

CLEAN CODE

1. 깨끗한 코드란 무엇일까. 나쁜코드는 어떤 문제를 가지고 올까

깨끗한 코드라는 건,

가독성이 좋은코드. 누구나 공들였다는 것의 눈에 보이는 코드를 말한다.

나쁜코드는 읽기 어렵기 때문에 개발 빠로를 저해시키고 사이드 effect를 가져온다.

경우에 따라 성능에도 영향을 끼칠 수도 있다

깨끗한 코드

- 보기에 즐거운 코드, 우아하고 효율적인 코드. 세세한 사항까지 꼼꼼하게 처리하는 코드

- 가독성↑, 잘 쓴 문장처럼 읽힌다. 의존성 최소, 다른 사람이 고치기 쉬운 코드.

- 모든 테스트를 통과, 중복x. 시스템 내 모든 설계 아키텍처를 표현, 클래스·메소드·함수 등을 최대한 줄인 코드

- 코드를 읽으면서 짐작했던 기능을 각 블록이 그대로 수행하는 것

나쁜코드

- 개발속도 저하, 팀 생산성 저하

* 나쁜코드의 위험을 이해하지 못하는 관리자의 말을 그대로 따르는 행동은 전문가답지 못하다.

2. 보이스카우트 규칙이 흔한코드와 어떤 관계가 있을까

보이스카우트의 규칙은 “옳을 때보다 더 깨끗하게 유지”하고 가라는 것. 우리가 코드를 짤 때도 이와 같아야 한다는 뜻이다

- “캠프장은 처음 옳을 때보다 더 깨끗하게 해놓고 떠나라”

3. 의미 있는 이름은 어떻게 지어야 할까, 클래스·메소드의 이름은?

불필요한 정보가 없이 명확하게 해당 클래스, 메소드 등이 어떤 일을 하는지 명시할 수 있는 것.

이해하기 쉽고 누구나 검색할 수 있는 이름?

- 변수, 함수, 클래스의 이름은 ‘존재 이유, 수행 기능, 사용 방법’에 대해 모두 담을 수 있어야 한다.

- 코드에 그릇된 정보가 담겨서는 안된다.

- 서로 혼동의미를 사용하지 않도록 유의해야 한다

- a1, a2.. 와 같이 무의미한 연속된 숫자를 덧붙이는 방법은 좋지 않다

- 방음하기 쉬운 이름을 사용하라

- 이름의 길이는 범위(크기)에 따라야 한다

- 검색하기 쉬운 이름을 사용하라

- m-name 과 같은 접두어를 사용할 필요는 없다

- 굳이 인터페이스에 IShapeFactory 처럼 “I”를 붙일 필요는 없다.

```
int realDaysPerIdealDay = 4;  
const int WORK_DAYS_PER_WEEK = 5;  
for (int i=0; i<10; i++) {  
    ...  
}
```

클래스이름 - 명사, 명사구를 사용하라

메서드 이름 - 동사, 동사구가 적합하다

- 너무 가변한 이름은 피하라

- 한 개념에 한 단어만 사용하라

- 의미 있는 맥락을 추가하라. (ex. firstName → addFirstName)

- 일반적으로 짧은 이름이 긴 이름보다 좋지만, 의미가 분명해야 한다

4. 함수에서는 클린코드를 어떻게 짜야 할까

- 읽기 쉽게 험수 → 다음 험수는 추상화 수준이 한 단계 높아야 한다 = 내려가기 규칙

- 파라미터는 적을 수록 좋다. 파라미터가 출역 값이 되는 것은 좋지 않다 = 이상적인 인수의 개수는 0개이다.

