## 문제 **1.**

대략 1달간 고생 많으셨습니다.

## 문제 **2.**

Domain에 대한 표현력을 직관적으로 전달하여 어떤 시스템인지 빠르고 쉽게 파악 할 수 있습니다.
아무리 복잡한 시스템이더라도 Domain이 잘 표현되어 있다면 대충 패키지를 보고 내부의 코드 구성을 보면서 무엇을 하려고 하는지 단숨에 파악 할 수 있습니다.

이를 통해 빠르게 업무에 투입시킬 수 있습니다.
Domain에 대한 표현이 엉망인 경우 새로운 시스템에 갓 들어온 시니어조차 해당 시스템을 파악하는데 꽤 많은 시간을 소요하게 됩니다.

## 문제 **3.**

GoogleTest를 사용하여 TDD를 수행하는 경우
전체적인 비즈니스 아젠다를 잡아놓고
그 아젠다에 대한 대략적인 추상화 설계를 선행 할 수 있습니다.

이를 통해 Domain에 대한 파악을 훨씬 더 잘 할 수 있습니다.
또한 테스트를 통해 무엇을 테스트 하는지 확인하면
현재까지 코드에서 진행하고 있는 요소들, 예외 상황들에 대해 훨씬 더 잘 파악 할 수 있습니다.

즉 지속적이고 점진적으로 확장하는 시스템을 개발하기 위해 필수적인 덕목입니다.

## 문제 **4.**

3번 문제에 적었듯이 TDD와 DDD는 상호간 밀접한 연관성을 가지고 있습니다.
Domain을 잘 찾고 더 표현력 좋은 SW를 만들기 위해 Test를 선행하는 것이 좋습니다.
결론적으로 TDD와 DDD는 사실상 한 묶음으로 가야함을 의미합니다.

## 문제 **5.**

- 문서화
  무엇을 하려고 하는지 팀원들 및 같이 협업하는 모든 이들에게 쉽고 간결게 전달 할 수 있어 합니다. 이것에는 각종 Backlog, EPIC, 관리가 필요한 여러가지 문서들 등등이 존재합니다.
- 알림
  누가 무엇을 하고 있는지, 무엇을 했는지, 어떤 이슈가 발생 했는지 등등을 빠르게 파악 할 수 있어야합니다. 이를 통해 무엇을 도와줘야 하는지, 혹은 이제 무엇을 만들것인지를 판정 할 수 있습니다.
- 이슈 관리
  인간은 역사를 반복한다고 합니다. 똑같은 문제가 수십, 수백번 반복됩니다. 아무런 기록이 없고 그 기록 자체가 공유되지 않는다면 당연 반복됩니다. 기록이 되어 있어도 반복이 되기 때문입니다 (상대적으로 줄어듬) 그러므로 여러 상황들에 대한

이슈들을 Domain 단위로 관리하여 각 상황들에 대해 발 빠르게 대응 할 필요가 있습니다.

- 소통
  술자리에서 '하하호호' 하는 소통을 의미하지 않습니다. 물론 감정이 상하면 인간 관계 자체가 틀어지기 때문에 감정이 상하는 것은 당연히 조심해야 합니다. 여기서 논하는 소통이란 결국 '무엇을 만들 것이고 이것을 어떤 목적으로 만드는 것인가' 에 대해 답 할 수 있어야 함을 의미합니다.
- DDD
  결국 위의 저런 것들을 다 하더라도 코드 레벨에서 Domain 단위의 관리가 이루어지지 않는다면 아무런 의미가 없을 가능성이 높습니다. 왜냐하면 결국 코드를 보면 이게 뭐 하려고 하는 것인지 판단하기가 어려울 수 있기 때문입니다. (이 시점에서는 프로그래밍 고고학이 진행되기 시작합니다)
- Test
  DDD 만으로 요약하여 설명하기 어려울 수 있습니다. 그러나 Test가 잘 작성되어 있다면 어떤 Domain에서 어떤 작업들을 진행하는지 더 빠르게 판단 할 수 있습니다.
- 의사 결정 속도
  무엇인가를 진행하기 위해서는 의사 결정이 필요합니다. 빠른 의사 결정을 진행하기 위해서는 각자에게 어느정도의 결정권이 있어야 합니다. 우리는 이것을 달성하기 위해 결국 소통의 속도를 가속 할 수 있도록 이슈, 알림, DDD, TDD 등을 진행한다 봐도 무방합니다.

