations," she said.

For baby boomers who are retiring now, or women who don't know what kind of present to give their avid-golfing husband, this is the perfect gift."

And the concept is not just for the obsessive male golfers or the corporate jet set.

On deck is a "fantasy camp for women," Jamieson said. It will likely include lessons from top instructors, al-

though details haven't been fully worked out.

"These are women who have never had the chance to golf at these premium golf courses for one reason or another."

"Women enjoy travelling in groups — they would love to do this," said Jamieson.

While her company's summer operations will only run from June to August, Jamieson plans to begin international wine-and-food helicopter tours in the winter.

Right now, there are no revenue projections, she said. "Really, there are no comparisons for me. I want to start modestly and see where we end up."

Skyway Helicopters marketing manager, Nancy Quinn welcomed any competition in the helicopter tour market.

"I think it's a good thing," said Quinn. "There's an absolutely vast visitorship to Niagara."

Establishing a niche market is important, especially when you look at the high-end consumer," she said.

"And I really believe there's enough for everybody here."

For more information, contact Niagara Heli-Golf at: www.niagraheli-golf.ca

세로 형식 맞추기 - 신문 기사처럼 작성하라

53

```
private void includeSetupAndTeardownPages() throws Exception {  
    includeSetupPages();  
    includePageContent();  
    includeTeardownPages();  
}
```

1st
Level

```
private void includeSetupPages() throws Exception{  
    if (isSuite) includeSuiteSetupPage();  
    includeSetupPage();  
}
```

2nd
Level

```
private void includeSuiteSetupPage() throws Exception {  
    include(SuiteResponder.SUITE_SETUP_NAME, "-setup");  
}  
  
private void includeSetupPage() throws Exception {  
    include("SetUp", "-setup");  
}
```

3rd
Level

```
private void includePageContent() throws Exception {  
    newPageContent.append(pageData.getContent());  
}
```

```
private void includeTeardownPages() throws Exception{  
    includeTeardownPage();  
    if (isSuite) includeSuiteTeardownPage();  
}
```

2nd
Level

```
private void includeTeardownPage() throws Exception {  
    include("TearDown", "-teardown");  
}
```

```
private void includeSuiteTeardownPage() throws Exception {  
    include(SuiteResponder.SUITE_TEARDOWN_NAME, "-teardown");  
}
```

3rd
Level

```
private void include(String pageName, String arg) throws Exception{  
    WikiPage inheritedPage = findInheritedPage(pageName);  
    ... ..  
}
```

4th
Level

세로 형식 맞추기 - 개념은 빈 행(Blank Line)으로 분리하라

54

- 각 행은 수식이나 절을 나타내고, 일련의 행 묶음은 완결된 생각 하나를 표현
 - 생각 사이에는 빈 행을 넣어 분리

```
package fitnessse.wikitext.widgets;
import java.util.regex.*;
public class BoldWidget extends ParentWidget {
    public static final String REGEXP = ""'+?''";
    private static final Pattern pattern = Pattern.compile("'''+(.*?)'''",
        Pattern.MULTILINE + Pattern.DOTALL);
    public BoldWidget(ParentWidget parent, String text) {
        super(parent);
        Matcher match = pattern.matcher(text);
        match.find();
        addChildWidgets(match.group(1));
    }
    public String render() throws Exception {
        StringBuffer html = new StringBuffer("<b>");
        html.append(childHtml()).append("</b>");
        return html.toString();
    }
}
```

전체가 한덩어리로 보여
코드의 가독성이 현저하게 떨어짐

```
package fitnessse.wikitext.widgets;

import java.util.regex.*;

public class BoldWidget extends ParentWidget {
    public static final String REGEXP = ""'+?''";
    private static final Pattern pattern = Pattern.compile("'''+(.*?)'''",
        Pattern.MULTILINE + Pattern.DOTALL
    );

    public BoldWidget(ParentWidget parent, String text) {
        super(parent);
        Matcher match = pattern.matcher(text);
        match.find();
        addChildWidgets(match.group(1));
    }

    public String render() throws Exception {
        StringBuffer html = new StringBuffer("<b>");
        html.append(childHtml()).append("</b>");
        return html.toString();
    }
}
```


세로 형식 맞추기 - 서로 밀접한 행은 세로로 가까이 놓기 (1/4)

55

- 같은 파일에 속할 정도로 밀접한 두 개념은 세로 거리로 연관성을 표현

```
public class ReporterConfig {  
    /**  
     * The class name of the reporter listener  
     */  
    private String m_className;  
    /**  
     * The properties of the reporter listener  
     */  
    private List<Property> m_properties  
        = new ArrayList<Property>();  
  
    public void addProperty(Property property) {  
        m_properties.add(property);  
    }  
}
```

```
public class ReporterConfig {  
    private String m_className;  
    private List<Property> m_properties  
        = new ArrayList<Property>();  
  
    public void addProperty(Property property) {  
        m_properties.add(property);  
    }  
}
```

세로 형식 맞추기 - 서로 밀접한 행은 세로로 가까이 놓기 (2/4)

56

- 비슷한 동작을 수행하는 함수들을 가까이 배치(밀접한 코드행은 세로로 가까이 놓음)

```
public class Assert {  
    static public void assertTrue(String message, boolean condition) { ... }  
  
    static public void assertTrue(boolean condition) { ... }  
  
    static public void assertFalse(String message, boolean condition) { ... }  
  
    static public void assertFalse(boolean condition) { ... }  
}
```


세로 형식 맞추기 - 서로 밀접한 행은 세로로 가까이 놓기 (3/4)

57

- 변수는 사용하는 위치에 최대한 가까이 선언
 - 각 함수의 처음 부분에서 지역 변수를 선언
 - 루프를 제어하는 변수는 가급적 루프 문 내부에 선언

세로 형식 맞추기 - 서로 밀접한 행은 세로로 가까이 놓기 (4/4)

58

- 호출 관계가 있는 종속 함수는 세로로 가까이 배치

```
public class WikiPageResponder implements SecureResponder {  
    protected WikiPage page;  
    protected PageData pageData;  
    protected String pageTitle;  
    protected Request request;  
    protected PageCrawler crawler;
```

c/c++ 예외
(Java: 인스턴스 변수는 클래스 맨 처음에 선언)

```
    public Response makeResponse(FitNesseContext context, Request request) throws Exception {  
        String pageName = getPageNameOrDefault(request, "FrontPage");  
        loadPage(pageName, context);  
        ...  
    }
```

호출 관계에 있는 함수끼리 가까이 배치

```
    private String getPageNameOrDefault(Request request, String defaultPageName) {  
        ...  
    }
```

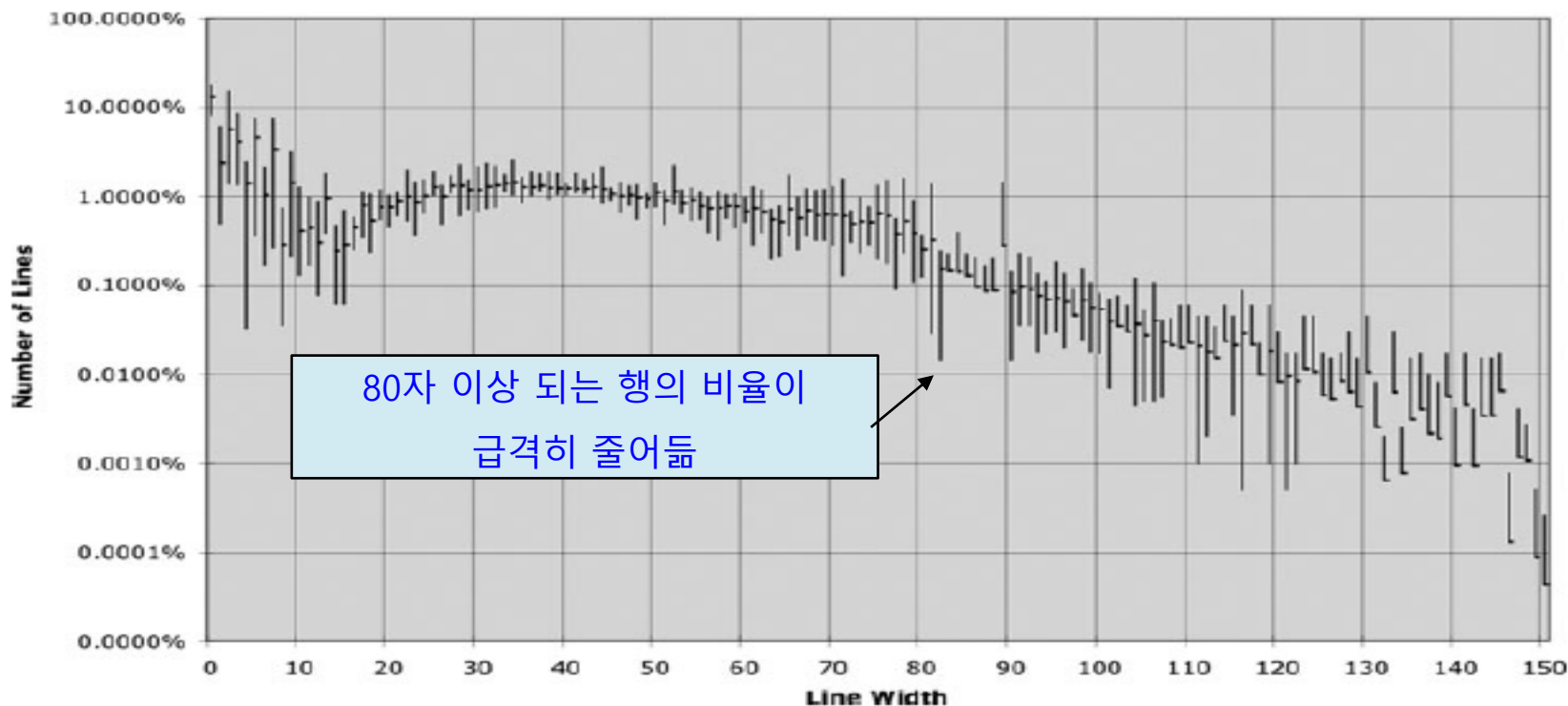
```
    protected void loadPage(String resource, FitNesseContext context) throws Exception {  
        ...  
    }
```

호출하는 순서대로 함수를 정의
→ 찾기 쉽고 가독성 높음
(호출하는 함수 먼저 배치)

■ 적절한 가로 길이를 유지하라

- 오른쪽으로 스크롤 할 필요 없도록 한 화면에 표현 될 수 있는 글자수 유지 (80~120 정도)

Open Source Project 의 가로 길이 분포 (java)



■ 가로 공백(Blank) 사용해 밀접한 개념과 느슨한 개념을 표현

```
private void measureLine①(String line) {  
    lineCount+②;  
    int lineSize = line.length();  
    totalChars += lineSize;  
    lineWidthHistogram.addLine(lineSize, lineCount);  
    recordWidestLine(lineSize);  
}
```

① 함수와 인수는 밀접한 관계

→ 함수 이름과 이어지는 괄호 사이에는 공백 X

②, ③ 두 요소가 개념적으로 분리가 필요할 경우
(할당 연산자, 함수 parameter 등)

→ 앞뒤에 공백 삽입

```
public class Quadratic {  
    public static double root1(double a, double b, double c) {  
        double determinant = determinant(a, b, c);  
        return (-b + Math.sqrt(determinant)) / (2*a);  
    }  
  
    public static double root2(int a, int b, int c) {  
        double determinant = determinant(a, b, c);  
        return (-b - Math.sqrt(determinant)) / (2*a);  
    }  
  
    private static double determinant(double a, double b, double c) {  
        return b*b - 4*a*c;  
    }  
}
```

연산자 우선 순위를 강조하기 위해서 공백 사용
→ 우선순위 높은 연산 (곱하기) : 공백 X

- 가로 정렬은 코드의 진짜 의도가 가려지게 할 수 있음 → 별로 유용하지 않음

```
public class FitNesseExpediter implements ResponseSender {
    private Socket          socket;
    private InputStream      input;
    private OutputStream    output;
    private Request          request;
    private Response         response;
    private FitNesseContext context;
    protected long          requestParsingTimeLimit;
    private long             requestProgress;
    private long             requestParsingDeadline;
    private boolean          hasError;

    public FitNesseExpediter(Socket s,
                               FitNesseContext context)
    {
        this.context = context;
        socket = s;
        input = s.getInputStream();
        output = s.getOutputStream();
        requestParsingTimeLimit = 10000;
    }
}
```

```
public class FitNesseExpediter implements ResponseSender {
    private Socket socket;
    private InputStream input;
    private OutputStream output;
    private Request request;
    private Response response;
    private FitNesseContext context;
    protected long requestParsingTimeLimit;
    private long requestProgress;
    private long requestParsingDeadline;
    private boolean hasError;

    public FitNesseExpediter(Socket s, FitNesseContext context)
    {
        this.context = context;
        socket = s;
        input = s.getInputStream();
        output = s.getOutputStream();
        requestParsingTimeLimit = 10000;
    }
}
```

선언부를 읽다 보면 변수 유형은 무시하고 이름만 읽게 됨
할당 연산자는 보이지 않고 오른쪽 피연산자에 눈이 감
→ 코드가 엉뚱한 부분을 강조

- 들여쓰기를 통하여 코드가 속하는 범위를 시각적으로 쉽게 파악 가능
 - 파일의 구조가 한눈에 들어올 수 있음 (가독성의 중요한 요소임)

```
public class FitNesseServer implements SocketServer { private
FitNesseContext context; public FitNesseServer(FitNesseContext
context) { this.context = context; } public void serve(Socket s)
{ serve(s, 10000); } public void serve(Socket s, long
requestTimeout) { try { FitNesseExpediter sender = new
FitNesseExpediter(s, context);
sender.setRequestParsingTimeLimit(requestTimeout);
sender.start(); }
catch(Exception e) { e.printStackTrace(); } } }
```

들여쓰기가 없다면 코드 읽기란 거의 불가능

```
public class FitNesseServer implements SocketServer {
    private FitNesseContext context;

    public FitNesseServer(FitNesseContext context) {
        this.context = context;
    }

    public void serve(Socket s) {
        serve(s, 10000);
    }

    public void serve(Socket s, long requestTimeout) {
        try {
            FitNesseExpediter sender = new FitNesseExpediter(s, context);
            sender.setRequestParsingTimeLimit(requestTimeout);
            sender.start();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

- 간단한 if/ while 문, 짧은 함수에서도 들여쓰기 사용

