소프트웨어 개선 그룹(SIG) 개발 원칙

“ 컴퓨터가 이해할 수 있는 코드는 바보라도 짤 수 있다.

유능한 프로그래머는 인간이 이해할 수 있는 코드를 짠다.”

개발이든 일이든 혼자하는 것이 아니라 같이 하는 것

참고도서 : 데이터 중심 애플리케이션 설계
유지보수 가능한 코딩 기술
참고 : https://www.refactoring.com/catalog/, https://refactoring.guru/refactoring

Hong Hyo Sang

Contents

I. 시스템을 설계 시 고려사항

1. Reliabilty(신뢰성), Scalabilty(확장성)
2. Maintainability(유지보수성)

II. 소프트웨어 개선 그룹(SIG) 원칙

1. 코드 단위를 짧게 하라
2. 코드 단위는 간단하게 짜라
3. 코드는 한번만 작성하라
4. 단위 인터페이스를 작게 하라
5. 관심사를 모듈로 분리 및 그외 원칙

Content

I. 일반개념

1. Reliabilty(신뢰성), Scalabilty(확장성)
2. Maintainability(유지보수성)

1. Reliabilty(신뢰성), Scalabilty(확장성)

신뢰성은 무언가 잘못되더라도 지속적으로 올바르게 동작함을 의미하며 확장성은 시스템에 부하가 증가할 때 대처가 가능한 시스템을 의미 함.

01. 신뢰성 (Reliabilty)

- **하드웨어 결함**
  - 하드디스크 고장, 램 결함, 정전 등
  - 일반적으로 하드웨어의 구성 요소를 중복(redundancy)을 추가하여 대응
    - 디스크 RAID 구성
    - 이중 전원 디바이스 사용
    - 예비 전원용 디젤 발전기 사용

- **소프트웨어 오류**
  - 커널 버그, 공유 자원을 과도하게 쓰는 프로세스 영향 등
  - **빈틈 없는 테스트, 프로세스 격리, 모니터링 등을 이용하여 대응**

- **인적 오류**
  - 하드웨어 결함에 비해 훨씬 높은 비율
  - **잘 설계된 추상화, 샌드박스 사용, 단위 테스트부터 통합 테스트까지 철저하게 테스트 등을 이용해 방지**

02. 확장성 (Scalabilty)

- **부하 매개변수**
  - 웹 서버의 초당 요청 수
  - 데이터베이스의 읽기 대 쓰기 비율
  - 동시 사용자 수

- **scaling up / scailing out**

참고 : 도서 : 데이터 중심 애플리케이션 설계

2. Maintainability(유지보수성)

소프트웨어 비용은 대부분은 초기 개발비 보다 유지보수 비용이 발생 함.

01. 유지보수들 위한 3가지 소프트웨어 설계 법

- **운영성 (Operability)**
  - 동일하게 반복되는 태스크를 쉽게 수행할 수 있어야 한다. 즉 운영이 쉬워야 함.
- **단순성 (Simplicity)**
  - 시스템을 단순하게 만드는 것으로 좋은 추상화를 통해 복잡한 세부 구현을 숨김으로써 시스템 복잡도를 줄이는 것
- **발전성 (Evolvability)**
  - 시스템에 변화하는 요구사항을 잘 수용하고 복잡한 시스템을 보다 쉽게 수정할 수 있는 민첩성

02. Software 품질은 국제 표준 ISO/IEC 25010:2011의 8가지 특성

- 기능 적합성 (Functional suitability), 수행 효율성 (Performance efficiency), 호환성 (Compatibility), 사용성 (Usability), 신뢰성 (Reliability), 보안 (Security), 유지보수성 (Maintainability), 이식성 (Portability) 수정할 수 있는 민첩성

03. 유지보수성 (Maintainability)