## 문제 **6.**

IoC는 결국 Inversion of Control의 약자로 의존성을 역전시킵니다.
의존성을 역전시키면 변경이 발생 할 때 그 변경의 파급을 최소화시킬 수 있습니다.
그리고 사실 SW의 변경이란 사실 필연적인 것이기 때문에 변경 유연성을 가져가는 것이 중요합니다.
IoC는 이런 변경의 유연성을 제공해주고 함수 포인터는 IoC를 달성하기 위해 필요한 방식입니다.

## 문제 **7.**

C++에서 함수 포인터를 사용하지 않고 IoC를 달성 할 수 있는 방식입니다.
Java에서 사용하는 interface와 유사합니다.

## 문제 **8.**

1. 우선 즉시 이슈를 발행합니다.
2. 이슈를 발행한 이후 문제를 해결해봅니다.
3. 만약 이슈에 대한 답변이 완료 되었음을 알려주는 알림이 도착하면 그를 토대로 작업을 마무리합니다.
4. 이후 자신이 문제를 해결 했다면 해당 이슈에 대한 기록을 남깁니다.
5. 꽤 긴 시간 동안 문제를 해결하지 못하고 있는 상황이라면 다른 작업으로 쉬프트합니다.

## 문제 **9.**

Backlog를 작성함으로서 현재 자신이 무엇을 하고 있는지 협력하는 모든 이들에게 알려 줄 수 있습니다. 이를 통해 회의를 진행하는 상황에서 훨씬 더 효율적인 회의를 진행 할 수 있고 현재 시점에서 주요한 아젠다가 무엇인지 빠르게 파악하고 전달 할 수 있습니다.

### 문제 **10.**

다른 사람들이 무엇을 하는지에 관심을 두지 않았기 때문입니다. 상호간 연결되는 부분이 있다면 해당 작업을 담당하는 담당자와 소통을 통해 데이터를 주고 받는 인터페이스를 맞추는 작업이 필요합니다. 이것을 모르면 서로 엉뚱한 작업을 하고 있거나 고고학을 펼쳐야합니다.

### 문제 **11.**

서로 어떤 목적으로 기능을 만드는 것인지 파악이 되지 않기 때문에 같은 목적을 가지고 있는 서로 다른 여러가지 기능들이 중복적으로 나타날 수 있고 이로 인해 코드 전체가 오염되는 문제가 발생하게 됩니다.

### [ 복합 문제 **12 ~ 20**번 ]

DB 테이블은 아래와 같이 만듭니다.

```
CREATE TABLE velocity (
        x_coordinate INT,
        y_coordinate INT,
        registration_datetime DATETIME(3)
);
```

**github** 링크

### 문제 **21.** 고찰에 대한 답변

사람의 숫자가 빈번하게 바뀌는 상황이면 1분마다 갱신합니다. 만약 사람이 별로 붐비지 않는 시간이라면 사람의 숫자에 변동이 있는 경우에만 값을 갱신하도록 구성합니다.

### 문제 **22.** 이론 답변

함수 마다 Stack이 별도 존재하기 때문에 경계선을 넘어 다른 함수의 Stack에 접근하기 위해 필요합니다.

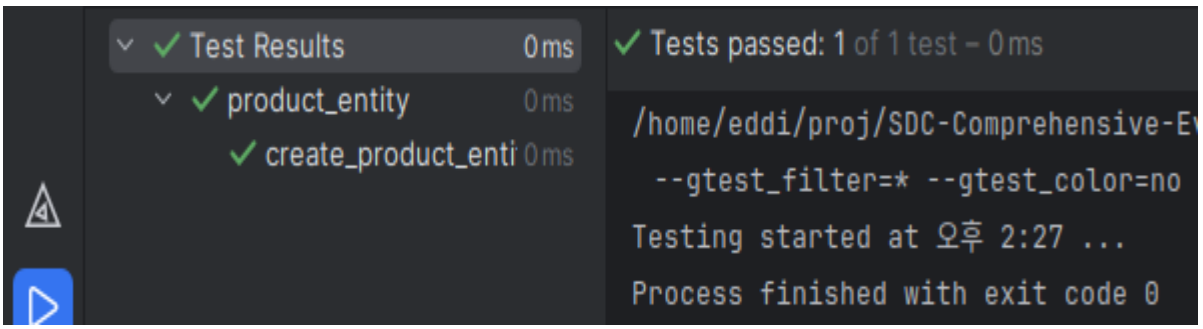
### 문제 **23.** 이론 답변

그래야 함수마다 별도의 Stack 공간 (지역 변수들)을 활용 할 수 있기 때문입니다.