```
public class CommentWidget extends TextWidget {  
    public static final String REGEXP = "^#[^\\r\\n]*(?:(?:\\r\\n)|\\n|\\r)?";  
    public CommentWidget(ParentWidget parent, String text){super(parent, text);}  
    public String render() throws Exception {return ""; }  
}
```

```
public class CommentWidget extends TextWidget {  
    public static final String REGEXP = "^#[^\\r\\n]*(?:(?:\\r\\n)|\\n|\\r)?";  
  
    public CommentWidget(ParentWidget parent, String text) {  
        super(parent, text);  
    }  
  
    public String render() throws Exception {  
        return "";  
    }  
}
```

- 빈 while/for 문 에는 세미콜론(;)를 새 행에다 제대로 들여 써서 넣어 준다

```
while(dis.read(buf, 0, readBufferSize) != -1) ;
```

```
while(dis.read(buf, 0, readBufferSize) != -1)  
;
```

While문에서 하는 것이 없다는 의미로 가독성을 위해
세미콜론(;)을 아래에 표기하는 것이 더 좋다

- 팀에 속하여 있다면 자신이 선호해야 할 규칙은 팀 규칙
- 팀은 한 가지 규칙에 합의 해야하며, 모든 팀원은 그 규칙을 따라야함
 - 소프트웨어가 일관적인 스타일로 작성됨
(개개인이 맘대로 짜는 코드는 피해야함)
- 좋은 소프트웨어 시스템은 읽기 쉬운 문서로 이뤄지고,
읽기 쉬운 문서는 스타일이 일관적이고 매끄러워야 한다.

함수

■ 길이가 길고, 중복된 코드, 괴상한 문자열, 낯설고 모호한 자료유형과 API 사용

```
1780 sal_Bool BasicFrame::QueryFileName(String& rName, FileType nFileType, sal_Bool bSave )
1781 {
1782     NewFileDialog aDlg( this, bSave ? WinBits( WB_SAVEAS ) :
1783     WinBits( WB_OPEN ) );
1784     aDlg.SetText( String( SttResId( bSave ? IDS_SAVEDLG : IDS_LOADADDLG ) ) );
1785
1786     if ( nFileType & FT_RESULT_FILE )
1787     {
1788         aDlg.SetDefaultExt( String( SttResId( IDS_RESFILE ) ) );
1789         aDlg.AddFilter( String( SttResId( IDS_RESFILTER ) ),
1790             String( SttResId( IDS_RESFILE ) ) );
1791         aDlg.AddFilter( String( SttResId( IDS_TXTFILTER ) ),
1792             String( SttResId( IDS_TXTFILE ) ) );
1793         aDlg.SetCurFilter( SttResId( IDS_RESFILTER ) );
1794     }
1795
1796     if ( nFileType & FT_BASIC_SOURCE )
1797     {
1798         aDlg.SetDefaultExt( String( SttResId( IDS_NONAMEFILE ) ) );
1799         aDlg.AddFilter( String( SttResId( IDS_BASFILTER ) ),
1800             String( SttResId( IDS_NONAMEFILE ) ) );
1801         aDlg.AddFilter( String( SttResId( IDS_INCFILTER ) ),
1802             String( SttResId( IDS_INCFILE ) ) );
1803         aDlg.SetCurFilter( SttResId( IDS_BASFILTER ) );
1804     }
1805
1806     if ( nFileType & FT_BASIC_LIBRARY )
1807     {
1808         aDlg.SetDefaultExt( String( SttResId( IDS_LIBFILE ) ) );
1809         aDlg.AddFilter( String( SttResId( IDS_LIBFILTER ) ),
1810             String( SttResId( IDS_LIBFILE ) ) );
1811         aDlg.SetCurFilter( SttResId( IDS_LIBFILTER ) );
1812     }
```

```
1813
1814     Config aConf( Config::GetConfigName( Config::GetDefDirectory(),
1815         CUniString( "testtool" ) ));
1816     aConf.SetGroup( "Misc" );
1817     ByteString aCurrentProfile = aConf.ReadKey( "CurrentProfile", "Path" );
1818     aConf.SetGroup( aCurrentProfile );
1819     ByteString aFilter( aConf.ReadKey( "LastFilterName" ) );
1820     If ( aFilter.Len() )
1821         aDlg.SetCurFilter( String( aFilter, RTL_TEXTENCODING_UTF8 ) );
1822     else
1823         aDlg.SetCurFilter( String( SttResId( IDS_BASFILTER ) ) );
1824
1825     aDlg.FilterSelect(); // Selects the last used path
1826     // if ( bSave )
1827     if ( rName.Len() > 0 )
1828         aDlg.SetPath( rName );
1829
1830     if( aDlg.Execute() )
1831     {
1832         rName = aDlg.GetPath();
1833         /* rExtension = aDlg.GetCurrentFilter();
1834         var i:integer;
1835         for ( i = 0 ; i < aDlg.GetFilterCount() ; i++ )
1836             if ( rExtension == aDlg.GetFilterName( i ) )
1837                 rExtension = aDlg.GetFilterType( i );
1838         */
1839         return sal_True;
1840     } else return sal_False;
1841 }
```

※ Apache's OpenOffice 3.4.1.

- 함수를 만드는 가장 중요한 규칙은 '작게!'
 - *"함수는 100줄을 넘어서는 안 된다. 아니 20줄도 길다."*
 - 함수를 작게 만들어야 읽고 이해하기가 쉬워짐
- if/else, while문 등에 들어가는 블록은 한 줄이어야 함
 - 대부분 함수를 호출
 - 중첩 구조가 생길 만큼 함수가 커져서는 안됨
 - 각 함수의 들여쓰기 수준은 2단을 넘어서는 안됨, 그래야 함수 읽고 이해하기 쉬워짐

작게 만들어라

```
static void processEventsDueNow(Time * time, ScheduledLightEvent * event){
    Day today = time->dayOfWeek;
    int minuteOfDay = time->minuteOfDay;

    if (event->id != UNUSED)
    {
        Day day = event->day;
        if ( (day == EVERYDAY) || (day == today) || (day == WEEKEND &&
            (today == SATURDAY || today == SUNDAY)) ||
            (day == WEEKDAY && (today >= MONDAY
                && today <= FRIDAY)))
        {
            /* it's the right day */
            if (minuteOfDay == event->minuteOfDay + event->randomMinutes) {
                if (event->event == TURN_ON)
                    LightController_TurnOn(event->id);
                else if (event->event == TURN_OFF)
                    LightController_TurnOff(event->id);

                if (event->randomize == RANDOM_ON)
                    event->randomMinutes = RandomMinute_Get();
                else
                    event->randomMinutes = 0;
            }
        }
    }
}
```

```
static void processEventsDueNow(Time * time, ScheduledLightEvent * event){
    if (event->id != UNUSED)
    {
        if (isEventDueNow(time, event))
        {
            operateLight(event);
            resetRandomize(event);
        }
    }
}
```

한 가지만 해라

70

■ “함수는 한 가지를 해야 한다. 그 한 가지를 잘 해야 한다. 그 한 가지만을 해야 한다.”

- 지정된 함수 이름 아래에서 추상화 수준이 하나면 한 가지 작업만 수행
- 의미 있는 이름으로 다른 함수가 추출 가능하다면, 함수가 여러 작업을 한다고 볼 수 있다

■ 함수 당 추상화 수준은 하나로

- 함수 내 모든 문장의 추상화 수준이 동일해야 함수가 한 가지 작업만 한다.

```
static void processEventsDueNow(Time * time,
ScheduledLightEvent * event){
    if (event->id != UNUSED)
    {
        if (isEventDueNow(time, event))
        {
            operateLight(event);
            resetRandomize(event);
        }
    }
}
```

다른 부분과 추상화 레벨이 다르다.

```
static void processEventsDueNow(Time * time,
ScheduledLightEvent * event){
    if (!isInUse(event))
        return;
    if (isEventDueNow(time, event)) {
        operateLight(event);
        resetRandomize(event);
    }
}
```

■ 위에서 아래로 코드 읽기: 내려가기 규칙

- 코드는 위에서 아래로 이야기처럼 읽혀야 좋다.
- 한 함수 다음에는 추상화 수준이 한 단계 낮은 함수가 온다.

■ 다형성(polymorphism)을 이용해라

- Switch문은 작고, 한 가지 작업만 하도록 만들기 어렵다.
- 다형성을 이용하여 추상 팩토리에 숨겨 다형성 객체를 생성하는 코드에서만 사용한다.

```
Money calculatePay(Employee e)
{
    switch (e.type) {
        case COMMISSIONED:
            return calculateCommissionedPay(e);
        case HOURLY:
            return calculateHourlyPay(e);
        case SALARIED:
            return calculateSalariedPay(e);
        default:
            return InvalidEmployeeType(e);
    }
}
```

문제점

calculatePay()이외에도 isPayDay(), deliverPay() 등 함수마다 switch문이 중복 사용 하게 되고, 새로운 유형의 직원 추가시 급여 조건이 추가되어 여러 코드 변경이 필요하게 됨

※ 추상 팩토리(Abstract factory) : 클라이언트에서 interface를 통해 관련성을 갖는 객체들의 집합을 생성하거나, 서로 독립적인 객체들의 집합을 생성할 수 있음

■ 다형성(polymorphism)을 이용해라

- Switch문은 작고, 한 가지 작업만 하도록 만들기 어렵다.
- 다형성을 이용하여 추상 팩토리에 숨겨 다형성 객체를 생성하는 코드에서만 사용한다.

```
class Employee {  
public:  
    virtual ~Employee() = default;  
    virtual bool isPayday();  
    virtual Money calculatePay();  
    virtual void deliverPay(Money pay);  
}
```

```
Employee aEmployee = ceateEmp(EmployeeRecord r);  
shared_ptr<Employee> createEmp(EmployeeRecord r) {  
    switch (r.type) {  
        case COMMISSIONED:  
            return make_shared<CommissionedEmployee>(r);  
        case HOURLY:  
            return make_shared<HourlyEmployee>(r);  
        case SALARIED:  
            return make_shared<SalariedEmployee>(r);  
        default:  
            return make_shared<InvalidEmployeeType>(r);  
    }  
}
```

Switch문은 Factory 안에만 존재

- 상속 관계로 숨겨져 다른 코드에 노출되지 않게 한다
- 유지보수성을 높일 수 있음

■ 이상적인 인수의 개수는 0개(무항)

- 인수는 코드 이해에 방해가 되며 테스트를 어렵게 함.
- 출력인수: 함수의 반환 값이 아닌 입력 인수로 결과를 받는 경우
(*출력인수는 입력 인수보다 이해하기 어려움.)

■ 함수에 인수를 1개 전달하는 경우

- 인수에 질문을 던지는 경우

```
bool isAlpha('A');
```

- 인수를 뭔가로 변환해서 결과를 반환하는 경우

```
std::ifstream open("MyFile.txt");
```

- 이벤트 함수일 경우

- 이벤트 함수 : 입력인수 1개, 출력인수 없는 함수
- 이벤트라는 사실이 코드에 명확하게 드러나야 함

➔ 위 3가지 경우가 아니라면 단항 함수를 가급적 피하라

■ 플래그 인수

- bool 값을 넘긴다는 것은 함수가 여러 가지 일을 처리한다고 공표하는 것
- 플래그 인수를 사용하지 말고 대신 함수를 나뉘라

```
render(const bool isSuite)
```



```
renderForSuite();  
renderForSingleTest();
```

■ 이항 함수

- 다음과 같은 경우는 이항 함수가 적절하지만, 가능하면 단항으로 바뀌어야 한다

```
void set_Point (int _x, int _y);
```

■ 삼항 함수

- 이항 함수보다 더 이해하기 어려우므로 신중히 고려해야 함

■ 인수 객체 활용

- 인수가 2~3개 필요하다면, 객체를 생성하여 인수를 줄인다.

```
//삼항 함수예  
Circle makeCircle(double x, double y, double radius);
```

```
//인수 객체 활용 예  
Circle makeCircle(Point center, double radius);
```

■ 동사와 키워드

- 단항 함수는 함수와 인수가 동사/명사 쌍을 이뤄야 함

```
write(name);
```

- 함수 이름에 인수 이름을 추가하면 순서를 기억할 필요가 없음

```
assertExpectedEqualsActual(expected, actual);
```

부수 효과를 일으키지 마라(1/2)

76

- 한 함수에서는 딱 한가지만 수행해야 함
 - 아래의 함수에서 **Session.initialize();** 는 함수명과 맞지 않는 부수 효과

```
class UserValidator {  
public:  
    bool checkPassword(const string& userName, const string& password) {  
        User user = UserGateway.findByName(userName);  
        if (user != User.NULL) {  
            string codedPhrase = user.getPhraseEncodedByPassword();  
            string phrase = cryptographer.decrypt(codedPhrase, password);  
            if (phrase == "Valid Password") {  
                Session.initialize();  
                return true;  
            }  
        }  
        return false;  
    }  
private:  
    Cryptographer cryptographer;  
}
```

checkPassword() 암호를 확인하는 함수인데 세션 초기화가 부수적으로 일어남. Session을 초기화해도 괜찮은 경우만 호출되어야 함. 만약 함수 분리가 어렵다면, **checkPasswordAndInitializeSession**이라는 이름이 훨씬 좋다.

부수 효과를 일으키지 마라(2/2)

77

■ 출력 인수 : 함수의 반환 값이 아닌 입력 인수로 결과를 받는 경우

- 출력 인수로 쓰일 경우 인수 값이 변하는데, 사용자가 이를 인지하기 어려움

```
//출력 인수 사용 예  
makeCommand(commandName, argumentList, result)
```

함수 선언부를 봐야 result가 출력 인수라는 사실 확인 가능

```
makeCommand(const std::string& name,  
            const std::vector<std::string>& arguments, std::string result) {  
    ...  
}
```



```
//멤버 변수 사용 예  
result.makeCommand();
```

객체지향프로그래밍 전에는 출력 인수가 불가피한 경우도 있다. 그러나 객체지향에서는 멤버 변수를 사용한 예와 같이 호출하는게 좋다.
여러 개의 데이터 반환을 위해 많이 사용
=> **C++: std::tuple or a std::pair**를 사용

명령과 조회를 분리하라

78

■ 한 함수가 두가지를 하면 혼란을 초래함

- 명령과 조회가 분리되지 않은 함수가 호출되는 부분을 볼 때, 의미가 모호함

```
//명령과 조회 둘 다 수행하는 예
bool set(string attribute, string value);

...
if(set("username", "unclebob"))...
```

조회와 명령 두가지 기능을 수행함.

[조회]: 설정이 성공하면 true를, 실패하면 false를 반환

[명령]: attribute를 value로 설정

아래와 같은 이상한 코드가 나타남, 의미 파악이 어려움

해석① (set을 동사로 해석) – 원래 의도

"username을 unclebob으로 설정 하고, 만약 성공했다면..."

해석② (set을 형용사로 해석)

"username이 unclebob으로 설정되어 있다면..."