- 모듈성 (Modularity) : 외부에 대하여 최소의 영향을 가진 개별 구성 요소로 시스템(SW)이 구성된 정도
- 재사용성 (Reusability) : 자산(모듈)이 한 개 이상의 시스템에서 사용될 수 있거나, 다른 자산에 구축할 수 있는 정도
- 분석성 (Analyzability) : 제품(시스템)의 문제를 식별하고, 고장의 원인을 진단하고 변경사항을 반영하기 위하여 수정하여야 하는 부분을 식별하기 쉬운 정도
- 수정 가능성 (Modifiability) : 제품(시스템)을 수정할 때, 기존 제품의 품질을 저하시키거나 장애를 발생시키지 않으면서 효과적이고 효율적으로 수정할 수 있는 정도
- 시험 가능성 (Testability) : 제품(시스템, 구성요소)을 검증한 근거가 충분한지를 확인할 수 있는 정도
- 이식성 (Portability) : 제품(시스템, 구성요소)이 다른 다양한 환경(SW/HW/Network) 등으로의 전환이 용이한 정도
- 적합성 (Adaptability) : 제품(시스템)을 다른 SW(HW)나 사용환경에 효과적이고 효율적으로 적용할 수 있는 정도
- 설치 가능성 (Installability) : 제품(시스템)이 성공적으로 설치/제거될 수 있는 정도
- 대치성 (Replaceability) : 제품이 동일한 환경에서 동일한 목적을 위해 다른 지정된 SW 제품으로 대체될 수 있는 정도

참고 : 도서 : 데이터 중심 애플리케이션 설계, https://devforperson.tistory.com/entry/ISO-25010-%ED%92%88%EC%A7%88-%ED%8A%B9%EC%84%B1

2. Maintainability(유지보수성)

소프트웨어 비용은 대부분은 초기 개발비 보다 유지보수 비용이 발생 함.

04. 유지보수 행위

분류기준	종 류	내 용
사유에 의한 유지보수	교정 유지보수 (Corrective Maintenance)	- 오류로 인한 유지보수 발생 - 처리오류, 수행오류, 구현오류 등의 식별과 처리
	적응 유지보수 (Adaptive Maintenance)	- 데이터 환경과 인프라 환경 변화 적응을 위한 처리
	완전화 유지보수 (Perfective Maintenance)	- 새로운 기능 추가, 변경, 품질을 위한 유지보수
시간에 의한 유지보수	계획 유지보수 (Scheduled Maintenance)	- 주기적인 유지보수
	예방 유지보수 (Preventive Maintenance)	- 미리 예방차원의 유지보수 예) 보안관련
	응급 유지보수 (Emergent Maintenance)	- 유지보수의 사후 승인 필요 시
대상에 의한 유지보수	데이터/프로그램 보수	- 데이터의 Conversion 등 필요 시 처리
	문사화 유지보수	- 문서표준의 변경이나 기타 필요 시 처리
	시스템 유지보수	

05. 유지보수 원칙

- **단순한 가이드 라인을 지키기만 해도 유지보수성은 좋아진다.**
- **유지보수성은 나중으로 미룰 문제가 아니라 개발 프로젝트 시작 단계부터 반드시 염두에 두고 하나씩 실천해야 한다.**
  - 개발 가이드 라인 충실히 이행 -> 깨진 창문 효과 방지
- **가이드라인을 충실히 잘 따를수록 소프트웨어 시스템의 유지보수성이 좋아 진다.**

참고 : http://i-bada.blogspot.com/2012/07/blog-post_3628.html

Content

II. 소프트웨어 개선 그룹(SIG) 원칙

1. 코드 단위를 짧게 하라
2. 코드 단위는 간단하게 짜라
3. 코드는 한번만 작성하라
4. 단위 인터페이스를 작게 하라
5. 관심사를 모듈로 분리하라
6. 그외 원칙

1. 코드 단위를 짧게 하라

메소드 단위는 15라인을 넘어가지 않게 작성해야 해당 메소드를 이해(분석)하고, 테스트하고, 재사용하기 쉬워 진다.

01. 메소드 추출 - 긴 단위의 프로그램을 하나의 단위로 메소드화 한다.