- 험수 하나에서는 하나의 기능만을 수행해야 한다

- 가능한 작게 만들어야 한다

- 험수는 한 가지를 해야 하며 그 한 가지를 잘 해야 한다

- switch문은 추상파트리 안에 숨기도록 한다

- 사용적인 이름을 사용하라. 험수가 작고 단순할 수록 이름을 고르기 쉽다

- 이름을 볼 때 일관성이 있어야 한다

- 플래그 인수는 가능하면 사용하지 않는 편이 좋다

- 단항 험수는 험수와 인수가 동사/명사의 쌍을 이루어야 한다 ex. writeField names

- 험수는 무언가 수행하거나 담하거나. 둘 중 하나만 해야 한다. 둘다하면 혼란을 야기할 수 있다

5. 좋은주석, 나쁜주석?

좋은주석 - 의도가 있는 경우. 수식과 같이 필요에 의한 주석. 등...

나쁜주석 - 관리x 인 주석. 히스토리 관리 목적인 경우. 코드와 같은 의미를 손 주석 등

- 주석은 필요악이다. 언제나 실패를 의미한다.

- 주석은 나쁜코드를 보완하지 못한다. 코드로 의도를 표현하라

좋은 주석 ↗ 저자의 의도 ex. // 여유시간이 둉문하지 않다면 실행 X

- 법적인 주석, 으도를 설명하는 주석, 정보를 제공하는 주석, 의미를 명료하게 봐주는 주석, 결과를 참고하는 주석

TODO 주석, 중요성을 강조하는 주석

나쁜 주석 ↗ ex) @param the 제목

- 마지막에 손 주절거리는 주석, 같은 이야기를 주석으로 중복하는 것. 외해할 여지가 있는 주석, 의무적으로 다는 주석

이력을 기록하는 주석, 너무 당연한 사실을 적은 있으나 미나한 주석, 함수·변수로 표현 가능한 경우,

위치를 표시하는 주석 (ex. //), 담는 패킷에 따른 주석, 저자를 표시하는 주석, 주석 처리한 코드, HTML 주석

너무 많은 정보 = TMI, 주석과 코드 간의 관계가 모호한 경우, 히든 캐스팅의 Javadoc

6. 형식은 어떻게 맞출 수 있을까

줄바꿈, 가로·세로를 몇 줄로 할 것인지 등을 팀에서 맞추면 가독성↑

이름은 쉽게. 순서는 내려쓰기로, 등등?

형식을 맞추는 목적 - 시간이 지나 원래 코드의 흔적을 찾아보기 어려워지더라도 맨 처음에 잡아놓은 구현 스타일과 가독성 수준은 유지보수 용이성과 확장성에 계속 영향을 미치기 때문

- 적절한 행 길이를 유지하라. 일반적으로 큰 파일보다 작은 파일이 이해하기 쉽다
- 소스 파일 첫 부분은 고차원 개념과 알고리즘을 설명하고 아래로 내려갈수록 의도를 세세하게 묘사한다
- 각 개념은 한 행 (center)으로 분리하라
- 서로 밀접한 코드의 행은 서로 가까이에 위치해야 한다. 수직거리
- 가로로는 120자 정도가 적당하다
- 범위로 나눠진 계층을 표현하기 위해서는 코드를 들여써야 한다
- 소스일은 일관적이고 매끄러워야 한다

7. 자료 추상화의 의미는?

추상 인터페이스를 제공해 사용자가 구현을 모른채 자료의 핵심을 조작할 수 있어야 한다

아무 생각 없이 조회(제터) / 설정(세터) 함수를 추가하는 것은 나쁘다

Public Interface Vehicle {
 double getFuelTankCapacityInGallons();
 double getGallonsOfGasoline(); } Better → Public Interface Vehicle {
 double getPercentFuelRemaining(); }

8. 절차적인 코드 vs 객체적인 코드

- 절차적인 도형 클래스

```
public class Square {  
    public Point topLeft;  
    public double side;  
}
```

```
public class Circle {  
    public Point center;  
    public double radius;  
}
```

```
public class Geometry {  
    public final double pi = 3.1415 ...;  
    public double area(Object shape)  
        throws NoSuchException  
    if(shape instanceof Square) {  
        Square s = (Square)shape;  
        return s.side * s.side;  
    }  
    else if ...  
    }  
    throw new NoSuchException();  
}
```