```
//명령과 조회를 분리한 예
if (attributeExists("username")) {
    setAttribute("username", "unclebob");
    ...
}
```

모호한 set 대신 명확한 용도로 이름 변경하여 사용

오류 코드보다 예외를 사용하라(1/2)

79

- try/catch를 사용하면 오류 처리 코드가 원래 코드에서 분리되므로 깔끔해 짐
- try/catch는 정상 동작과 오류 처리 동작이 뒤섞인 구조이므로 별도 함수로 추출

//if문을 통한 오류 코드

```
if (deletePage(page) == E_OK) {  
    if (registry.deleteReference(page.name) == E_OK) {  
        if (configKeys.deleteKey(page.name.makeKey())  
            == E_OK){  
            logger.log("page deleted");  
        } else {  
            logger.log("configKey not deleted");  
        }  
    } else {  
        logger.log("deleteReference from registry failed");  
    }  
} else {  
    logger.log("delete failed");  
    return E_ERROR;  
}
```

여러 단계로 중첩되는 코드를 야기하고, 오류 코드를 반환하면 호출자는 오류 코드를 바로 처리해야한다는 문제에 부딪힘

// try/catch 사용

```
void deletePageAndAllReferences(Page page)  
{  
    try {  
        deletePage(page);  
        registry.deleteReference(page.name);  
        configKeys.deleteKey(page.name.makeKey());  
    }  
    catch (const DeletePageException& ex) {  
        logger.log(ex.what());  
    }  
}
```

예외처리를 사용하면 정상 동작과 오류 처리 동작을 분리하여 코드가 깔끔해짐

```
class DeletePageException : public std::exception {  
public:  
    std::string s;  
    DeletePageExceptionError(std::string ss) : s(ss) {}  
    ~ DeletePageExceptionError() throw () {} // Updated  
    const char* what() const throw() { return s.c_str(); }  
}
```

오류 코드보다 예외를 사용하라(2/2)

80

- 오류코드에서 enum 클래스 사용을 경계하라
 - 새 오류코드를 추가하거나 변경할 때 코스트가 많이 든다.

```
enum class Error {  
    OK,  
    INVALID,  
    NO_SUCH,  
    LOCKED,  
    OUT_OF_RESOURCES,  
    WAITING_FOR_EVENT;  
}
```

Error enum 코드 변경 시
해당 Error enum을 사용하는 클래스 전부를
다시 컴파일 하고 배치 필요

반복하지 마라(1/2)

■ 중복은 모든 악의 근원

- 코드 길이가 늘어날 뿐 아니라 알고리즘이 변하면 손봐야 할 곳도 많고 실수의 위험이 높음
- 소프트웨어 개발에서 지금까지 일어난 혁신은 중복을 제거하기 위한 노력

■ 내포된 반복

```
static void scheduleEvent(int id, Day day, int minuteOfDay, int
event, int randomize){
    int i;
    for (i = 0; i < MAX_EVENTS; i++)
    {
        if (eventList[i].id == UNUSED)
        {
            eventList[i].id = id;
            eventList[i].day = day;
            eventList[i].minuteOfDay = minuteOfDay;
            eventList[i].event = event;
            eventList[i].randomize = randomize;
            resetRandomize(&eventList[i]);
            break;
        }
    }
}
```

함수를 분리하여 추출하고,
여러 함수에서 반복적으로 쓰이는 알고리즘을
include 함수로 추출하여 중복을 제거함

```
static void scheduleEvent(int id, Day day, long int minuteOfDay,
int control, int randomize){
    ScheduledLightEvent* event = findUnusedEvent();
    if (event) {
        event->id = id;
        event->day = day;
        event->minuteOfDay = minuteOfDay;
        event->event = control;
        event->randomize = randomize;
        resetRandomize(event);
    }
}
```

```
static ScheduledLightEvent* findUnusedEvent(void){
    int i;
    ScheduledLightEvent* event = eventList;
    for (i = 0; i < MAX_EVENTS; i++, event++) {
        if (!isInUse(event))
            return event;
    }
    return NULL;
}
```

Regular pointer를 피하라(1/2)

- Prefer simple object construction on the stack instead of on the heap

```
#include "Customer.h"
// ...
Customer customer;
```

```
//old C++ => return 0이 nullptr이라면, 호출자가 자원을 관리해야함,
Customer* createDefaultCustomer() {
    Customer* customer = new Customer();
    // Do something more with customer, e.g. configuring it, and at the end...
    return customer;
}
```

```
// Since C++11
Customer createDefaultCustomer() {
    Customer customer;
    // Do something with customer, and at the end...
    return customer;
}
```

Regular pointer를 피하라(2/2)

84

- In a function's argument list, use (const) references instead of pointers

```
void function(Type* argument);
```

```
void function(Type& argument);
```

- Return 이 nullptr인지 확인할 필요가 없다.
 - Function 안에서 *를 사용하지 않으므로 코드가 깨끗하다.
- If it is inevitable to deal with a pointer to a resource, use a smart one
 - you should wrap it immediately and take advantage of the so-called RAII idiom
 - If an API returns a raw pointer...
 - "it-depends-problem."

■ 함수를 어떻게 짜야 하는가

- 소프트웨어를 짜는 행위는 글짓기와 비슷하다.
- 초안은 서투르고 어수선하므로 계속 고치고 다듬고 정리한다.
- 처음에는 함수가 길고 복잡하며, 들여쓰기 단계도 많고 중복된 루프도 많으며, 인수 목록도 길지만 계속 다듬으면 결국 좋은 함수가 나온다.

(코드를 빠짐없이 테스트하는 단위 테스트를 만들고, 코드를 다듬고, 함수를 만들고, 이름을 바꾸고, 중복을 제거 한다. 지속적으로 코드를 다듬고 정리한다)

1

작게 만들어라

2

한 가지만 해라

3

Switch문

4

함수 인수

8

반복하지 마라

7

오류 코드보다
예외를 사용하라

6

명령과 조회를
분리하라

5

부수 효과를
일으키지 마라

Old C-style in C++ projects

Clean C++ Sustainable Software Development
Patterns and Best Practices with C++ 17

By Stephan Roth

- C는 아직 C++언어의 일부
 - C구성에 의해 클린(clean), 안전한(safe), 모던(modern) 코드에 문제점 야기

Prefer C++ Strings and Streams over Old C-Style char*

- Apart from a few exceptions, strings in a modern C++ program should be represented by C++ strings taken from the Standard Library.

```
char name[] = "Stephan";  
char* pointerToName = name;  
void function(char* pointerToCharacterArray) {  
    //...  
}
```

```
std::string name("Stephan");
```

- 이점
 - C++ 스스로 메모리 관리. String data의 lifetime에 자유롭다.
 - 다양한 방법의 변경
 - 편리한 iterator 인터페이스
 - C++ I/O 스트림(예: ostream, stringstream, fstream 등)과 완벽하게 연동
 - move-optimized

- 예외

- String constant

```
const char* const PUBLISHER = "Apress Media LLC";
```

- C-style API의 각 라이브러리와의 호환성

Avoid Using printf(), sprintf(), gets(), etc.

89

- Avoid using printf(), and also other unsafe C-functions, such as sprintf(), puts(), etc
 - 속도에 대한 이야기는 무의미
 - Type-unsafe

```
class Invoice {
public:
    explicit Invoice(const UniquelIdentifier& invoiceNumber);
    Invoice() = delete;
    void setRecipient(const InvoiceRecipientPtr& recipient);
    void setDateTimeOfInvoicing(const DateTime& dateTimeOfInvoicing);
    Money getSum() const;
    Money getSumWithoutTax() const;
    void addLineItem(const InvoiceLineItemPtr& lineItem);
    // ...possibly more member functions here...

private:
    friend std::ostream& operator<<(std::ostream& outstream, const Invoice& invoice);
    std::string getDateTimeOfInvoicingAsString() const;

    UniquelIdentifier invoiceNumber;
    DateTime dateTimeOfInvoicing;
    InvoiceRecipientPtr recipient;
    InvoiceLineItems invoiceLineItems;
};
```

Avoid Using printf(), sprintf(), gets(), etc.

90

- The insertion operator for class Invoice

```
// ...
std::ostream& operator<<(std::ostream& ostream, const Invoice& invoice) {
    ostream << "Invoice No.: " << invoice.invoiceNumber << "\n";
    ostream << "Recipient: " << *(invoice.recipient) << "\n";
    ostream << "Date/time: " << invoice.getDateTimeOfInvoicingAsString() << "\n";
    ostream << "Items:" << "\n";
    for (const auto& item : invoice.invoiceLineItems) {
        ostream << " " << *item << "\n";
    }
    ostream << "Amount invoiced: " << invoice.getSum() << std::endl;
    return ostream;
}
// ...
```

```
std::cout << instanceOfInvoice;
```

Prefer Standard Library Containers over Simple C-style Arrays

91

■ 이점

- Size를 포함

```
#include <array>
using MyTypeArray = std::array<MyType, 10>;
void function(const MyTypeArray& array) {
    const std::size_t arraySize = array.size();
    //...
}
```

- STL 호환 인터페이스

```
#include <array>
#include <algorithm>
using MyTypeArray = std::array<MyType, 10>;
MyTypeArray array;
void doSomethingWithEachElement(const MyType& element) {
    // ...
}
std::for_each(std::cbegin(array), std::cend(array), doSomethingWithEachElement);
```

Use C++ casts Instead of Old C-Style Casts

92

```
double d { 3.1415 };  
int i = (int)d;
```

```
int i = static_cast<int>(d);
```

■ 이점

- Compile 시 error 체크하도록
- 에디터에서 검색의 편의

■ `dynamic_cast<>` , `reinterpret_cast<>` : 사용하지 않도록

Avoid Macros

- object-like macros => constant expressions

```
#define DANGEROUS 1024+1024
```

```
constexpr int HARMLESS = 1024 + 1024;
```

int value = DANGEROUS * 2; 의 결과는???

- function-like macros =>

```
#define MIN(a,b) (((a)<(b))?(a):(b))  
#define MAX(a,b) (((a)>(b))?(a):(b))  
  
int maximum = MAX(12, value++);
```

```
#include <algorithm>  
// ...  
int maximum = std::max(12, value++);
```

오류 처리

■ 오류 처리는 프로그램에 반드시 필요한 요소

- 뭔가 잘못될 가능성은 언제나 존재
- 잘못된 것을 바로 잡을 책임은 바로 우리 프로그래머에게 있음

■ 오류 처리 코드로 인해 프로그램을 이해하기 어려워진다면 깨끗한 코드가 아님

- 여기저기 흩어진 오류 처리 코드 때문에 실제 코드가 하는 일을 파악하기 어려움
- 오류 처리 코드로 인해 프로그램 논리를 이해하기 어려워진다면 깨끗한 코드라 부르기 어렵다.

■ 오류처리하는 기법과 고려사항 소개

오류 코드보다 예외(Exception)를 사용하라

96

- Exception를 지원하지 않는 언어는 오류를 처리하고 보고하는 방법이 제한적임
 - 오류 플래그를 설정하거나 호출자에게 오류 코드를 반환하는 방법이 전부임

//에러 코드 사용

```
class DeviceController {  
    ...  
    void sendShutDown() {  
        DeviceHandle handle = getHandle(DEV1);  
        // 디바이스 상태를 점검한다.  
        if (handle != DeviceHandle.INVALID) {  
            // 레코드 필드에 디바이스 상태를 저장한다.  
            retrieveDeviceRecord(handle);  
            // 디바이스가 일시정지 상태가 아니라면 종료한다.  
            if (record.getStatus() != DEVICE_SUSPENDED) {  
                pauseDevice(handle);  
                clearDeviceWorkQueue(handle);  
                closeDevice(handle);  
            } else {  
                logger.log("Device suspended. Unable to shut down");  
            }  
        } else {  
            logger.log("Invalid handle for: " + DEV1.toString());  
        }  
    }  
    ...  
}
```

함수를 호출한 즉시 오류를 확인해야 하기에 호출자 코드가 복잡해짐 → 실수로 처리하지 않을 가능성이 높아짐

//예외 사용

```
class DeviceController {  
    ...  
    void sendShutDown() {  
        try {  
            DeviceHandle handle = getHandle(DEV1);  
            retrieveDeviceRecord(handle);  
            pauseDevice(handle);  
            clearDeviceWorkQueue(handle);  
            closeDevice(handle);  
        }  
        catch (const DeviceShutDownError& ex) {  
            logger.log(ex.what());  
        }  
    }  
  
    DeviceHandle getHandle(DeviceID id) {  
        ...  
        if (handle != DeviceHandle.INVALID) {  
            throw DeviceShutDownError("it's the end of the world!");  
        }  
        ...  
    }  
}
```

논리/로직을 수행하는 알고리즘(디바이스 종료)과 오류를 처리하는 알고리즘이 분리됨 → 호출자 코드가 깔끔해짐

Try-Catch 문부터 작성하라 (1/4)

- Exception 이 발생할 코드를 짤 때 try - catch 문부터 구현
 - try블록은 발생할 수 있는 예외 상황들을 고려하지 않고 기본 로직 구현 가능
 - 예외가 발생하여 실행이 중단될 경우, catch 블록에서 변경된 사항들을 원래대로 관리
- ➔ Try블록에서 무슨 일이 생기든지 catch 블록은 일관성 있게 유지해야함

Try-Catch 문부터 작성하라 (2/4)

98

■ 예제 - 파일을 열어 직렬화된 객체를 읽어 들이는 코드 작성

- Step 1 : Exception을 던지는지 알아보는 **단위 테스트** 작성

```
void retrieveSectionShouldThrowOnInvalidFileName {  
    ASSERT_THROW(retrieveSection("Invalid - file"), StorageException)
```

파일이 없으면 exception이 발생하는지
알아 보는 단위 테스트

- Step 2 : 단위 테스트에 맞춰 코드 구현 → 코드가 Exception 발생하지 않음으로 테스트 실패

```
vector<string> retrieveSection(string fileName){  
    return vector<string>{};  
}
```

- Step 3 : 단위 테스트를 통과하는 **코드 구현** (잘못된 파일 접근을 시도하도록 구현)