메서드에 행이 많이 발견될 수록 메서드가 하는 일을 파악하는 것이 더 어려워 진다.

```
public void test() {
   if (true) {
      log.debug("step 1");
   }

   if(!actionA()) {
      for ( .... ) {
         action ....
      }
   }

   if(!actionB()) {
      for ( .... ) {
         action ....
      }
   }

}
```

```
public void test() {
   if (true) {
      log.debug("step 1");
   }
   actionA_B()
  }

private void actionA_B() {
   if(!actionA()) {
      for ( .... ) {
         action ....
      }
   }

   if(!actionB()) {
      for ( .... ) {
         action ....
      }
   }

}
```

```
public void test() {
   if (true) {
      log.debug("step 1");
   }
   actionA_B()
  }

private void actionA_B() {
   runActionA();
   runActionB();
}

private void runActionA() {
   if(!actionA()) {
      for ( .... ) {
         action ....
      }
   }
}

private void runActionB() {
   if(!actionB()) {
      for ( .... ) {
         action ....
      }
   }
}
```

전체 코드의 볼륨은 커지지만 단위를 짧게 하므로 분석할 대상은 복잡도는 줄어 든다.

1.더 읽기 쉬운 코드
2.적은 코드 중복
3.코드의 독립적인 부분을 분리함으로 서, 오류의 발생 가능성이 적어짐

02. 메소드를 메소드 객체로 대체 - 긴 메소드가 있는데, 지역변수 때문에 Extract Method를 적용할 수 없는 경우에는 메소드를 그 자신을 위한 객체로 바꿔서 모든 지역변수가 그 객체의 필드가 되도록 한다. 이렇게 하면 메소드를 같은 객체 안의 여러 메소드로 분해할 수 있다.

참고 : 마틴 파울러의 리팩토링

03. 생각해 볼 문제 – 유지보수를 생각 해야 함

- **단위를 많이 나뉘면 메소드 호출이 잦아서 성능이 떨어 짐**
  - 이론적으로는 호출이 많아 지므로 성능에 영향을 주지만 기껏해야 밀리 초 단위
  - 수백만 개의 단위의 Loop가 아닌 이상 성능에 불이익이 없음 (JVM이 최적화함)
- **코드를 여러 단위로 나뉘면 해석이 어려움**
  - 다른 개발자가 코드를 분석 하는 경우 쉽게 읽고 이해 할 수 있는 코드는 ?
- **단위를 나눈다고 나아지는 것이 없음**
  - 각 단위가 어떤 일을 하고 있는지 기능을 나누고 이름을 부여 하여 읽고 이해가 쉬운 코드

*** SIG 단위 크기 기준**

메서드 코드 라인 수	별 4개를 받기 위한 기준치
60 이상	최대 6.9% 까지
30 이상	최대 22.3% 까지
15 이상	최대 43.7% 까지
15 미만	최소한 56.3%

2. 코드 단위는 간단하게 짜라

결정 포인트가 적을수록 단위 별 수정 및 테스트가 쉬워 진다. (단위 당 분기점은 4개 이하로 제한하고 복잡한 단위는 더 잘게 나누어 서로 뭉쳐 있지 않게 하면 단위 수정 및 테스트가 쉬어 진다.)

01. 순환 복잡도를 작게 하라 – 분기점이 많으면 테스트 CASE가 많아짐

- **분기 커버리지** : 단위 분기점 개수는 그 단위의 모든 분기점으로 만들어진 경로를 모두 커버하기 위해 최소한의 경로 개수
- **조합 효과** : 한 단위의 처음 부터 끝까지 모든 경로를 계산하다 보면 순서 문제로 발생 하는 문제
- **단위 실행 경로** : 단위를 지나는 경로의 최대 수
- **순환 복잡도 (맥캐브 복잡도)** : 분기점 + 1 으로 "매케브 복잡도를 5로 제한 하라" 이면 단위를 커버하는데 필요한 최소한의 테스트 개수를 뜻한다.

```
switch (a) {
  case A :
    ...
    break;
  case B :
    ...
    break;
  case C :
    ...
    break;
  case D :
    ...
    break;
  case E :
    ...
    break;
default :
    ...
    break;
}
```

