프로그래밍 역량 강화 전문기관, 민코딩

SSD 프로젝트
기능 추가 요청

목차

1. 기능 추가 미션 소개
2. SSD – Erase 기능 추가
3. Logger 기능 추가
4. Runner 기능 추가
5. SSD - Command Buffer 기능 추가
6. 시나리오 추가시, 재빌드 이슈 해결
7. 기능 추가 미션, 유의사항

기능 추가 미션 소개

기능 추가, 리팩토링 미션 개요

CRA과정 프로젝트 진행 스케쥴

<table>
<tr><th colspan="2">시간</th><th>1일차</th><th>2일차</th><th>3일차</th><th>4일차</th><th>5일차</th></tr>
<tr><td>8:30</td><td>9:30</td><td rowspan="2">Team Building &
Communication</td><td rowspan="4">Team Project</td><td rowspan="4">프로젝트 안내
Team Project</td><td rowspan="4">Team Project</td><td rowspan="2">Team Project
*발표준비</td></tr>
<tr><td>9:30</td><td>10:30</td></tr>
<tr><td>10:30</td><td>11:30</td><td rowspan="2">Team Project
개발환경 셋업 및 안내</td><td rowspan="2">Team Project
발표, 평가 준비</td></tr>
<tr><td>11:30</td><td>12:30</td></tr>
<tr><td>12:30</td><td>13:30</td><td colspan="5">점심 시간</td></tr>
<tr><td>13:30</td><td>14:30</td><td rowspan="2">개발환경 안내
Team Project</td><td rowspan="4">Team Project</td><td rowspan="4">Team Project</td><td rowspan="4">Team Project</td><td rowspan="2">발표</td></tr>
<tr><td>14:30</td><td>15:30</td></tr>
<tr><td>15:30</td><td>16:30</td><td rowspan="2">Team Project</td><td rowspan="2">현업활동 Guide</td></tr>
<tr><td>16:30</td><td>17:30</td></tr>
</table>

3 ~ 4일차 미션

미션

1. **SSD - Erase 명령어 추가**
2. **Logger 기능 추가**
3. **Runner 기능 추가**
4. **SSD - Command Buffer 기능 추가**
5. **시나리오 추가시, 재빌드 이슈 해결**

미션 1, 2번은 필수 사항입니다.
3, 4, 5번은 도전 미션입니다.

디자인 패턴을 사용시, 추가 점수

디자인 패턴은 여러 곳에서 발생될 수 있는 문제를 해결하는 일반화된 모범 사례입니다. 이를 클린코드의 Best Practice 라고 합니다. 이로 인해, GoF 디자인패턴을 사용하면 약간의 추가 점수가 주어집니다.

예시)

- Sigleton Pattern
- Command Pattern 등

SSD – Erase 기능 추가

Erase 기능이 지원되도록, 가상 SSD와 Shell 에 명령어를 추가한다.

SSD - Erase 기능 추가

Write 명령어 처럼 SSD 자체에 Erase 기능을 추가하고,
Shell에서 명령어를 수행할 수 있도록 한다.

- SSD 명령어 : E [LBA] [SIZE]
  - 특정 LBA 부터 특정 Size 까지 내용을 삭제한다.
  - 삭제할 수 있는 최대 Size는 최대 10칸이다.
  - SSD에서 삭제하면 0x00000000이 된다.
- Shell 명령어 1 : erase [LBA] [SIZE]
  - SSD에 명령어를 수행한다.
- Shell 명령어 2 : erase_range [Start LBA] [End LBA]
  - Start LBA 부터 END LBA 직전 까지 내용을 삭제한다.

Logger 기능 추가

Test Shell 이 출력하는 Log를 관리해주는 시스템 제작

Logger 가 필요한 이유

테스트를 진행하면서 Fail이 발생하면 Log Message를 불량 분석을 하게 된다.

검증팀의 여러 사람들이 각자 테스트 시나리오를 작성하다 보면,
통일되지 않은 로그 포맷을 사용하지 않아, 불량 분석에 시간이 걸리기도 하고
로그들을 Text파일로 저장하는데 로그 데이터가 대량으로 쌓이면서 파일 관리가 어렵다.

위와 같은 문제를
해결해주는 기능을 추가하자.