절차적인 코드

- 기존 자료구조 변경없이 새 함수를 추가하기 용이
- 새로운 자료구조의 추가 어려움
- 새로운 함수가 필요한 경우에 적합

객체지향적인 코드

- 기존 함수의 변경 없이 새 클래스 추가가 용이
- 새로운 함수의 추가 어려움
- 새로운 자료구조가 필요한 경우에 적합

즉, 객체지향코드에서 어려운 변경은 절차적인 코드에서 수우며

절차지향코드에서 어려운 변경은 객체지향에서 한다

- 객체지향적인 도형 클래스

```
public class Square implements Shape {  
    private Point topLeft;  
    private double side;  
    public double area() {  
        return side * side;  
    }  
}
```

```
public class Circle implements Shape {  
    ...  
}
```

9. 디미터 법칙이란?

어떤 함수를 호출하는데 있어서 그 내부까지 알면 안된다는 의미.

$a = b.c().d()$.. 이런 것 X

- 모듈은 자신이 조작하는 객체의 뒷사정을 몰라야 한다는 법칙

- 기차 헤드 ex) final String outputDir = ctxt.getOption("outputDir").getAbsolutePath();

예시와 같이 여러 객체가 이어진 형태

- 객체의 경우 내부 구조를 숨겨야 하므로 위의 예보다 디미터 법칙에 위반되지만 자료구조의 경우 내부 구조를 노출하는게

당연하므로 위반되지 않는다

- 집중구조 : 정반은 객체, 정반은 자료구조인 상태. 되도록 피하는 편이 좋다

10. 자료 전달 객체란?

- DTO를 의미한다. 오로지 변수만 있는 것

- 공개변수만 있고 함수는 없는 클래스 . DTO (Data Transfer Object)라고도 한다

- 활성레코드 : DTO의 특수한 형태 save, find와 같은 툴식 함수를 제공한다

데이터베이스 테이블이나 다른 소스에서 자료를 직접 반환한다

11. 오류처리는 어떻게 하는게 좋을까?

- 오류코드X 예외O. 예외가 발생하는 경우 exception은 잘 걸어주어야 한다. 플랫폼이 잘 될 수 있도록.

- try-catch 문을 사용하자 → 예외가 발생하는 코드를 할 때는 try-catch-finally 문으로 시작하면 좋다

- 오류코드보다는 예외를 사용하자

- 예외를 던질 때에는 실패한 연산과 원인 등을 언급해야 원인 찾기가 수월진다.

- null은 반환하지 않도록 해야 한다. null을 반환하고 싶은 유혹이 있다면 예외를 던지거나 특수 사례 객체를 반환하도록 한다

- 메소드에 null을 전달하는 것도 지양해야 한다. 차라리 예외를 던지거나

12. 클린코드에서 경계의 의미는?

- 외부코드(라이브러리 등) 와의 경계를 의미. 그 사이를 어떻게 처리할 것인가에 관한 것
- 패키지·프레임워크 제공자는 적용성을 최대한 높이려고 하는 반면 사용자는 자신의 요구에 집중하는 인터페이스를 바란다
- 경계 인터페이스를 이용할 때는 이를 이용하는 클래스나 클래스 계열 밖으로 노출되지 않아야 한다
- 곧바로 우리 코드에 외부 코드를 사용하는 대신 먼저 테스트 코드를 짜면 좋다 = 학습 테스트
- 학습 테스트는 해당 API에 대한 이해도를 높여준다
- 경계에 위치하는 코드는 절감히 분리한다. 또는 기대치를 정의하는 테스트 케이스를 작성한다