순환 복잡도 : 6
Case 문이 추가 될 때 마다 복잡도는 증가 함

- **JAVA 분기점**
  - if,
  - case,
  - ?,
  - &&, ||
  - while
  - for
  - catch

If-then-else 문이 중첩 되어 있으면
- **복잡도는 증가**
- **테스트 케이스 개수 증가**
- **분석하기 어려워 짐**

02. 조건문 단순화

- **조건부 분해** : 복잡한 조건문을 조건부의 복잡한 부분을 메서드로 분해
- **조건식 통합** : 동일한 결과 또는 조치로 이어지는 여러 조건이 있으면 모든 조건을 단일 표현식으로 통합
- **중복 된 조건부 조각 통합** : 조건부의 모든 분기에서 동일한 코드가 있으면 외부로 코드 이동
- **제어 플래그 제거** : 부울 표현식으로 제어 플래그를 사용 한다면 변수 대신 break, continue및을 사용. 이동
- **중첩 된 조건을 보호 절로 바꾸기** : 중첩 된 조건부 그룹이 있으면 정상적인 코드 실행 흐름을 결정 하기 어려운 경우 모든 검사와 CASE를 별도로 분리 하여 주요 검사 앞에 코딩 함,
- **조건부를 다형성으로 바꾸기** : 개체 유형 또는 속성에 따라서 다양한 작업을 하는 경우 조건부 분기와 일치 하는 하위 클래스를 만든다. 객체 클래스에 따라 다형성을 통해 적절한 구현

03. 생각해 볼 문제 – 유지보수를 생각 해야 함

- **메서드를 나눈다고 복잡도가 줄어 들지 않는다**
  - 메서드를 나눈다고 복잡도가 줄어 들지는 않지만 기능과 CASE를 이해 하고 쉽게 명명한, 많아야 분기점 4새 이하인 간단한 코드로 코딩 하면 테스트가 쉽고 읽기 편하고 단위 테스트를 통해 실패한 테스트의 근본 원인을 더 쉽게 밝혀 낼 수 있음.

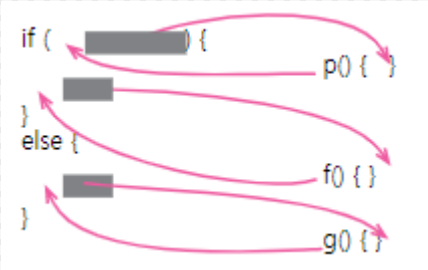
*** SIG 단위 복잡도 기준**

메서드 코드 라인 수	별 4개를 받기 위한 기준치
> 맥캐브 25 이상	최대 1.5% 까지
> 맥캐브 10 이상	최대 10.0% 까지
> 맥캐브 5 이상	최대 25.2% 까지
<= 맥캐브 5 미만	최소한 74.8%

2. 코드 단위는 간단하게 짜라

결정 포인트가 적을수록 단위 별 수정 및 테스트가 쉬워 진다. (단위 당 분기점은 4개 이하로 제한하고 복잡한 단위는 더 잘게 나누어 서로 뭉쳐 있지 않게 하면 단위 수정 및 테스트가 쉬어 진다.)

01. 조건부 분해 (Decompose Conditional)




```
if (date.before(SUMMER_START) || date.after(SUMMER_END)) {
  charge = quantity * winterRate + winterServiceCharge;
}
else {
  charge = quantity * summerRate;
}
```

```
if (isSummer(date)) {
  charge = summerCharge(quantity);
}
else {
  charge = winterCharge(quantity);
}
```

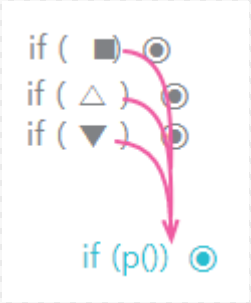
*** 리팩터링하는 방법 ***

1. Extract Method 를 통해 조건문을 별도의 방법으로 추출
2. then및 else블록에 대해 프로세스를 반복 .

2. 코드 단위는 간단하게 짜라

결정 포인트가 적을수록 단위 별 수정 및 테스트가 쉬워 진다. (단위 당 분기점은 4개 이하로 제한하고 복잡한 단위는 더 잘게 나누어 서로 뭉쳐 있지 않게 하면 단위 수정 및 테스트가 쉬어 진다.)