```
vector<string> retrieveSection(string fileName){  
    ifstream read;  
    try{  
        read.open(fileName);  
    }catch(std::exception const& e){  
        throw new StorageException("retrieval error");  
    }  
    return vector<string>{};  
}
```

Try-Catch 문부터 작성하라 (3/4)

- Step 4 : 테스트 성공 후 리팩터링 (catch 블록의 예외 유형을 좁힘)

```
vector<string> retrieveSection(string fileName){  
    ifstream read;  
    try{  
        read.open(fileName);  
        read.close();  
    }catch(const std::exception& e){  
        throw new FileNotFoundException("retrieval error");  
    }  
    return vector<string>{};  
}
```

Catch블록에서 예외 유형을 좁혀
실제로 FileInputStream 생성자가 던지는
FileNotFoundException을 잡아낸다

- Step 5 : TDD를 활용해 나머지 논리/로직 추가
 - Try-catch 구조로 범위를 정의했으므로 TDD를 사용해 필요한 나머지 논리를 추가함
 - 나머지 논리는 FileInputStream을 생성하는 코드와 close 호출문 사이에 넣으며, 오류나 예외가 전혀 발생하지 않는다고 가정한다.

Try-Catch 문부터 작성하라 (4/4)

100

■ 권장 구현 방법

1. 먼저 **강제로 예외를 일으키는** 테스트 케이스 작성
2. 테스트를 **통과하게 코드를 작성**
 - ➔ 테스트를 통과하기 위한 기본 로직을 먼저 구현하게 됨
 - ➔ 기존에 구현하려고 했던 의도를 잃지 않으면서 구현 가능

Exception에 의미를 제공하라

101

- Exception을 던질 때는 전후 상황을 충분히 포함하도록 함
 - 실패한 코드의 의도를 파악하려면 Exception 의 호출 스택만으로 부족
 - 오류 메시지에 정보를 담아 Exception과 함께 던짐
 - 오류가 발생한 연산 이름, 오류 유형 전달
 - 애플리케이션이 로깅 기능을 사용한다면
 - catch 블록에서 오류를 기록하도록 충분한 정보를 넘겨줌
- ➔ 오류가 발생한 원인과 위치를 찾기 쉬워짐

호출자를 고려해 Exception 클래스를 정의하라(1/2)

102

- 오류를 분류하는 방법은 많음
 - 오류가 발생한 위치 (오류 발생 컴포넌트)
 - 오류의 유형 (디바이스 실패, 네트워크 실패, 프로그래밍 오류 등)
- But, 오류를 잡아내는 방법이 더 중요

호출자를 고려해 Exception 클래스를 정의하라(2/2)

103

■ 모든 예외 catch 사용

```
ACMEPort port = ACMEPort(12);
try {
    port.open();
} catch (DeviceResponseException e) {
    reportPortError(e);
    logger.log("Device response exception", e);
} catch (ATM1212UnlockedException e) {
    reportPortError(e);
    logger.log("Unlock exception", e);
} catch (GMXError e) {
    reportPortError(e);
    logger.log("Device response exception");
}
```

외부 라이브러리가 던질 모든 예외를 잡음
→ 다수 중복 코드 발생

대다수 상황에서 오류를 처리하는 방식
1) 오류를 기록 (로깅)
2) 프로그램을 계속 수행할지 확인
→ 예외 처리 방식이 예외 유형과 무관

■ Wrapper 사용

```
LocalPort port = LocalPort(12);
try {
    port.open();
} catch (PortDeviceFailure e) {
    reportError(e);
    logger.log(e.getMessage(), e);
}
```

외부 API를 감싸면 외부 라이브러리와 프로그램 사이의 의존성이 크게 감소
→ 교체 비용 감소, 테스트 쉬워짐, 특정 업체의 API 에서 독립

```
class LocalPort {
public:
    LocalPort(int portNumber) {
        innerPort = ACMEPort(portNumber);
    }
    void open() {
        try {
            innerPort.open();
        } catch (DeviceResponseException e) {
            throw PortDeviceFailure(e);
        } catch (ATM1212UnlockedException e) {
            throw PortDeviceFailure(e);
        } catch (GMXError e) {
            throw PortDeviceFailure(e);
        }
    }
private:
    ACMEPort innerPort;
}
```

※ LocalPort 는 ACMEPort 가 던지는 예외를 변환하는 Wrapper 클래스

정상 흐름(Normal Flow)을 정의하라

104

- 비즈니스 논리와 오류 처리를 잘 분리하면 코드가 깨끗해지지만 오류 감지가 프로그램 언저리로 밀려남

// 비용 청구 애플리케이션에서 총계를 계산하는 코드

```
try {  
    MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());  
    m_total += expenses.getTotal();  
} catch(MealExpensesNotFound e) {  
    m_total += getMealPerDiem();  
}
```

식비를 비용으로 청구했다면 (try 블록)
직원이 청구한 식비를 총계에 더함
비용으로 청구하지 않았다면 (catch 블록)
일일 기본 식비를 총계에 더함
➔ 예외로 특수한 상황을 처리하여 논리를 따라가기 어려움

- 특수 사례 패턴(Special Case Pattern) 을 사용하라

- 클래스를 만들거나 객체를 조작해 특수 사례를 처리하는 방식,
그러면 클라이언트 코드가 예외적인 상황을 처리할 필요가 없음

//언제나 MealExpense 객체를 반환

```
MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());  
m_total += expenses.getTotal();  
...  
class PerDiemMealExpenses : public MealExpenses {  
public:  
    virtual int getTotal() {  
        // 기본값으로 일일 기본 식비를 반환한다.  
    }  
}
```

expenseReportDAO는 언제나 MealExpenses 객체를 반환,
만약 청구한 식비가 없다면 일일 기본 식비를 반환하는
MealExpense 객체(PerDiemMealExpenses) 를 반환

➔ 클래스나 객체가 예외적인 상황을 캡슐화해서 처리하면,
클라이언트 코드가 예외적인 상황을 처리할 필요가 없음

- null을 반환하는 습관은 개발자가 흔히 저지르는 오류를 유발하는 행위
 - null을 반환하는 코드는 일거리를 늘릴 뿐만 아니라 호출자에게 문제를 떠넘김
 - 누구 하나라도 null 확인을 빼먹는다면 애플리케이션이 통제 불능!

//null 확인 코드로 가득한 애플리케이션

```
Customer* customer = findCustomerByName("Stephan");
if (customer != nullptr) {
    OrderedProducts* orderedProducts = customer->getAllOrderedProducts();
    if (orderedProducts != nullptr) {
        // Do something with orderedProducts...
    } else {
        // And what should we do here?
    }
} else {
    // And what should we do here?
}
```

둘째 행에 customer 객체 null 확인 누락
→ NullPointerException이 발생. null 확인이 누락된 문제라 말하기 쉬우나, 실상은 null 확인이 너무 많은 것이 문제

getEmployees 메서드가 null 대신 **빈 리스트(emptyList)**를 반환하도록 변경

- 메서드에서 null을 반환하려면, 예외를 던지거나 특수 사례 객체를 반환
- 사용하려는 외부 API가 null을 반환 한다면, wrapper 메서드 구현

// null 반환 예제

```
vector<Employee> employees = getEmployees();
if (&employees != nullptr) {
    for(Employee e : employees) {
        totalPay += e.getPay();
    }
}
```

// 빈 리스트 반환

```
vector<Employee> employees = getEmployees();
for(Employee e : employees) {
    totalPay += e.getPay();
}
...
vector<Employee> getEmployees() {
    if( .. 직원이 없다면 .. )
        return std::vector<Employee>(0);
}
```

null을 전달하지 마라

106

- 정상적인 인수로 null을 기대하는 API가 아니라면 메서드로 null 전달은 지양
 - 대다수 프로그래밍 언어는 호출자가 실수로 넘기는 null을 적절히 처리할 방법이 없음
 - 애초에 null을 넘기지 못하도록 금지하는 정책을 쓰는 것도 좋음

// null 전달 예제

```
class MetricsCalculator
{
public:
    double xProjection(Point* p1, Point* p2) {
        return (p2->x - p1->x) * 1.5;
    }
    ...
}

calculator.xProjection(nullptr, new Point(12, 13));
```

누군가 인수로 null을 전달하면
어떤 일이 벌어질까?
NullPointerException 발생

// 대안1) 새로운 Exception 유형 던지기

```
class MetricsCalculator
{
public:
    double xProjection(Point* p1, Point* p2) {
        if (p1 == nullptr || p2 == nullptr) {
            throw InvalidArgumentException(
                "Invalid argument for MetricsCalculator.xProjection");
        }
        return (p2->x - p1->x) * 1.5;
    }
}
```

InvalidArgumentException
처리가 여전히 필요

// 대안2) assert문 사용

```
class MetricsCalculator
{
public:
    double xProjection(Point* p1, Point* p2) {
        assert(p1 != nullptr);
        assert(p2 != nullptr);
        return (p2->x - p1->x) * 1.5;
    }
}
```

누군가 null을 전달하면
여전히 실행 오류 발생

Manual resource release의 위험

107

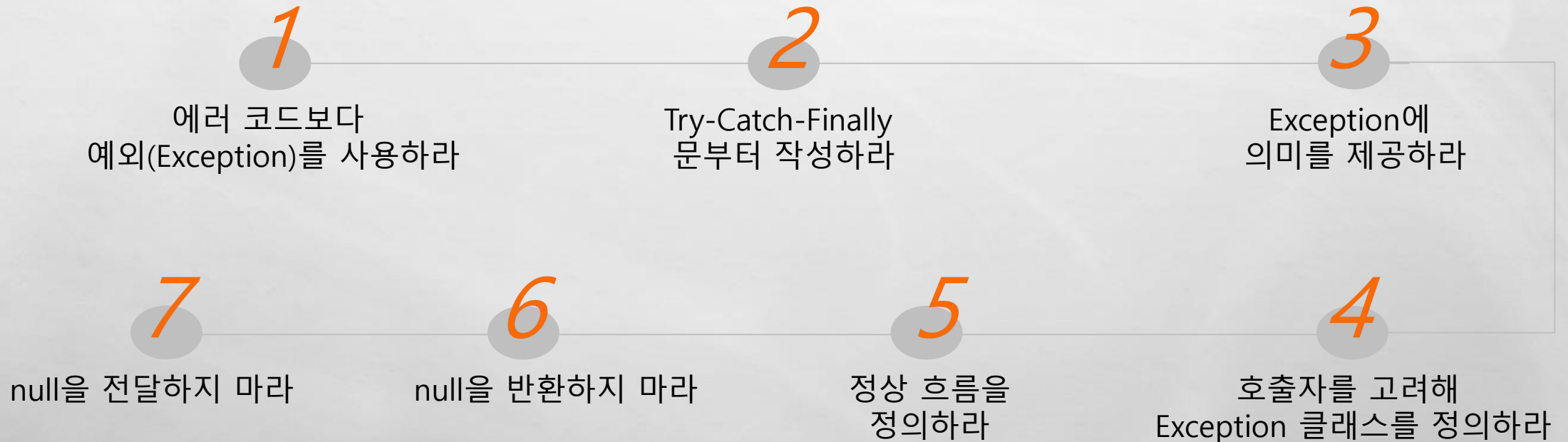
- 에러가 발생한 경우 수동으로 release 하는 것은 위험
 - 자원이 항상 해제 되도록 만든다.
 - Smart pointer 의 사용

```
void f(int i)
{
    int* p = new int[12];
    if (i < 17) throw Bad{"in f()", i};
    ...
}
```

```
void f(int i)
{
    int* p = new int[12];
    if (i < 17) {
        delete[] p;
        throw Bad{"in f()", i};
    }
    ...
}
```

```
void f(int i)
{
    auto p = make_unique<int[]>(12);
    ...
    if (i < 17) throw Bad{"in f()", i};
    ...
}
```

- 깨끗한 코드는 읽기도 좋아야 하지만 안정성도 높아야 함
 - 오류 처리를 프로그램 논리와 분리
 - ➔ 독립적인 추론이 가능해지며 코드 유지보수성도 크게 높아짐
- 우아하고 고상하게 오류를 처리하는 기법과 고려 사항



경계

경계(Boundaries) 개요

110

- 시스템에 들어가는 모든 소프트웨어를 직접 개발하는 경우는 드물
 - 오픈 소스 활용, 패키지 구매, 다른 팀이 제공하는 컴포넌트
- ➔ 우리 코드와 외부 코드의 경계를 깔끔하게 통합 필요



■ 패키지/프레임워크의 인터페이스 제공자와 사용자 사이에 입장차 존재

- **제공자**: 더 많은 환경에서 돌아가도록 적응성을 최대한 넓히려 함
- **사용자**: 자신의 요구에 집중하는 인터페이스를 바람

```
at
begin
cbegin
cend
clear
count
crbegin
crend
emplace
emplace_hint
empty
end
equal_range
erase
extract
find
get_allocator
insert
insert_or_assign
key_comp
lower_bound
max_size
merge
operator+=
operator[]
rbegin
rend
size
swap
try_emplace
upper_bound
value_comp
~map
```

#include<map>은 다양한 인터페이스로 수많은 기능을 제공함
➔ 제공자 측면의 입장에서 기능성과 유연성 확보

But, 사용자 입장에서 잘못 사용할 여지가 많음

Map을 인수나 반환값으로 여기저기 넘긴다면,

예상치 못한 결과가 발생할 수 있음

(예를 들어 Map 사용자는 누구나 내용을 수정/삭제 권한을 갖게 되어 저장한 데이터가 삭제될 수 도 있음)

외부 코드 사용하기 (2/2)

112

■ 경계 인터페이스를 여기저기 넘기지 말라

// 기본형

```
map<string, Sensor> sensors;  
...  
Sensor s = sensors.at(sensorId);
```

// map 을 캡슐화하여 사용