우리 팀만의 print 함수를 만들면 된다.
□ 화면 출력이 되면서, 동시에 파일 Write가 되는 print 함수

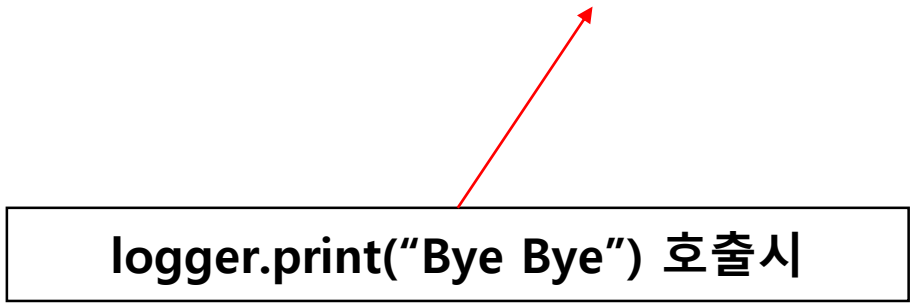
Logger 기능 1 : 통일된 로그

로그 포맷

- 출력 포맷 : [날짜 시간] 함수명( )　　: 로그메세지
- 함수명이 들어가는 공간에는 **30칸**을 확보하여 함수명을 기입한다.

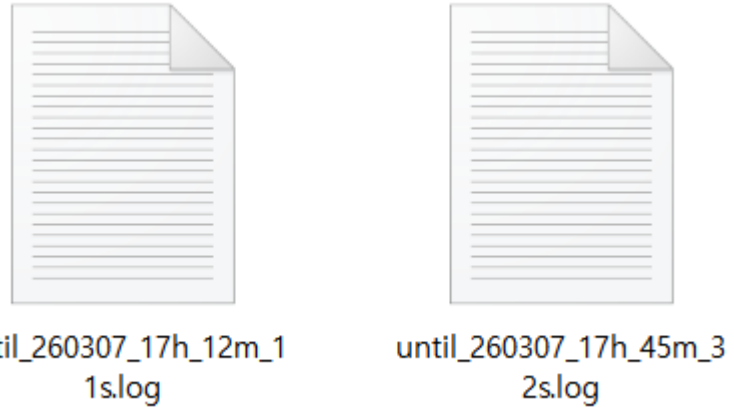
로그 포맷 예시

[26.01.01 17:04] sendMessageWithData( )　　: send and wait sync message.
[26.01.01 17:04] checkMessage( )　　: FAIL - Response message didn't arrive.
[26.01.01 17:04] release( )　　: Bye bye.



Logger 기능 2 : 로그파일 관리

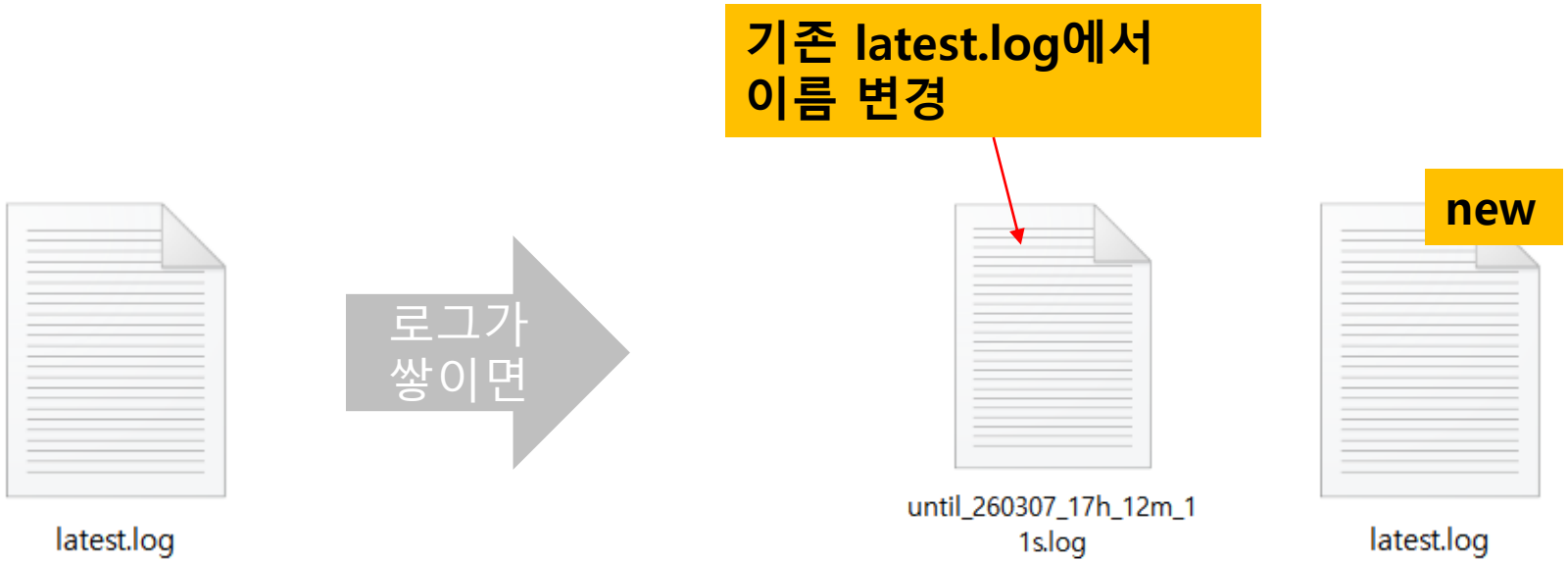
실제로 대부분의 불량 분석은 Last log 10MB 로 해결된다.
하지만 하나의 파일이 TB 이상이 된 상태로 불량분석에 시간이 오래걸리게 된다.
우리의 프로젝트에서는 조금 더 사이즈를 축소하여 10MB가 아닌
10KB 단위로 파일을 관리해보자.