13. 깨끗한 단위테스트를 짜려면 어떻게 해야 할까

- 실패하는 코드 → 성공하는 코드 순으로 짜야 한다
- 명세를 알아보기 쉬워야 한다
- 테스트 케이스지 등을 통해 케이스를 모두 테스트 했는지 확인하면 좋다
- 하나의 테스트에는 하나의 명세 만을 확인하는게 좋다
- TDD 법칙 세 가지
 1. 실패하는 단위테스트를 작성할 때까지 실제 코드를 작성하지 않는다
 2. 컴파일은 실패하지 않으면서 실행이 실패하는 정도로만 단위 테스트를 작성하다
 3. 현재 실패하는 테스트를 통과할 정도로만 실제 코드를 작성한다

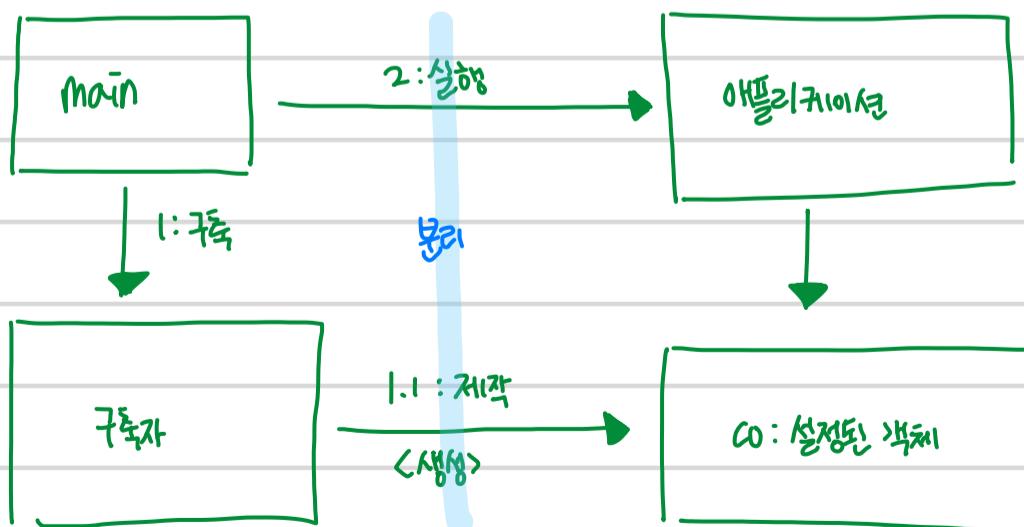
] 대략 30초 정도로 주기가 둡니다
- 실제 코드가 진화하면 테스트 코드도 변해야 한다. 테스트 코드가 지저분할 수록 변경하기 어려워진다. 즉 실제 코드 만큼 깨끗하게 짜야 한다
- 자동화된 단위테스트는 설계와 아키텍처를 최대한 깨끗하게 보존해준다
- 테스트는 유연성, 유지보수성, 재사용성을 제공한다
- 테스트 코드도 가독성이 중요하다. 간결하고 표현력이 좋아야 하지만 효율적이지 않아도 된다
- 단일 assert 문을 사용하는 편이 좋다 = 테스트 함수마다 하나의 개념만 테스트 하라
- F.I.R.S.T
Fast, Independant, Repetable, Self-Validating, Timely
= 결과는 항상 Boolean = 즉시에

14. 클래스에서는 어떻게 클린코드를 가져가야 할까

- 일반적으로 클래스의 순서는 전백면수 → 비밀면수 → 메소드 순이다
- 가능한 캡슐화를 유지하는 것이 좋다 = 캡슐화를 풀어주는 것은 최후의 수단이어야 한다
- 클래스는 작아야 한다
- 클래스의 이름은 but, if, or 등을 사용하지 않고 25자 내외로 가능해야 한다
- 클래스의 이름은 해당 클래스의 책임을 기술해야 한다
- 단일 책임 원칙을 지켜야 한다 * 단일 책임 원칙 (SRP) = 클래스나 모듈을 변경할 때가 단 하나뿐이어야 함
- 인스턴스 변수가 적어야 한다. 메소드가 변수를 더 많이 사용할수록 메소드와 클래스의 응집도는 높아진다
 - ↳ 응집도가 높다는 것은 클래스에 대한 메소드와 변수가 서로 의존하여 논리적인 단위로 묶인다는 의미
- 응집도가 높아질수록 변수와 메소드를 적절히 분리해 새로운 두세개의 클래스로 꾸며준다
- OCP (open-closed principle) 를 지원할 수 있어야 한다
 - ↳ 확장에 개방적이고 수정에 폐쇄적이어야 함
- 인터페이스와 추상 클래스를 사용하여 구현에 미치는 영향을 최소화해야 한다