```
class Sensors{  
public:  
    Sensor getByld(string id) {  
        return m_sensors.at(id);  
    }  
...  
private:  
    map<string, Sensor> m_sensors;  
};
```

Generics를 사용하여 코드 가독성 증가

But,

1. 여전히 사용자에게 필요하지 않은 기능 제공
2. map<string, Sensor> 를 여기저기서 사용한다면, map 인터페이스가 변할 경우 수정할 코드가 많아짐

경계 인터페이스인 Map을 Sensors class 안으로 숨김

1. Sensors 사용자는 내부 구현은 신경 쓸 필요 없음
2. Map 인터페이스가 변하더라도 나머지 프로그램에 영향을 미치지 않음
3. 프로그램에 필요한 인터페이스만 제공
→ 오용 방지를 위해 설계 규칙과 비즈니스 규칙을 따르도록 강제

※ 주의 : 매번 캡슐화하라는 의미는 아님, Map을 여기저기 넘기지 말라는 의미

Map과 같은 경계 인터페이스를 이용할때는 이를 이용하는 클래스나 클래스 계열 밖으로 노출되지 않도록 주의해야함

(Map 인스턴스를 공개 API의 인수로 넘기거나 반환값을 사용하지 않아야 함)

- 우리 자신을 위해 외부 패키지/라이브러리를 테스트하자
 - 외부 라이브러리의 사용법이 분명치 않다면
 1. 대개는 하루나 이틀 (아니면 더 오랫동안) 문서를 읽으며 사용법을 결정
 2. 우리 코드를 작성하여 라이브러리가 예상대로 동작하는지 확인
 3. 우리 버그인지 라이브러리 버그인지 찾아내느라 오랜 디버깅...
 - **학습 테스트 : 간단한 테스트 케이스를 작성해 외부 코드를 익힘**
 - 프로그램에서 사용하려는 방식대로 테스트 케이스에서 외부 API를 호출
 - 통제된 Test 환경에서, API 를 사용하려는 목적에 초점을 맞춰 해당 API를 제대로 이해하는지 확인

1. 패키지를 내려 받아 소개 페이지 Open
2. 문서를 자세히 읽기 전에 첫 번째 테스트 케이스 작성
3. 테스트 케이스 수행 ➔ 오류 발생
4. 구글 검색을 통해 수정
5. 이해가 안 되는 부분에 대하여 구글을 뒤지고, 문서를 읽어보고, 테스트를 돌려서 이해한다.
6. 습득한 지식을 통해 독자적인 클래스로 캡슐화 : 나머지 프로그램은 경계 인터페이스를 몰라도 됨

■ 학습 테스트는 공짜 이상이다

- 학습 테스트에 드는 비용은 없습니다(어쨌든 API를 배워야 하므로...)

오히려 필요한 지식을 확보할 수 있는 손쉬운 방법, 이해도를 높여주는 정확한 실험

- 학습 테스트를 통해 확보한 테스트 케이스를 통해,

패키지의 새 버전이 나오면 학습 테스트를 수행하여 우리 코드와 호환되는지 확인 가능

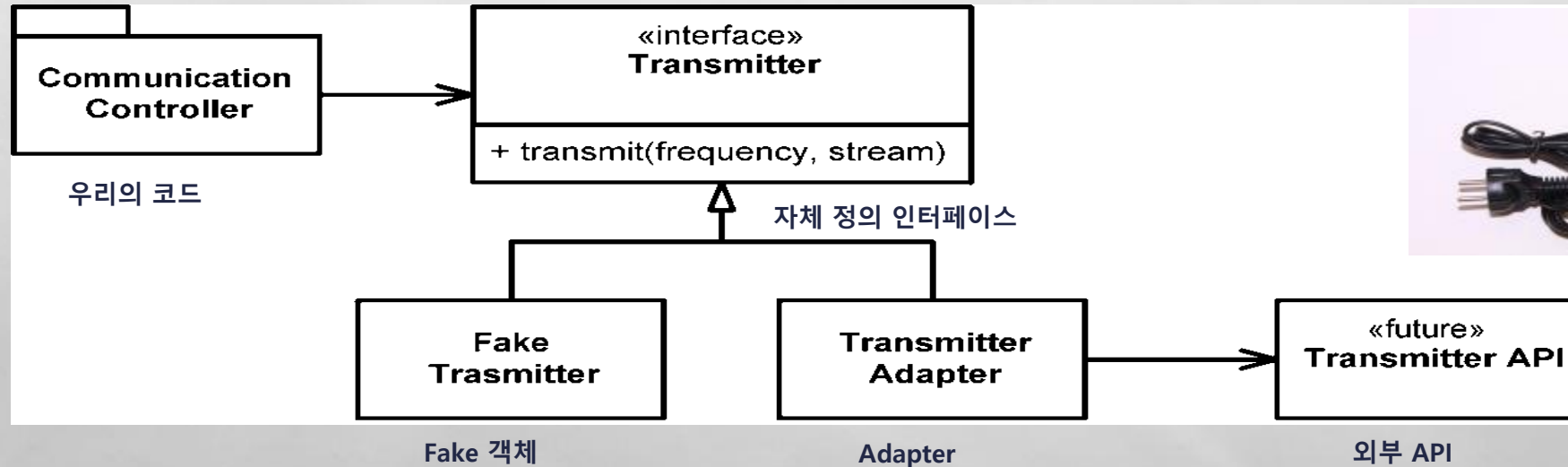
➔ 경계 테스트가 있다면 **패키지의 새 버전으로 이전하기 쉬워짐,**

그렇지 않다면 낡은 버전을 필요 이상으로 오랫동안 사용하려는 유혹에 빠지기 쉽다.

아직 존재하지 않는 코드를 사용하기

116

- 현재 알지 못하는 코드를 사용해야 한다면 자체 인터페이스를 정의하여 사용
 - 다른 팀과 동시에 진행되는 협력 프로젝트의 경우, 아직 인터페이스도 정의되지 않았지만 구현을 진행해야 하는 경우, 발생 가능
- ➔ 우리에게 필요한 인터페이스를 정의하여 이를 통해 구현하고, 향후 외부 API가 정의되면 Adapter 패턴으로 API 사용을 캡슐화하여 사용
- 이와 같은 설계는 테스트에서도 이득
 - Fake 객체를 사용하여 우리의 코드를 테스트 가능
 - 향후 외부 API와 라이브러리가 제공되면, 경계 테스트 케이스를 생성하여 API를 올바르게 사용하는지 동작 테스트 가능



- 경계에서는 많은 변경이 발생
 - 소프트웨어 설계가 우수하다면 변경하는데 비용을 최소화 할 수 있음
 - 통제하지 못하는 코드를 사용할 때, 향후 변경 비용을 줄일 수 있도록 주의 필요
- 경계에 위치하는 코드를 깔끔히 분리해야 함
 - 외부 패키지를 세세하게 알 필요 없음
 - 통제가 불가능한 외부 패키지에 의존하기 보다 통제 가능한 우리 코드에 의존
- 깨끗한 경계를 위한 제안 사항
 - 기대치를 정의하는 테스트 케이스 작성
 - 외부 패키지를 호출하는 코드를 가능한 줄여 경계를 관리
 - 패키지가 제공하는 인터페이스를 우리가 원하는 인터페이스로 변환하여 사용
 - 새로운 클래스로 경계를 감싸거나 Adapter 패턴을 사용
 - ➔ 코드 가독성 / 경계 인터페이스를 사용하는 일관성 향상됨
 - ➔ 외부 패키지가 변했을 때 변경할 코드 줄어듦

클래스

Test-Driven development for Embedded C

By James W.Grenning

■ 깨끗한 클래스

- 코드의 표현력과 함수에 아무리 신경을 쓰더라도 깨끗한 코드를 얻기는 어려움
→ 더 높은 차원(클래스 레벨)의 노력이 필요
- 가능한 비공개 상태를 유지(캡슐화) 할 온갖 방법을 강구해야 함

■ 함수와 마찬가지로 클래스도 작게

- 한 클래스는 하나의 책임만 가져야 한다[SRP(Single Responsibility Principle)]
- 응집도(Cohesion)를 유지하도록 여러 개 클래스로 분리

■ 변경하기 쉬운 클래스

- 클래스를 체계적으로 정리하여 변경에 수반하는 위험을 낮춰야 함 [OCP(Open/Close principle)]
- 인터페이스와 추상 클래스를 사용해 구현이 미치는 영향을 격리해야 함
[DIP(Dependency Inversion Principle)]
→ 프로그래머는 "추상화에 의존해야지, 구체화에 의존하면 안 된다."

Keep Classes Small

120

■ 클래스 길이가 작으면 끝?

- 함수 크기는 '물리적인 행 수'로 측정, 클래스 크기는 '맡은 책임'으로 측정

Single Responsibility Principle (SRP)

121

■ 클래스는 단 하나의 책임만 가져야 한다 [SRP – Single Responsibility Principle]

- 클래스나 모듈을 변경할 이유가 하나, 단 하나 뿐 이어야 한다.
- 단일 책임 원칙은 객체 지향 설계에서 중요하고, 이해하고 지키기 수월한 개념
- 그러나, "깨끗하고 체계적인 소프트웨어"보다 "돌아가는 소프트웨어"에 초점을 맞추다 보니 클래스 설계자가 가장 무시하는 규칙이기도 하다.

■ 클래스 이름은 해당 클래스의 책임 기술해야 한다

- Processor, Manager, Super 와 같은 모호한 단어가 들어있다면, 클래스에 여러 책임을 떠 안겼다는 증거
- 클래스 설명은 if, and, or, but을 사용하지 않고, 25 단어 내외로 설명 가능해야 한다

■ "소프트웨어를 돌아가게 만드는 활동" vs "소프트웨어를 깨끗하게 만드는 활동"

- 프로그램이 돌아간다고 일이 끝난 게 아님 → 깨끗하고 체계적인 SW로 유지 관리 필요
- 큰 서랍에 모두 던져놓기 vs 작은 서랍을 많이 두고 기능과 이름이 명확한 컴포넌트로 관리

➔ **큰 클래스 몇 개가 아니라 작은 클래스 여럿으로 이뤄진 시스템이 더 바람직하다.**

즉, 작은 클래스는 각자 맡은 책임이 하나며, 변경할 이유가 하나며,

다른 작은 클래스와 협력해 시스템에 필요한 동작을 수행한다.

클래스는 작아야 한다 (1/4)

122

■ 클래스 길이가 작으면 끝?

- 함수 크기는 '물리적인 행 수'로 측정, 클래스 크기는 '맡은 책임'으로 측정

충분히 작을까?

```
class SuperDashboard {  
public:  
  ①Component getLastFocusedComponent();  
    void setLastFocused(Component lastFocused);  
  ②int getMajorVersionNumber();  
    int getMinorVersionNumber();  
  ③int getBuildNumber();  
}
```

SuperDashboaard가 method 몇 개만 포함한다면?

→ method 수가 적음에도 **책임이 너무 많음**

1. 마지막 포커스 컴포넌트 get/set
2. 버전 추적 메커니즘
3. 빌드 번호 추적 메커니즘



단일 책임 클래스

```
class Version{  
public:  
    int getMajorVersionNumber()  
    int getMinorVersionNumber()  
    int getBuildNumber()  
}
```

버전 정보를 다루는 method를
독자적 클래스로 빼내 **책임을 줄인다**

클래스는 작아야 한다 (2/4)

123

- 클래스는 멤버 변수 수는 적게, 각 메서드는 멤버 변수를 하나이상 사용
 - 일반적으로 메서드가 변수를 더 많이 사용할수록 메서드와 클래스는 응집도가 높다
- 응집도가 높다는 것은
 - 클래스에 속한 메서드와 변수가 서로 의존하며, 논리적인 단위로 묶인다는 의미

```
class Stack {
public:
    int size() {
        return topOfStack_;
    }

    void push(int element) {
        topOfStack_++;
        elements_.push_back(element);
    }

    int pop {
        if (topOfStack_ == 0)
            throw "ERROR : STACK IS EMPTY";
        int element = elements_[--topOfStack_];
        elements_.remove(topOfStack_);
        return element;
    }
private:
    int topOfStack_;
    vector<int> elements_;
}
```

Size()를 제외한 모든 메서드는 두 변수 모두 사용
(응집도가 높다)

※ 함수를 작게, 매개변수 목록을 짧게'라는 전략을 따르다 보면 때때로 몇몇 메서드만이 사용하는 인스턴스 변수가 아주 많아진다.
이는 십중팔구 새로운 클래스로 쪼개야 한다는 신호다.
응집도가 높아지도록 변수와 메서드를 적절히 분리해 새로운 클래스로 쪼개 준다.

클래스는 작아야 한다 (3/4)

124

■ 클래스가 응집력을 잃으면 쪼개라

- 큰 함수를 작은 함수로 쪼개면 종종 작은 클래스 여럿으로 쪼갤 기회가 생김

➔ 프로그램이 점점 더 체계가 잡히고 구조가 투명해짐

예) 변수가 아주 많은 큰 함수 일부를 작은 함수 하나로 빼내고자 하는 경우

- 그런데 빼내려는 코드가 큰 함수에 정의된 변수 넷을 사용한다
- 변수 네 개를 새 함수의 인수로 넘겨야 옳을까?
➔ 전혀 아니다!
- 만약 네 변수를 클래스 멤버 변수로 승격하면 새 함수 인수가 필요 없어진다.
➔ 그만큼 함수를 쪼개기 쉬워진다
- 그러나 몇몇 함수만 사용하는 멤버 변수가 점점 늘어나기 때문에, 클래스 응집력을 잃는다
- 이때 응집력을 잃은 클래스를 쪼갬다
- 큰 함수를 작은 함수 여럿으로 쪼개다 보면 종종 작은 클래스 여럿으로 쪼갤 기회가 생긴다.
그러면서 프로그램에 점점 더 체계가 잡히고 구조가 투명해진다.