### [ 복합 문제 **24 ~ 25** ]

**github** 링크

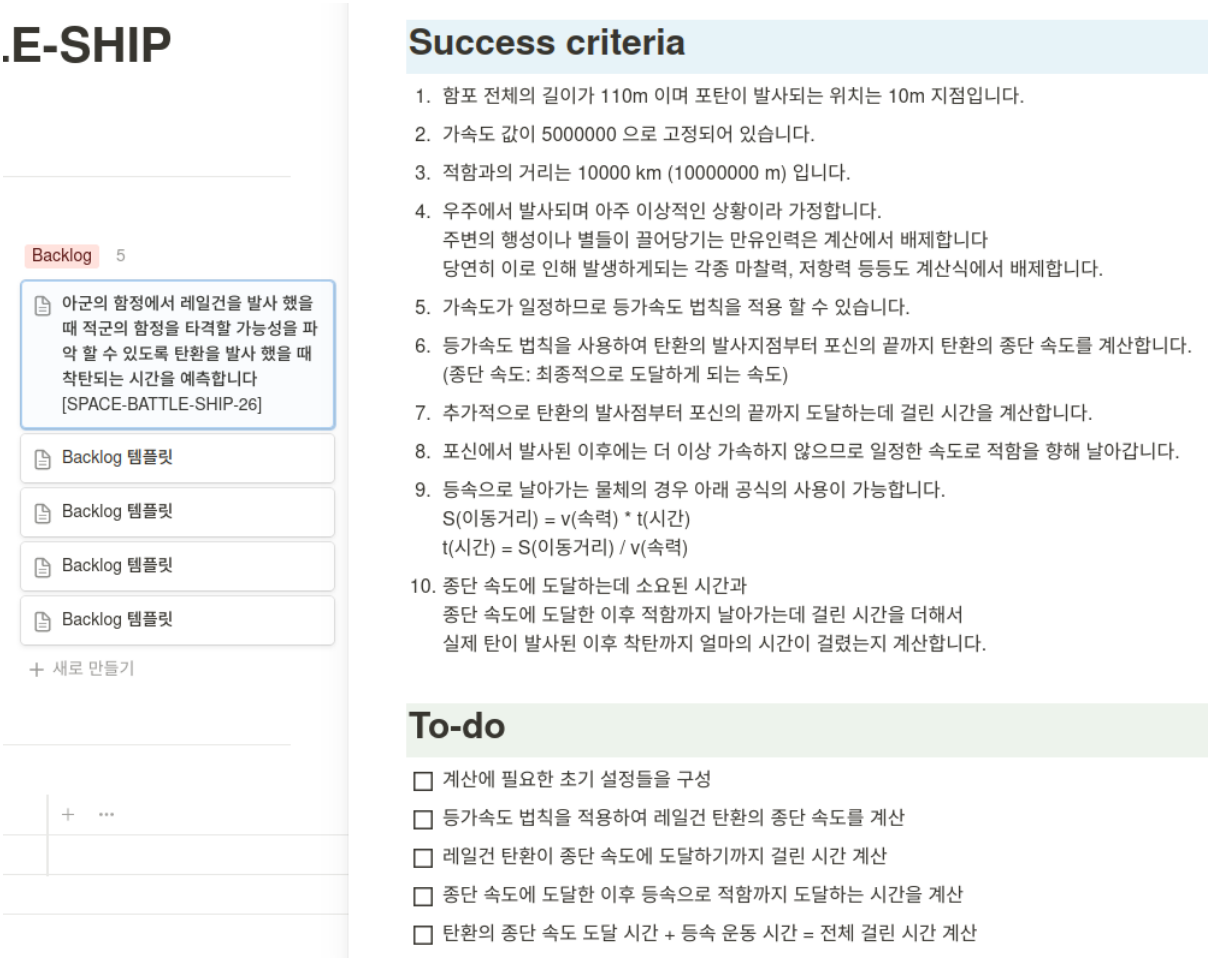
### **GoogleTest** 결과




[ 복합 문제 **26 ~ 32** ]

**github** 링크

작성한 **Backlog**




문제 **33.** 이론 답변

점차적으로 복잡도 높아지고 있기 때문에 해당 작업에 대한 Domain을 별도로 분리합니다. 이후 분리한 Domain으로 인해 변경되는 사항들에 대한 Refactoring(리팩토링) 작업이 필요합니다. 작업량이 많을지라도 반드시 Refactoring 하여 추후 발생 할 문제들을 최소화시키는 작업이 필요합니다.

[ 복합 문제 **34 ~ 36** ]

**github** 링크

**Valgrind**로 **Memory Leak** 검사한 결과 스크린샷 첨부 **(**하단 참고**)**