01. 조건부 통합 (Consolidate Conditional Expression)




```
if (anEmployee.seniority < 2) return 0;
if (anEmployee.monthsDisabled > 12) return 0;
if (anEmployee.isPartTime) return 0;
```

⇩

```
if (isNotEligableForDisability()) return 0;

private boolean isNotEligableForDisability() {
   return ((anEmployee.seniority < 2)
           || (anEmployee.monthsDisabled > 12)
           || (anEmployee.isPartTime));
}
```

*** 리팩터링하는 방법 ***

1. 단일 표현의 조건문을 통합 and하고 or. 통합 :
   - 중첩 된 조건문은 and
   - 연속 조건문은 결합 or

2. 연산자 조건에 대해 Extract Method 를 수행 하고 표현식의 목적을 반영하는 메서드에 호출

2. 코드 단위는 간단하게 짜라

결정 포인트가 적을수록 단위 별 수정 및 테스트가 쉬워 진다. (단위 당 분기점은 4개 이하로 제한하고 복잡한 단위는 더 잘게 나누어 서로 뭉쳐 있지 않게 하면 단위 수정 및 테스트가 쉬어 진다.)

01. 중복 된 조건부 조각 통합 (Consolidate Duplicate Conditional Fragments)

```
if (isSpecialDeal()) {
  total = price * 0.95;
  send();
}
else {
  total = price * 0.98;
  send();
}
```

```
if (isSpecialDeal()) {
  total = price * 0.95;
}
else {
  total = price * 0.98;
}
send();
```

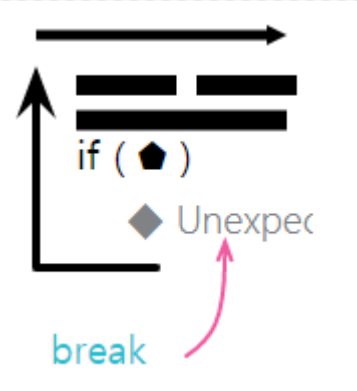
*** 리팩터링하는 방법 ***

1. 중복 코드 함수 생성

2. 코드 단위는 간단하게 짜라

결정 포인트가 적을수록 단위 별 수정 및 테스트가 쉬워 진다. (단위 당 분기점은 4개 이하로 제한하고 복잡한 단위는 더 잘게 나누어 서로 뭉쳐 있지 않게 하면 단위 수정 및 테스트가 쉬어 진다.)

01. 제어 플래그 제거 (Replace Control Flag with Break)



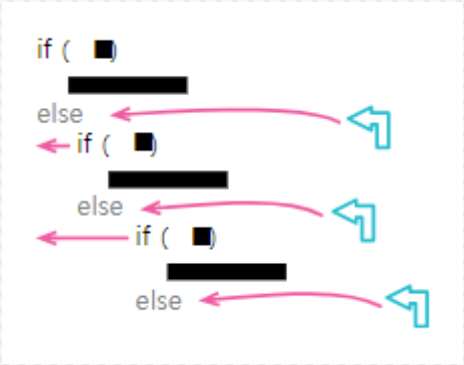

```
for (const p of people) {
  if (! found) {
    if ( p === "Don") {
      sendAlert();
      found = true;
    }
```

```
for (const p of people) {
  if ( p === "Don") {
    sendAlert();
    break;
  }
```

2. 코드 단위는 간단하게 짜라

결정 포인트가 적을수록 단위 별 수정 및 테스트가 쉬워 진다. (단위 당 분기점은 4개 이하로 제한하고 복잡한 단위는 더 잘게 나누어 서로 뭉쳐 있지 않게 하면 단위 수정 및 테스트가 쉬어 진다.)

01. 중첩 된 조건을 보호 절로 바꾸기 (Replace Nested Conditional with Guard Clauses)



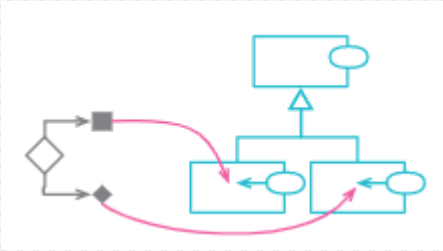

```
public double getPayAmount() {
  double result;
  if (isDead){
    result = deadAmount();
  }
  else {
    if (isSeparated){
      result = separatedAmount();
    }
    else {
      if (isRetired){
        result = retiredAmount();
      }
      else{
        result = normalPayAmount();
      }
    }
  }
  return result;
}
```

```
public double getPayAmount() {
  if (isDead){
    return deadAmount();
  }
  if (isSeparated){
    return separatedAmount();
  }
  if (isRetired){
    return retiredAmount();
  }
  return normalPayAmount();
}
```

2. 코드 단위는 간단하게 짜라

결정 포인트가 적을수록 단위 별 수정 및 테스트가 쉬워 진다. (단위 당 분기점은 4개 이하로 제한하고 복잡한 단위는 더 잘게 나누어 서로 뭉쳐 있지 않게 하면 단위 수정 및 테스트가 쉬어 진다.)