```
void PrimePrinter::print()
{
    const int M = 1000;
    const int RR = 50;
    const int CC = 4;
    const int ORDMAX = 30;
    int P[M + 1];
    int J;
    int K;
    bool JPRIME;

    ...
    // 변수 17개

    while (K < M) {
        do {
            ...
        } while (!JPRIME);
        K = K + 1;
        P[K] = J;
    }
    {
        ...
    }
}
```

소수 목록 생성 부분은
PrimeGenerator 클래스로 분리

클래스는 작아야 한다 (4/4)

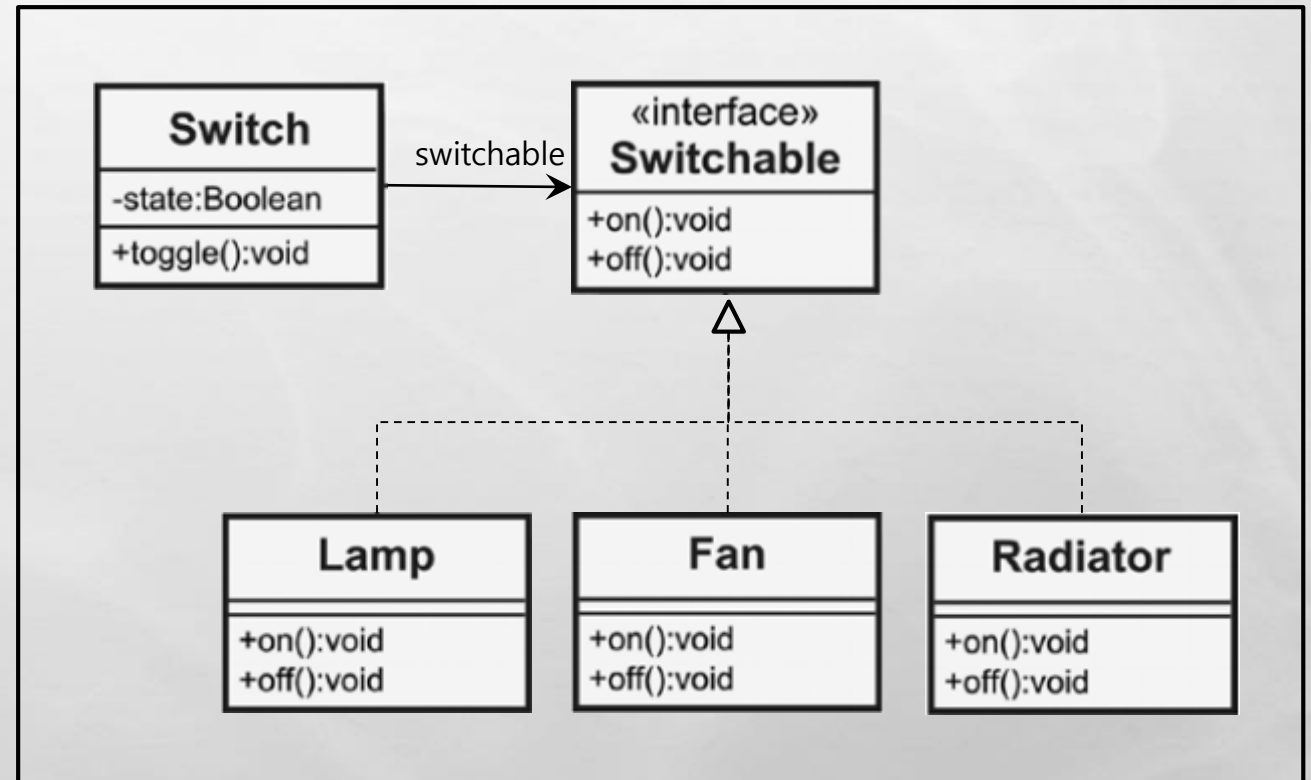
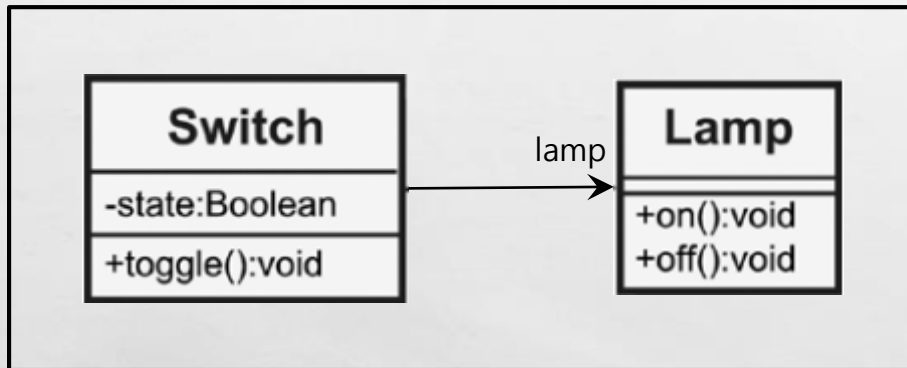
125

- 클래스를 쪼갰더니 프로그램이 길어졌다. 그 이유는?
 - 좀 더 길고 서술적인 변수 이름이 사용되었다
 - 주석을 추가하는 수단으로 함수 선언과 클래스 선언을 활용하였다
 - 가독성을 높이고자 공백을 추가하고 형식을 맞추었다
- 재구현이 아니다! 프로그램을 처음부터 다시 짜지 않았다.
 - 자세히 살펴보면 동작 원리가 동일하다는 사실을 눈치 챌 수 있다 ☺
 - 원래 프로그램의 정확한 동작을 검증하는 Test Suite 작성,
그리고 한 번에 하나씩 수 차례에 걸쳐 조금씩 코드를 변경하여
원래 프로그램을 정리한 최종 결과 프로그램

Open-Closed Principle (OCP)

126

- 이상적인 시스템은 신규 기능 추가 시, 기존 코드 변경 없이 시스템만 확장
 - 깨끗한 시스템은 클래스를 체계적으로 정리해 변경에 수반하는 위험을 낮춘다.
 - 클래스는 확장에 개방적이고, 수정에 폐쇄적이어야 한다. **[OCP: 개방-폐쇄 원칙]**



Open-Closed Principle (OCP)

127

```
class Switchable {
public:
    virtual void on() = 0;
    virtual void off() = 0;
};

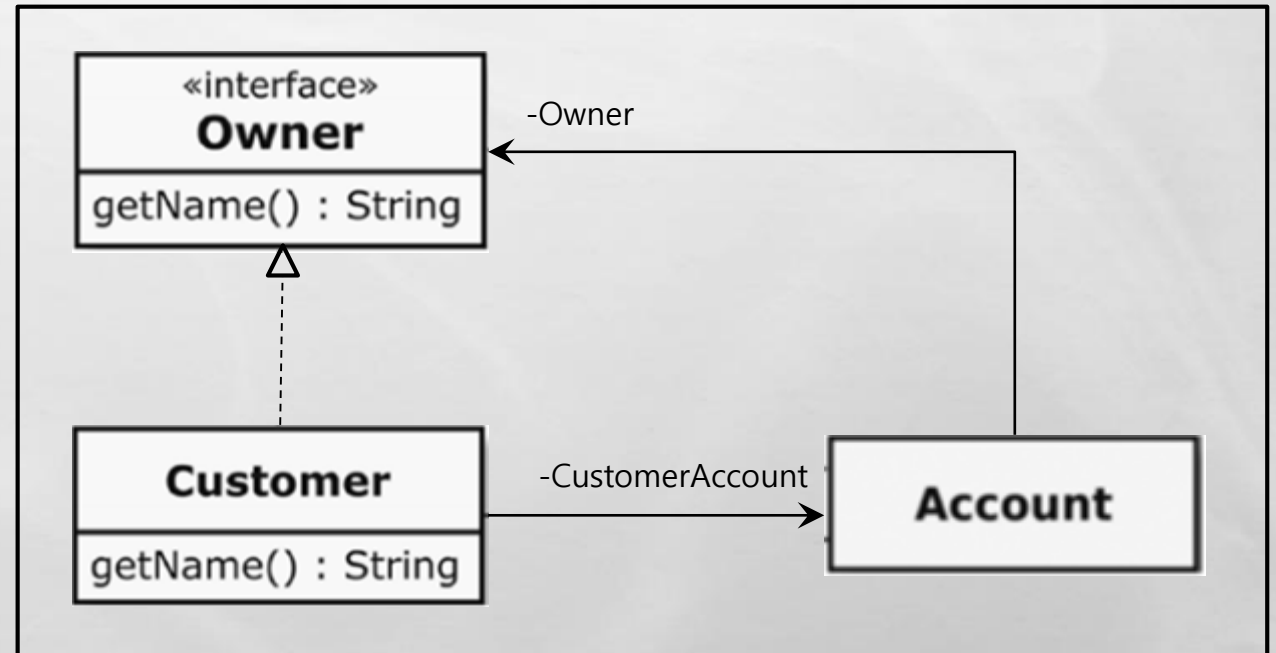
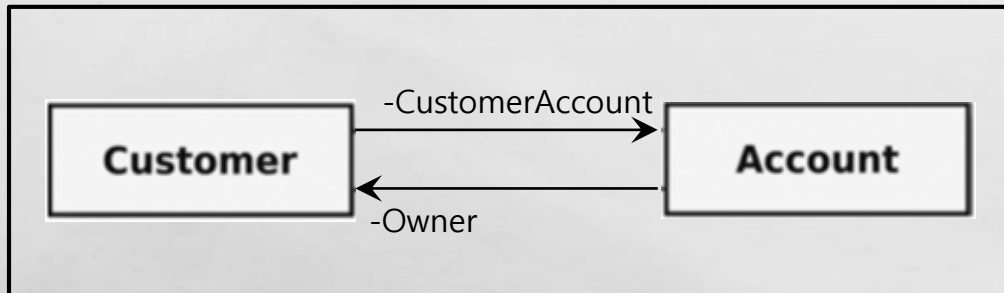
class Switch {
private:
    Switchable& switchable;
    bool state {false};
public:
    Switch(Switchable& switchable) : switchable(switchable) {}
    void toggle() {
        if (state) {
            state = false;
            switchable.off();
        } else {
            state = true;
            switchable.on();
        }
    }
};
```

```
class Lamp : public Switchable {
public:
    void on() override {
        // ...
    }
    void off() override {
        // ...
    }
};
```

Dependency Inversion Principle (DIP)

128

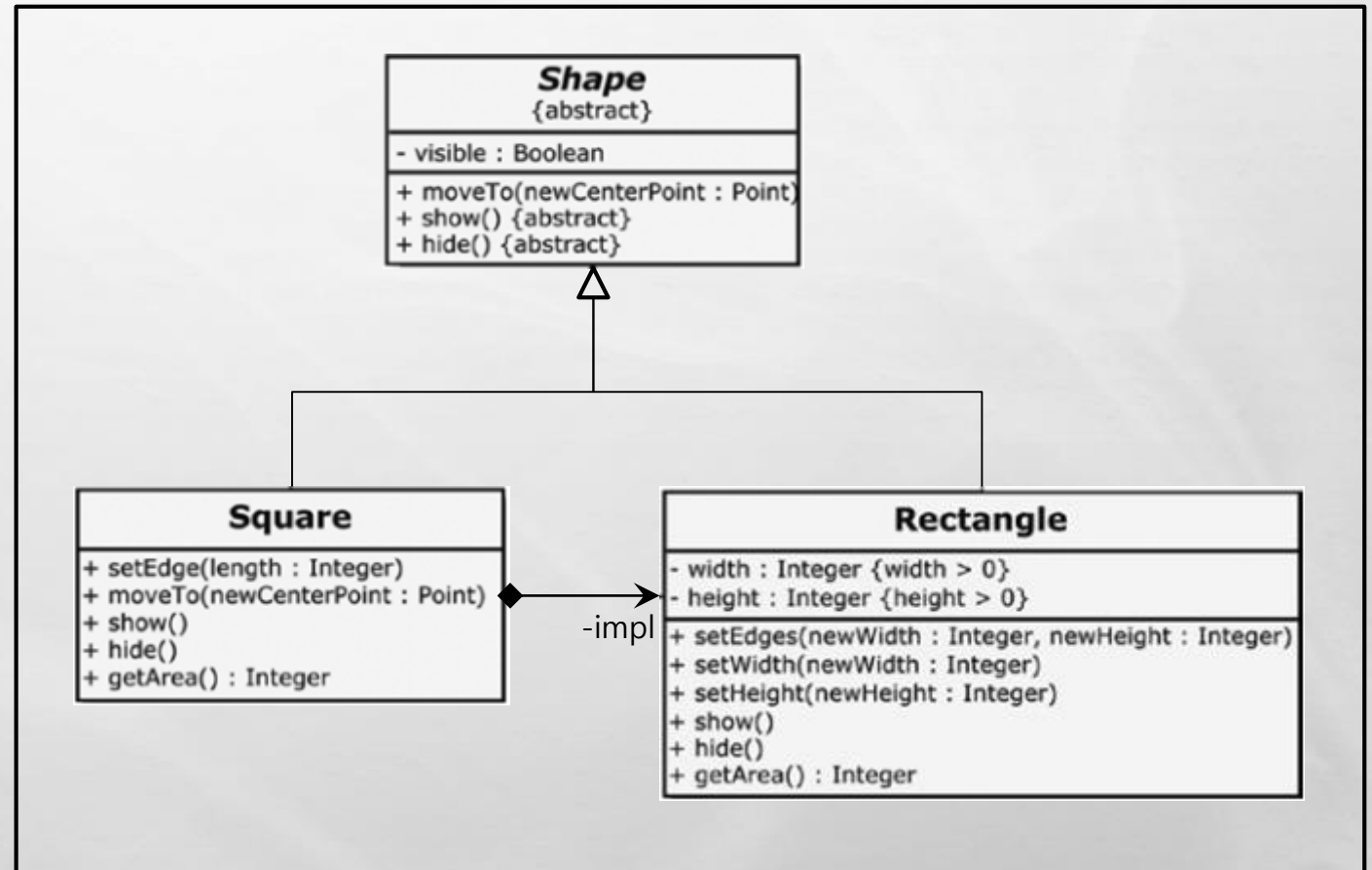
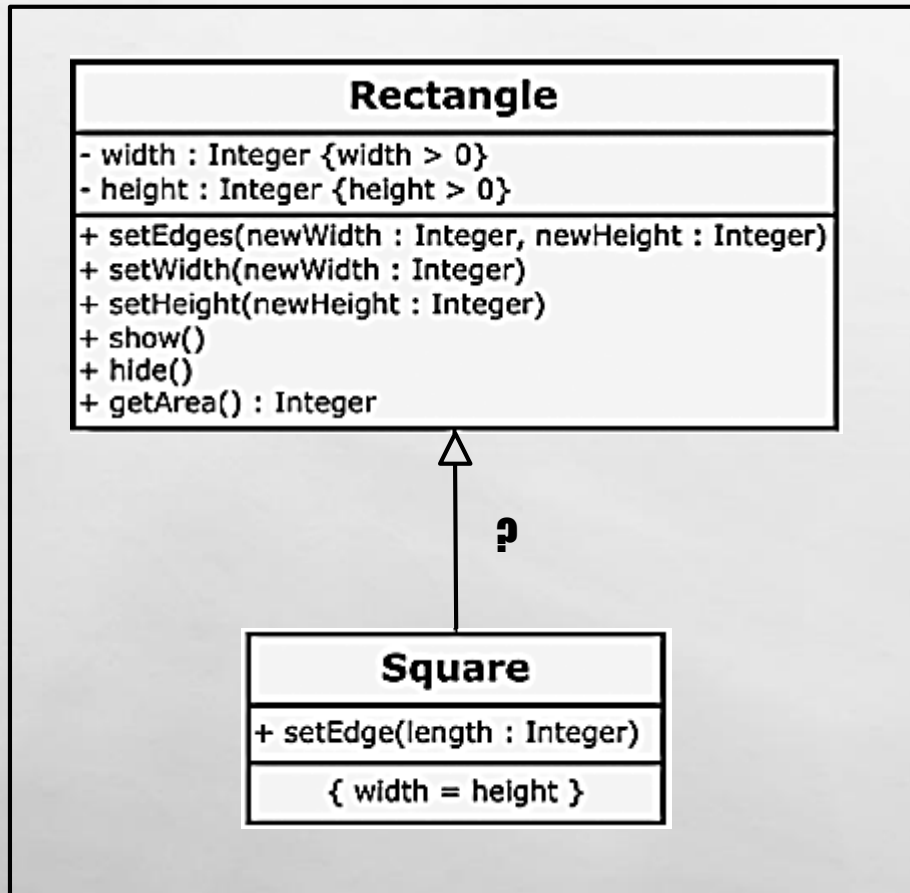
- 인터페이스와 추상 클래스를 사용해 구현이 미치는 영향을 격리해야 한다
 - 상세한 구현에 의존하는 클라이언트 클래스는 테스트가 어렵다
 - 의존관계 역전 원칙 (DIP - Dependency inversion principle)
: 클래스는 상세한 구현이 아니라 추상화에 의존 해야 한다는 원칙



Liskov Substitution Principle (LSP)

129

- 프로그램의 객체는 프로그램의 정확성을 깨뜨리지 않으면서 하위 타입의 인스턴스로 바꿀 수 있어야 한다 [LST: 리스코프 치환 원칙]



Interface Segregation Principle (ISP)

130

- 특정 클라이언트를 위한 인터페이스 여러 개가 범용 인터페이스 하나보다 낫다. **[ISP: 인터페이스 분리 원칙]**