15. 시스템의 의미와 이와 관련한 클린코드의 내용은?

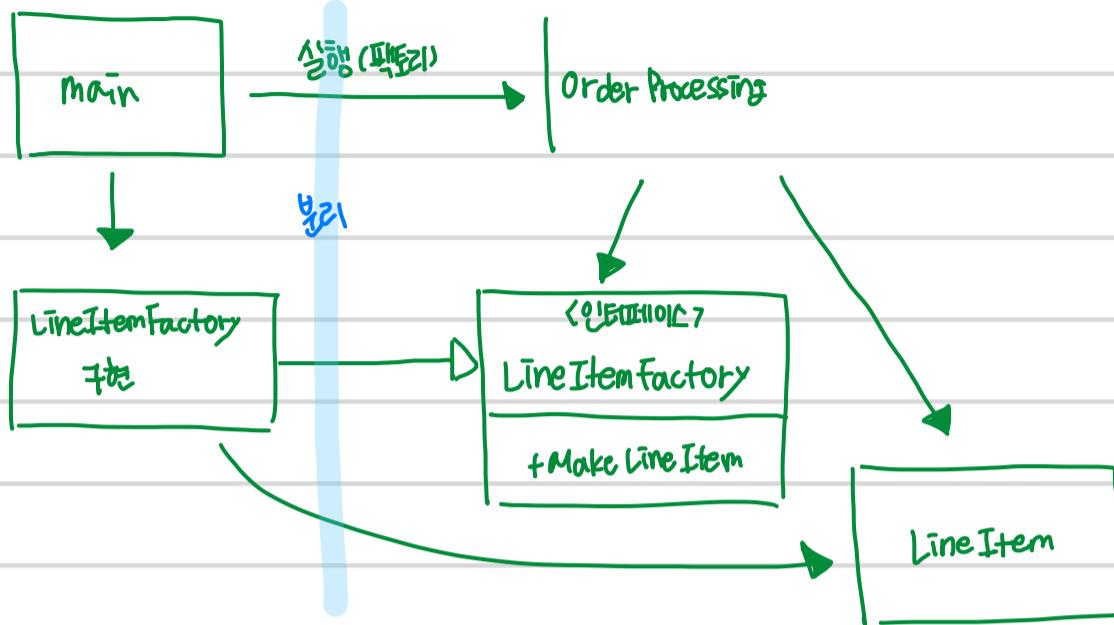
- 시스템 = 높은 추상화 수준을 의미
- 소프트웨어 시스템은 어플리케이션 객체를 제작하고 의존성을 서로 연결하는 준비과정과 준비과정 이후에 이어지는 런타임 로직을 분리해야 한다
- Main 분리
 - ↳ 시스템 생성과 시스템 사용을 분리하는 방법 중 하나
 - ↳ 생성과 관련한 코드는 모두 main이나 main이 호출하는 모듈로 옮기고 나머지 시스템은 모든 객체가 생성되었고 모든 의존성이 연결되었다고 가정