01. 조건부를 다형성으로 바꾸기



```
class Bird {
  // ...
  double getSpeed() {
    switch (type) {
      case EUROPEAN:
        return getBaseSpeed();
      case AFRICAN:
        return getBaseSpeed() - getLoadFactor() * numberOfCoconuts;
      case NORWEGIAN_BLUE:
        return (isNailed) ? 0 : getBaseSpeed(voltage);
    }
    throw new RuntimeException("Should be unreachable");
  }
}
```

```
private static final Map<Nationallity, Flag> FLAGS = new HashMap<>();

static {
 FLAGS.put(EUROPEAN, new European();
 FLAGS.put(AFRICAN, new African();
 FLAGS.put(NORWEGIAN_BLUE, new NorwegianBlue();
}

public double getSpeed(Nationallity nationallity)  {
  Flag flag = FLAGS.get(nationallity);
  flag = flag != null ? Flag : new DefaultNational();
  return flag.getSpeed();
}
```

```
public interface Bird {
   double getSpeed();
}

class European implements Bird {
  double getSpeed() {
    return getBaseSpeed();
  }
}
class African implements Bird {
  double getSpeed() {
    return getBaseSpeed() - getLoadFactor() * numberOfCoconuts;
  }
}
class NorwegianBlue implements Bird {
  double getSpeed() {
    return (isNailed) ? 0 : getBaseSpeed(voltage);
  }
}
```

3. 코드는 한번만 작성하라

코드를 복사하면 여러 곳에서 버그를 고쳐야 하기 때문에 비효율적이고 에러가 나기 쉬우므로 중복 코드의 경우 생성하지 않는다.

01. Copy & Paste - 사본을 하나씩 손봐야 할 일이 생기면 전체 사본을 죄다 뒤져봐야 한다.

- **절대 하지 말아야 한다.**
- **분석하기 어렵다. :** 여러 곳에 산재 됨
- **고치기가 어렵다 :** 여러 곳에 산재 됨
- 범위 : 같은 메서드, 같은 클래스의 다른 메소드, 같은 코드 베이스에 있는 다른 클래스의 클래스 메서드 포함

02. 복사 유형

- 적어도 6라인 이상의 코드가 동일 할 때
- 라인 수는 공백, 주석 제외

Type 1 Clone : 6라인 Clone 하나

```
public void setName(String name) {
   this.name = name;
}

public void setName(String name) {
   this.name = name;
}
```

Type 2 Clone :

```
public void setName(String name) {
   String lastName = "길자";
   this.name = name + lastName ;
   …
}

public void setName(String name) {
   String lastName = "감자";
   this.name = name + lastName ;
   …
}
```

03. 적용 방법

- **유틸리티 클래스 :** 중복된 code를 메서드로 분리 하고 유틸리티 클래스 static 으로 생성
- **상위 클래스 :** 중복된 code를 메서드로 분리 하고 상위 클래스에 만든 후 extends를 통해서 상속

04. 생각해 볼 문제 – 유지보수를 생각 해야 함

- **다른 코드 베이스의 코드를 복사하는 건 문제 없음**
  - 맞는 말 이다. 그러나 다음과 같은 상황에서는 고려 하여야 한다.
  - 1. 현재 개발 중인 소스
  - 2. 다른 코드 베이스가 유지 보수 되지 않는 상태에서 리빌드 중인 소스
- **복사한 코드를 조금씩 고쳐 쓴다.**
  - 코드 변형을 명확히 하고 독립적이면서 테스트 가능한 형태로 개발
- **이 코드는 절대, 절대 바뀔 일이 없다.**
  - 여러 요구 사항에 의해서 변경이 된다. 향후 사고를 인정 하는 행위 임