```
class Bird {
public:
    virtual ~Bird() = default;
    virtual void fly() = 0;
    virtual void eat() = 0;
    virtual void run() = 0;
    virtual void tweet() = 0;
};

class Sparrow : public Bird {
public:
    virtual void fly() override {
        //...
    }
    virtual void eat() override {
    }
    virtual void run() override {
    }
    virtual void tweet() override {
    }
};
```

```
class Penguin : public Bird {
public:
    virtual void fly() override {
        // ???
    }
    //...
};
```

Interface Segregation Principle (ISP)

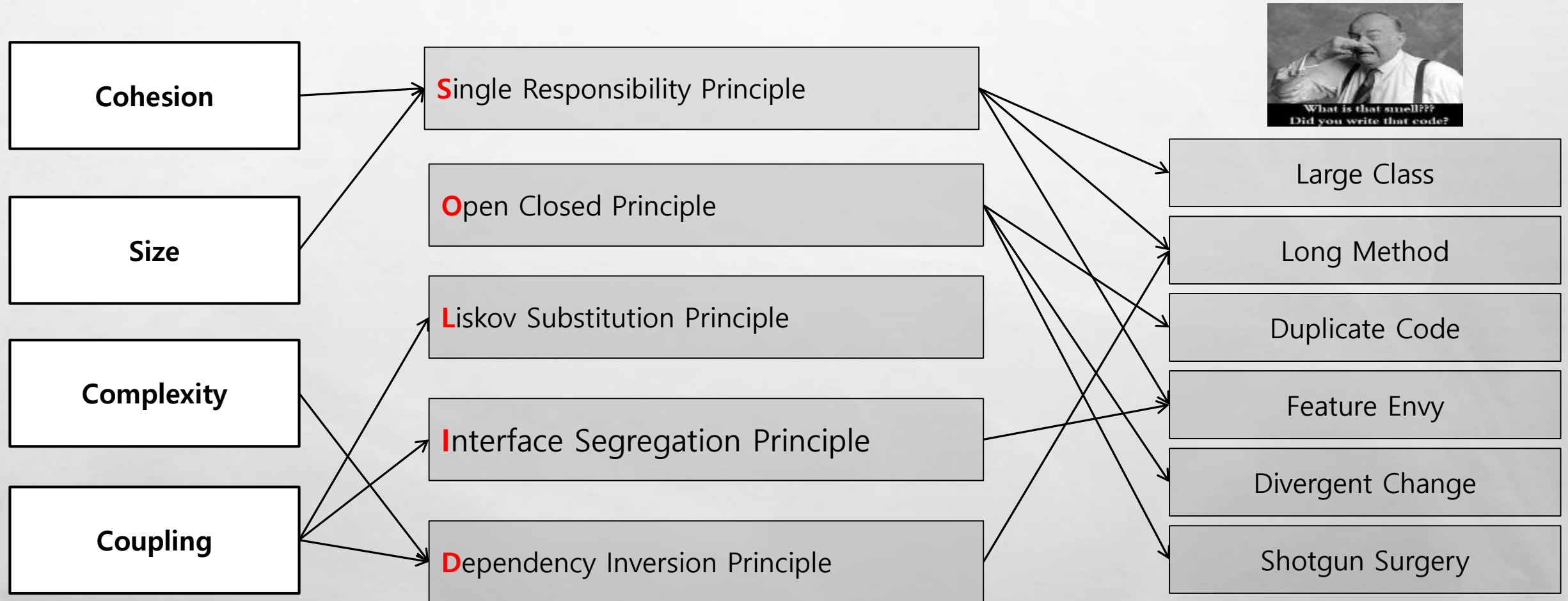
131

//role interfaces

```
class Lifeform {  
public:  
    virtual void eat() = 0;  
    virtual void move() = 0;  
};  
class Flyable {  
public:  
    virtual void fly() = 0;  
};  
class Audible {  
public:  
    virtual void makeSound() = 0;  
};
```

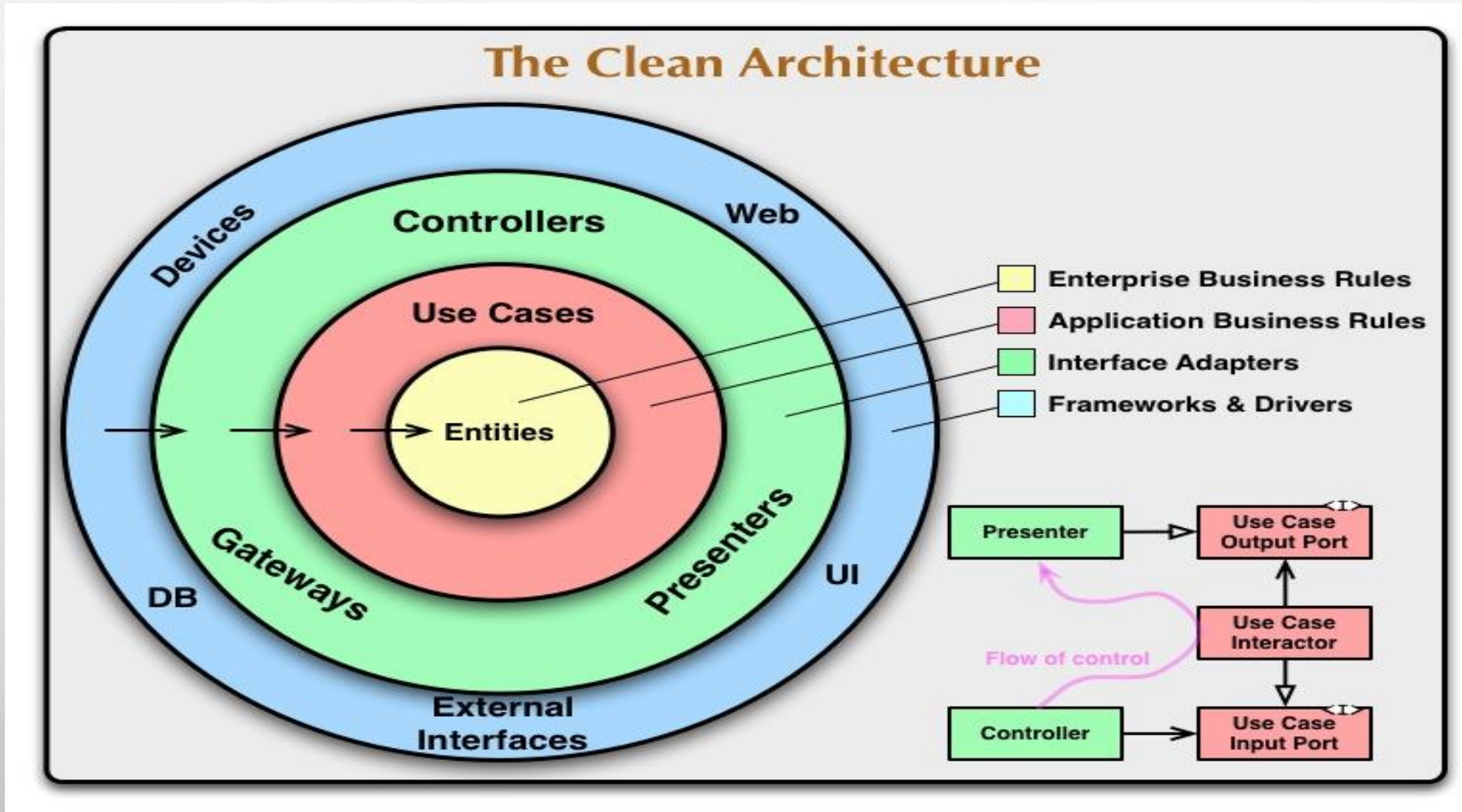
```
class Sparrow : public Lifeform, public Flyable,  
public Audible {  
    //...  
};  
  
class Penguin : public Lifeform, public Audible {  
    //...  
};
```

Design Concepts, SOLID and Bad Smells



Clean Architecture

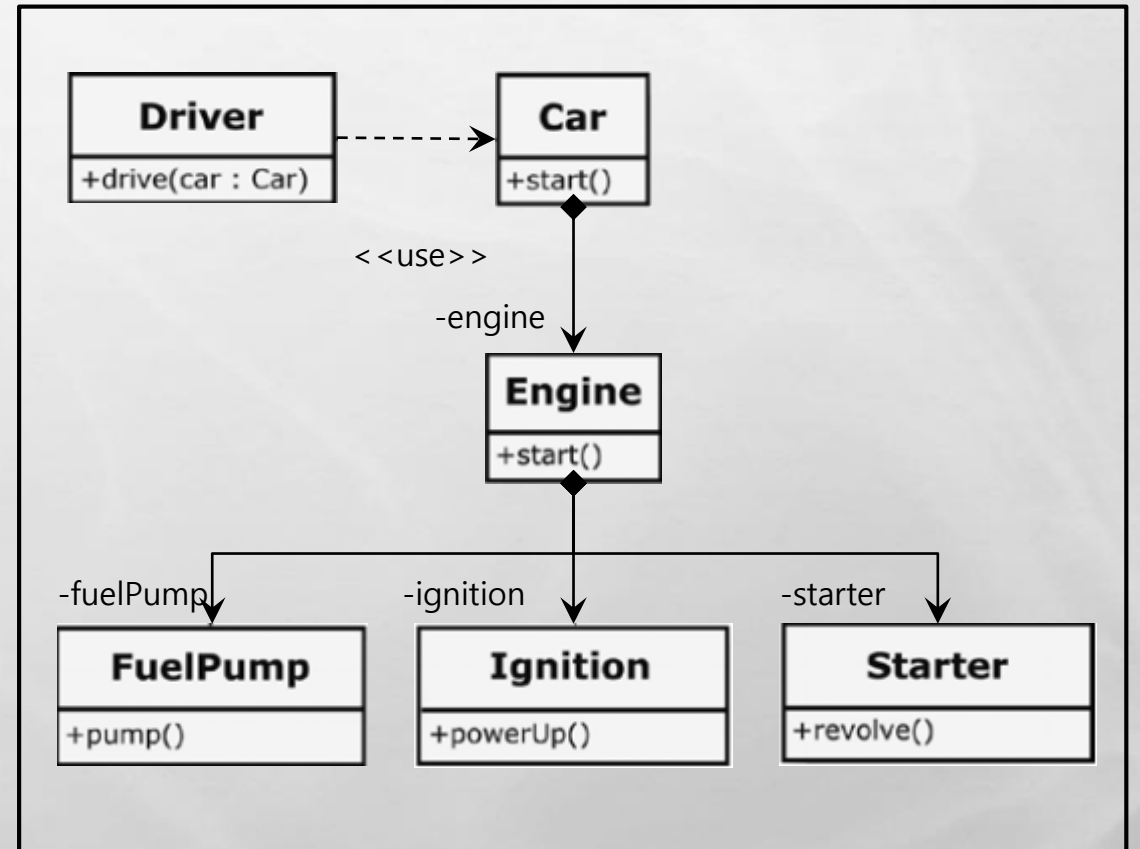
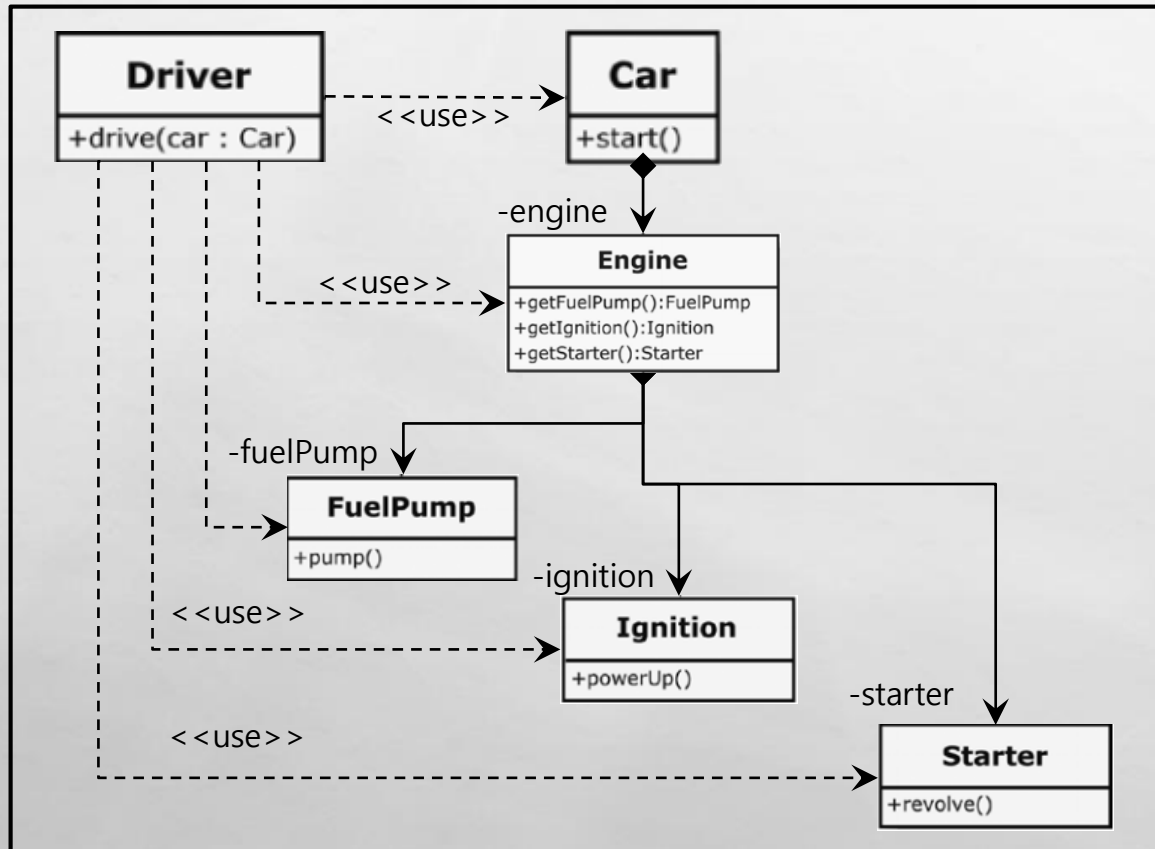
- <https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html>



Don't Talk to Strangers (Law of Demeter)

134

- 객체는 그것이 내부적으로 보유하고 있거나 메시지를 통해 확보한 정보만 가지고 의사 결정을 내려야 한다
 - 최소한의 정보만 접근 및 보유한다.
 - 느슨하게 결합한다.



Don't Talk to Strangers (Law of Demeter) - Tell, Don't Ask!

135