↳ 애플리케이션은 main이나 객체가 생성되는 과정은 전혀 모른다

- 팩토리

↳ 객체가 생성되는 시점을 애플리케이션이 결정할 필요가 있는 경우



↳ LineItem 을 생성하는 시점은 애플리케이션이 결정하지만 LineItem 을 생성하는 코드는 애플리케이션이 모른다

- 의존성주입 Dependency Injection, DI

↳ 사용과 제작을 분리하는 강력한 매커니즘

↳ 제어역전 기법 (IoC, Inversion of Control)을 의존성 관리에 적용한 것

↳ 새로운 객체는 넘겨받은 책임만 맡으로 단일 책임 원칙을 지킬 수 있다

= 외부에 값을 설정해놓고 가져다 쓰는 것

```
val bmw = CarBMW();
val tesla = CarTesla();
rideMyCar(bmw);
```

```
val initMyCar(car : Car) {
    this.car = car
}
rideMyCar(car);
```

- 확장

↳ 처음부터 모든 것을 구현할 수는 없다. 만족적이고 점진적으로 확장시켜야 한다

- POJO (Plain Old Java Object) ↔ EJB (Enterprise Java Bean)

= 별도로 종속되지 않는 모든 자바 객체

ie = 모든 기능이 다 들어야 있어서 무거움

ex. getter, setter 만 존재하는 Java Bean

이유는 개념이고

이유는 서비스 이름임

- 최선의 시스템 구조는 각각 POJO 객체로 구현되는 모듈화된 관심사 영역으로 구분된다.

서로 다른 영역은 해당 영역 코드에 최소한의 영향을 미치는 관점이나 유사한 도구를 사용해 통합한다.

- 시스템은 도메인 특화 언어가 필요하다

16. 창발성의 의미와 클린코드에서의 내용은 뭘까

- 창발성 : 남이 모르거나 하지 아니한 것을 처음으로 또는 새롭게 알려 내거나 이루어 내는 성질
- 단순한 설계 규칙 4가지가 소프트웨어의 품질을 높여준다 = 창발적 설계

1. 모든 테스트를 실행한다

↳ 테스트 가능한 시스템이 우선적이다, 즉 의존도가 ↓되어야 한다

2. 중복을 없앤다

↳ 똑같은 코드는 당연히 중복. 비슷한 코드는 더 비슷하게 고쳐주면 리팩토링이 쉬워진다

3. 프로그래머 의도를 표현한다 = 개발자의 의도를 정확히 표현할 수 있어야 한다

3.1 좋은 이름

3.2 함수와 클래스 구기를 가능한 줄인다

3.3 표준 명칭을 사용

3.4 단위 테스트를 꼼꼼하게 작성

4. 클래스와 메서드 수를 최소로 줄인다

↳ 극단적으로 X 가능한 정도로만 줄이는게 좋다

놓찍기 꽉찬 관계인지
잘모르겠음 -ㅅ-

리팩토링의 영역

17. 동시성을 클린-하게 짜는 방법은?

- 다중스레드와 관련한 내용. 데이터, 세마포어, 앤든 어찌구 스파게티 학자 etc...
- 동시성은 '무엇'과 '언제'를 분리하는 전략이다. 무엇과 언제를 나누면 애플리케이션 구조와 효율이 좋아진다
- 동시성의 이해와 미신

↳ 동시성은 항상 성능을 높여준다 → 때문에 성능을 높여준다

↳ 동시성을 구현해도 설계는 변하지 않는다 → 일관적으로 무엇과 언제를 분리하면 시스템 구조가 크게 달라진다

↳ 웹 또는 EJB 컨테이너를 사용하면 동시성을 이해할 필요가 없다

- 동시성의 타당한 생각

↳ 동시성은 다소 부하를 유발한다

↳ 동시성은 복잡하다

↳ 일관적으로 동시성 버그는 재현하기 어렵다

↳ 동시성을 구현하려면 흔히 기본적인 설계 전략을 재고해야 한다

- 동시성 코드는 다른 코드와 분리하라
- 자료를 캡슐화하라. 공유 자료를 최대한 줄여라 = 자료 사본을 사용하라
- 독자적인 스레드로, 가능하면 다른 프로세서에서, 둘려도 괜찮도록 자료를 독립적인 단위로 분할하라
- 스레드에 안전한 컬렉션