*** SIG 중복 기준**

코드로 분류한 결과	별 4개를 받기 위한 기준치
중복 아님	95.4% 이상
중복	4.6% 이하

4. 단위 인터페이스를 작게 하라

파라미터는 적은 단위(4개 이하)는 이해하기 쉽고, 테스트, 재사용하기 편리하다. 4개 이상이면 파라메터 객체로 분리

*** SIG 단위 인터페이스**

메서드 파라메터 개수	별 4개를 받기 위한 기준치
7개 이상	0.7 % 이하
5~6 개	2.7 % 이하
3~4 개	13.8 % 이하
2개 이하	86.2 % 이하

5. 관심사를 모듈로 분리하라

느슨하게 결합된 모듈(클래스)이 고치기 쉽고 내용을 미리 살피고 실행할 수 있다.

01. 모듈 간 결합을 느슨하게 하기 위해 큰 모듈은 삼가 한다.

- 모듈이란 자바에서 클래스를 의미 한다.
- 개별 모듈로 나누어 개발 하고 구현 상세는 인터페이스 안으로 감춘다.
- 결합도가 높은 시스템은 사고가 난다.
- **하나의 클래스에 많은 기능을 하는 메소드가 있다는 것은 관심사를 적절하게 분리 하지 못 했기 때문이다. 그러므로 향 후 낮은 숙련도의 개발자는 겁을 먹고 손을 대지 못 한다.**
- **결합은 소스 코드의 클래스 수준에서 발생**
  : 다른 클래스와의 강한 결합을 유발 하는 것은 해당 클래스에 있는 메서드의 조합
- **클래스를 나눈 다고 호출 횟수까지 줄어 들지는 않지만 자주 호출 되는 클래스의 크기는 작아야 한다.**

02. 관심사를 모듈로 분리

- **단일 책임의 원칙** : 클래스 당 한 가지 일만 담당
- **특정 구현부는 인터페이스 안에 숨김** : 결합도를 줄이려면 각자에 맞는 기능을 인터페이스로 정의
- **커스텀 코드를 서드파티 라이브러리/프레임 워크 대체** : StringUtils, FileUtils 등 .. 아파치 커먼스, 구글 구아바 등 ….

*** Java Interface는 적어도 2개 이상의 클래스가 구현 하는 것이 좋음**

6. 그외 원칙

그외 원칙.

06. 코드 베이스를 작게 하라.

- 덩치 큰 시스템은 분석, 수정, 테스트할 때 코드가 많으면 쉽지 않고 유지보수 생산성도 떨어 지므로 가능한 코드 베이스를 작게 한다

07. 테스트를 자동화 하라.

- 당연. 개발자가 일일이 손으로 한땀 한땀 테스트 하고 있을 수 없습니다. 자동화 할 수 있는 것들은 모두 자동화로 돌리는 것이 좋다

08. 클린 코드를 작성 하라.

- 코드 베이스에 TODO, 죽은 코드 같은 무의미한 것은 삭제 해라

버트 c. 마틴" 클린 코드를 작성하는 7대 규칙

- 단위 수준의 코드 악취를 남기지 말라.
- 나쁜 주석을 남기지 말라.
- 주석 안에 코드를 남기지 말라.
- 죽은 코드를 남기지 말라.
- 긴 식별자 이름을 남기지 말라.
- 매직 상수를 남기지 말라.
- 제대로 처리 안 한 예외를 남기지 말라.

09. 아키텍처 컴포넌트를 느슨하게 결합하라.

- 느슨하게 결합된 최상위 시스템 컴포넌트가 수정하기 더 쉽고 시스템을 모듈화 할 여지가 많다.

10. 아키텍처 컴포넌트의 균형을 잡아라

- 컴포넌트가 넘치거나 모자라지 않게 크기를 거의 비슷하게 균형을 맞춰 모듈화 한 아키텍처는 관심사를 분리할 수 있고 고치기가 쉽다.

THANKS

ABACUS

www.iabacus.co.kr

Tel. 82-2-2109-5400

Fax. 82-2-6442-5409