```
class Engine {
public:
// ...
void start() {
    if (! fuelPump.isRunning()) {
        fuelPump.powerUp();
        if (fuelPump.getFuelPressure() < NORMAL_FUEL_PRESSURE) {
            fuelPump.setFuelPressure(NORMAL_FUEL_PRESSURE);
        }
    }
    if (! ignition.isPoweredUp()) {
        ignition.powerUp();
    }
    if (! starter.isRotating()) {
        starter.revolve();
    }
    if (engine.hasStarted()) {
        starter.openClutchToEngine();
        starter.stop();
    }
}
// ...
```

```
// ...
private:
    FuelPump fuelPump;
    Ignition ignition;
    Starter starter;
    static const unsigned int NORMAL_FUEL_PRESSURE { 120 };
};
```


Don't Talk to Strangers (Law of Demeter) - Tell, Don't Ask!

136

```
class Engine {
public:
// ...
    void start() {
        fuelPump.pump();
        ignition.powerUp();
        starter.revolve();
    }

// ...
private:
    FuelPump fuelPump;
    Ignition ignition;
    Starter starter;
};
```

```
class FuelPump {
public:
// ...
    void pump() {
        if (! isRunning) {
            powerUp();
            setNormalFuelPressure();
        }
    }

// ...
private:
    void powerUp() {
        //...
    }
    void setNormalFuelPressure() {
        if (pressure != NORMAL_FUEL_PRESSURE) {
            pressure = NORMAL_FUEL_PRESSURE;
        }
    }
    bool isRunning;
    unsigned int pressure;
    static const unsigned int NORMAL_FUEL_PRESSURE { 120 };
};
```


- Anemic class: procedural programming with data structures
 - Object-oriented model 과 대응 되는 Anemic domain model

```
class Customer {  
public:  
    void setId(const unsigned int id);  
    unsigned int getId() const;  
    void setForename(const std::string& forename);  
    std::string getForename() const;  
    void setSurname(const std::string& surname);  
    std::string getSurname() const;  
    //...more setters/getters here...  
private:  
    unsigned int id;  
    std::string forename;  
    std::string surname;  
    // ...more attributes here...  
};
```

- Utility class => huge "god class"
 - 많은 static member functions들을 포함.
 - Hard wired
 - 객체지향보다는 절차적 프로그램을 향함
 - My advice is to avoid static member variables respectively and member functions largely
예외) private constants of a class

■ 참고 도서

- 클린 코드 , 로버트 C. 마틴, 인사이트
- Effective Modern C++, 스콧 마이어스, 인사이트
- Clean C++ Sustainable Software Development Patterns and Best Practices with C++ 17, Stephan Roth

■ 추천 자료

- C++ Core Guidelines
 - <http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>
- Google C++ Style Guide
 - <https://google.github.io/styleguide/cppguide.html>

THANK YOU.

별첨

■ Robert C. Martin (Uncle Bob)

- 오브젝트 멘토(Object Mentor Inc.)의 창립자이자 대표
- 객체 지향 설계, 패턴, UML, Agile 방법론과 eXtreme Programming 컨설팅 분야 선두

■ Publications



- 클린 소프트웨어 (Agile Software Development: Principles, Patterns, and Practices, 2002)
- 클린 코드 (Clean Code: a handbook of agile software craftsmanship, 2009)
- 클린 코더 (The Clean Coder: a code of conduct for professional programmers, 2011)
- UML 실전에서는 이것만 쓴다 (UML for Java Programmers, 2003)



※ Jolt Award : Software Development Magazine에서 매년 컴퓨터계에서 뛰어난 "Things"에 대해 주는 상입니다.
Jolt는 해커들이 밤샘을 할 때, 카페인이 많이 포함되어 있어서 즐겨 마셨다는 "Jolt Cola"에서 따왔다고 합니다.

코드리뷰의 현실

- 코드리뷰를 어떻게 하는지 모름
- 개인업무가 바쁜데 남의 코드를 리뷰 할 시간이 없음
- 남의 코드를 리뷰 하는 행위가 거북하고 불편함
- 지적에 대하여 기분이 좋지 않음
- 도메인 지식이 없어 시간 대비 코드리뷰 질이 낮음
- 잘하는 사람에게 코드리뷰가 몰림
- 코드리뷰 안 하는 사람들이 많아짐
- 시작하고 얼마 안돼서 시스템에서 형식적으로 수행

코드리뷰가 어려운 이유

■ 완벽한 코드는 존재하지 않는다.

- 코드는 완성상태가 아니며 주위상황에 따라 변경될 수 있음 (HW와의 차이점)
- 결국, 완벽한 리뷰도 존재할 수 없으며 변화에 따라 끊임없는 분석이 필요함

■ 효율성 있는 코드리뷰의 명확한 기준과 방법이 정의하기가 쉽지 않다.

- Checklist가 상세하고 자세한 경우 코드리뷰에 투입되는 시간도 커질 수밖에 없음
- 개인별 개발능력이 상이하여 Checklist를 이해수준이 달라짐

■ 리뷰어의 도메인 지식의 수준이 상이하다.

- 도메인 지식이 없으면 리뷰가 사실상 어렵거나 의견을 제시하기 힘든 경우가 있음
- 특히 Defect 발견이 코드리뷰의 주목적이 되는 경우 리뷰는 더욱 어려워짐.

■ 개인의 특성이 상이함.

- 개인별 리뷰 포인트가 상이하여 팀이 원하는 결과를 효율적으로 얻기 어려움
- 이로 인해 팀원끼리의 분란이 생기고 teamwork이 떨어짐

S/W 개발 실력을 쌓기 위한 개인 활동

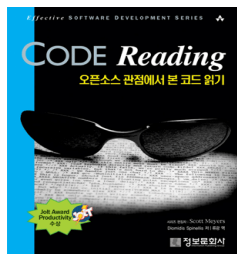
■ S/W 장인으로 성장하기 위한 개발자 스스로의 학습/참여/훈련 활동

- 코드 읽기, 오픈소스 참여, 코드 카타, 펫 프로젝트 등

코드 읽기 (Reading)

■ [학습] Code Reading

- 1) 좋은 코드를 작성하기 위해서
· 사전 학습용 책을 읽는 것과 같음
- 2) 협업 개발을 잘하기 위해서
· 타인의 코드를 읽어 이해하는 능력 ↑
- 3) 생각의 틀을 깰 수 있음
· 타인의 사고 방식, 구조, 논리 습득



‘프로그래밍의 기본은 다른 사람의 코드를 읽는 것부터 시작한다’ - “Code Reading 책 서문”



오픈 소스 활동

■ [참여] Open Source 기여

- 오픈 소스에는 전 세계 최고 수준의 개발자들이 참여하고 경험이 축적됨
- 단순사용 → 원리 분석 → 참여/기여
- 개발 역량 ↑ + 글로벌 인맥 형성



“시인이 되고 싶은 사람은 다른 사람들의 시를, 화가가 되고 싶은 사람은 다른 화가의 그림을 보고 배우는데, 왜 개발자들은 다른 분들의 코드를 보고 배우려 하지 않나요?”
- 오픈소스 개발자

코드 카타 (Code Kata)

■ [훈련] Code Kata

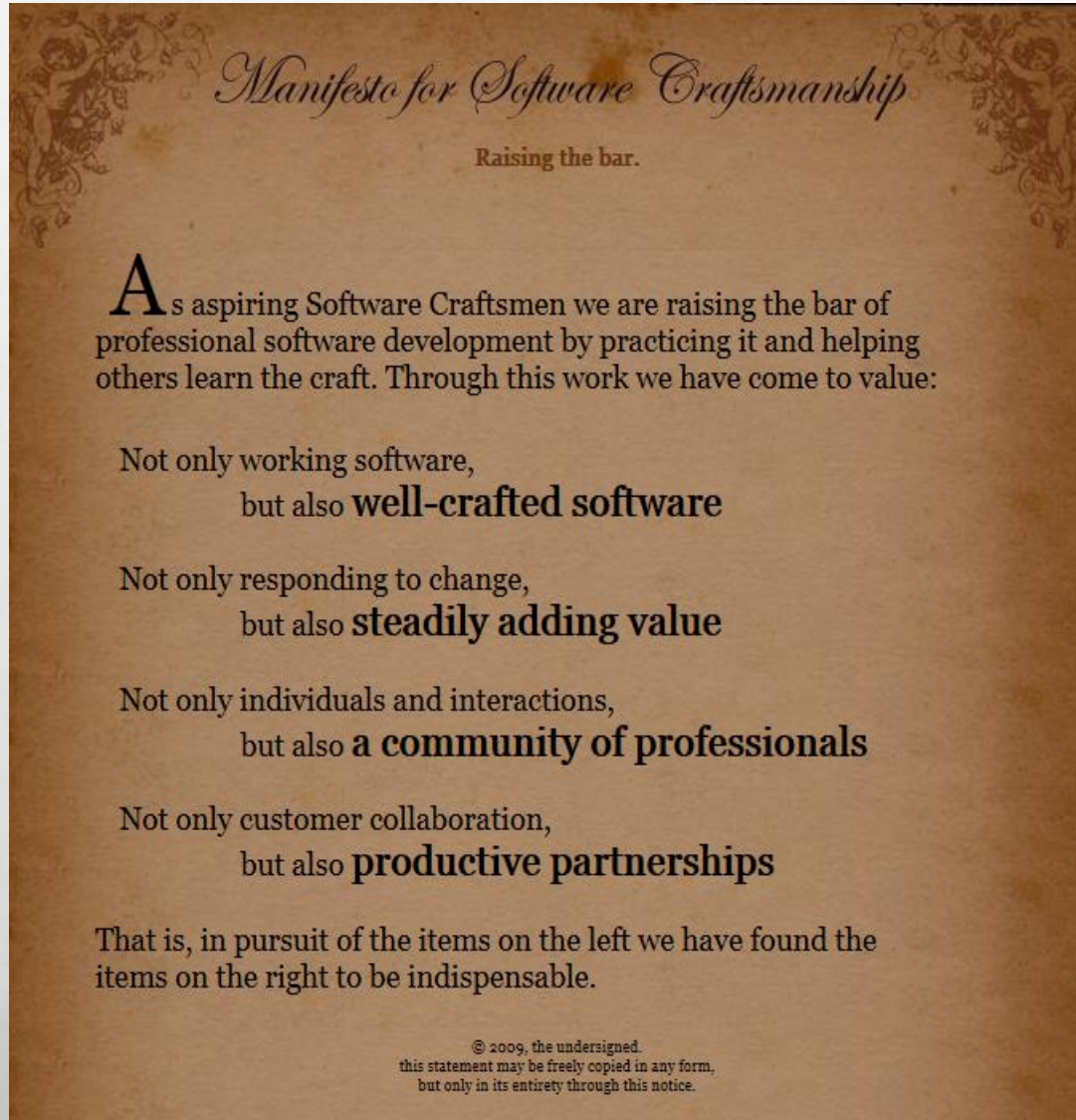
- 무술가들이 무술을 연마하듯 혹은 음악가들이 악기를 연습하듯, 같은 문제를 다양한 방법으로 접근하는 연습을 통해서 더 나은 프로그래머가 되자는 실천법
- 단순 매일 반복이 아닌 매 순간 “노력이 담긴 훈련”이 중요



※ 코드 카타 : 무술에서 정해진 움직임을 반복해서 연습하는 것을 프로그래밍의 영역으로 가져온 것

“Because experience is the *only* teacher”

Software Craftsmanship Manifesto



‘동작하는 소프트웨어뿐만 아니라, 정교하게 숨씨 있게 만들어진 작품들’

- 소스코드는 예측 가능하고, 유지보수 될 수 있으며, 수정이 두렵지 않은 상태여야 한다.
- 자동화 테스트를 통해서 짧은 시간 내에 잘못된 부분을 파악해야 한다.
- 테스트 주도 개발 / 단순한 디자인 / 비즈니스 용어로 표현된 코드

‘변화에 대응하는 것뿐 아니라, 계속해서 가치를 더하는 것을’

- 코드를 깔끔하게 정리하고, 구조를 개선하고, 확장성을 높이고, 테스트를 가능하게 하고, 쉽게 유지 보수 할 수 있게 한다.
- “캠핑장소를 처음 발견했을 때보다 더 깨끗하게 남겨두라.” – 보이스카웃
- “같은 일을 반복하면서 다른 결과를 기대하는 것은 미친 짓이다.” – 앨버트 아인슈타인

‘개별적으로 협력하는 것뿐만 아니라, 프로페셔널 커뮤니티를 조성하는 것을’

- 블로그 포스팅 / 오픈 소스 프로젝트 기여 / 작성한 코드 공개 / 지역 커뮤니티 참여 / 페어 프로그래밍 등

‘고객과 협업하는 것뿐만 아니라, 생산적인 동반자 관계를’

- 소프트웨어 장인은 적극적으로 프로젝트의 성공에 기여해야 한다.
- 요구사항에 질문하고, 비즈니스를 이해하고, 개선사항을 제안하고, 생산적인 동반자 관계를 맺는다.
- 코드와 관련된 일이 아니면 나의 일이 아니라고 생각하면 진정한 장인이 아니다.