Reentrant Lock : 한 메서드에서 잠고 다른 메서드에서 푸는 Lock

Semaphore : 전형적인 세마포어. 개수가 있는 Lock

Count Down Latch : 지정한 수만큼 이벤트가 발생한 후에 대기 중인 모든 스레드를 해제하는 Lock

모든 스레드에게 동시에 공평하게 시작할 기회를 준다

- 동시성의 관련 용어들

↳ 한정된 자원 (Bound Resource) : 다중 스레드 환경에서 사용하는 자원으로 크거나 숫자가 제한적이다

ex. DB연결, 글이가 일정한 일기 / 쓰기 버퍼

↳ 상호배제 (Mutual Exclusion) : 한번에 한 스레드만 공유 자료나 공유 자원을 사용할 수 있는 경우

↳ 기아 (Starvation) : 한 스레드나 여러 스레드가 굉장히 오랫동안 혹은 영원히 자원을 기다린다.

↳ 데드락 (Deadlock) : 여러 스레드가 서로가 끌어당기며 기다린다. 모든 스레드가 각기 필요한 자원을 다른 스레드가 점유하는 바람에

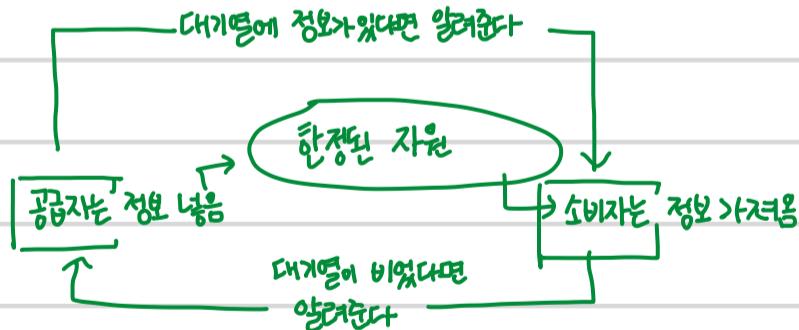
어느쪽도 더 이상 진행하지 못한다

↳ 라이브락 (Livelock) : 락을 가는 단계에서 각 스레드가 서로를 방해한다. 스레드는 계속해서 진행하려 하지만

공명으로 인해 굉장히 오랫동안 또는 영원히 진행하지 못한다

- 다중 프로그래밍 실행 모델

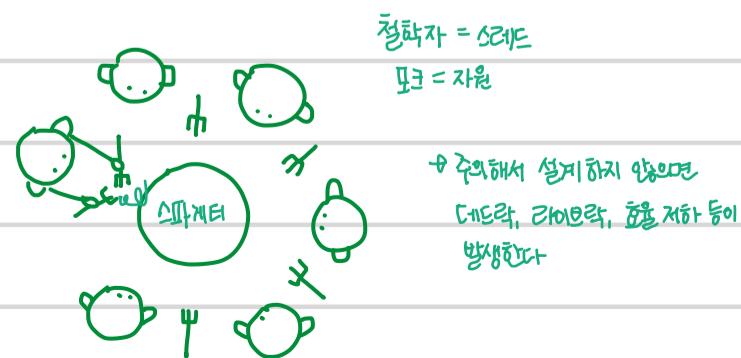
↳ 생산자 - 소비자 (Producer - Consumer)



↳ 읽기 - 쓰기



↳ 식사하는 철학자들



- 공유 객체 하나에는 하나의 멤버드만 사용하라. 여러 멤버드가 필요하다면 아래 세 가지 방법을 고려하라

1. 클라이언트에서 잠금 2. 서버에서 잠금 3. 연결 서버

- 동기화하는 부분은 최대한 작게 만든다

- 문제를 노출하는 텍스트 코드를 작성하라

- 시스템 실패를 일회성으로 치부하지 마라

- 스레드 환경 밖에서 생기는 버그와 스레드 환경에서 생기는 버그를 동시에 디버깅하지 마라