Logger 기능 2 : 로그 기록 규칙

로그 기록 규칙

1. 로그를 남기기 위해 latest.log 파일을 생성한다.
2. 10kb 까지 기록을 수행한다.
3. 10kb가 넘으면 해당 파일의 이름을 “until_날짜_시간.log” 으로 변경한다.



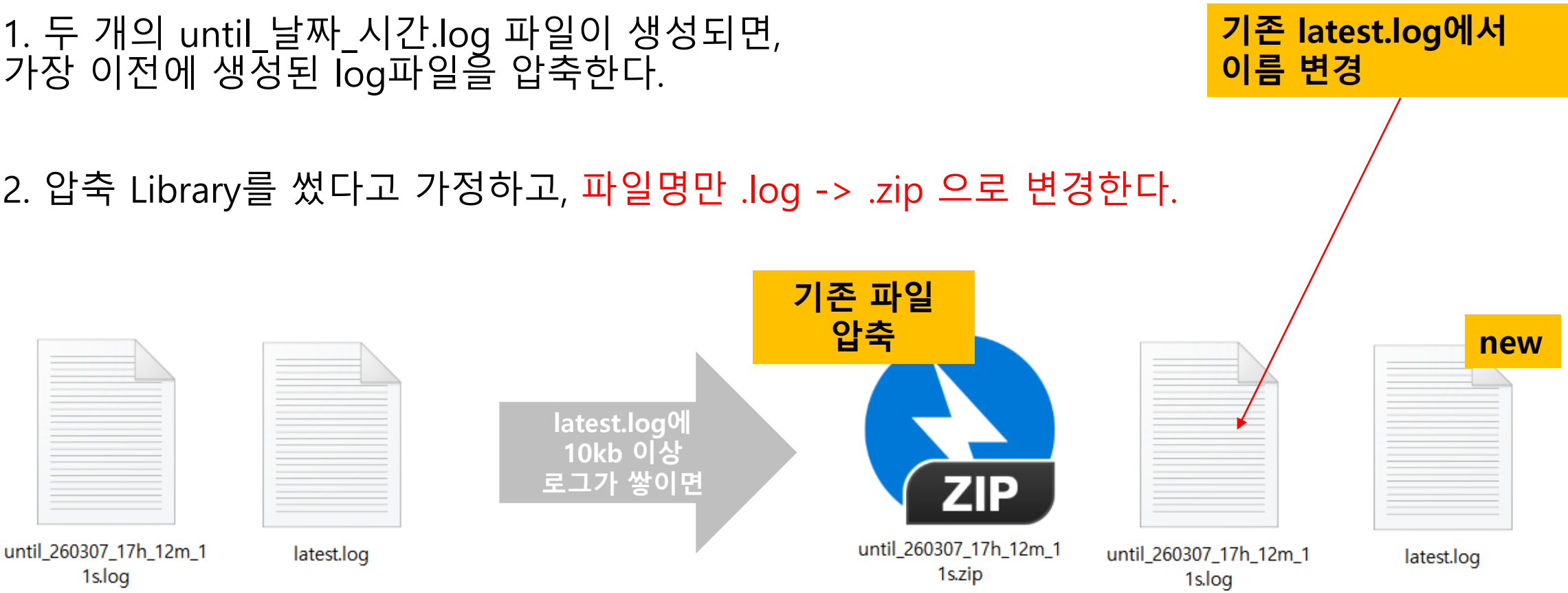
파일명 : 마지막 로그 기록의 날짜 + 시간

Logger 기능 2 : 로그 압축 규칙

로그 압축 규칙

1. 두 개의 until_날짜_시간.log 파일이 생성되면,
가장 이전에 생성된 log파일을 압축한다.

2. 압축 Library를 썼다고 가정하고, 파일명만 .log -> .zip 으로 변경한다.

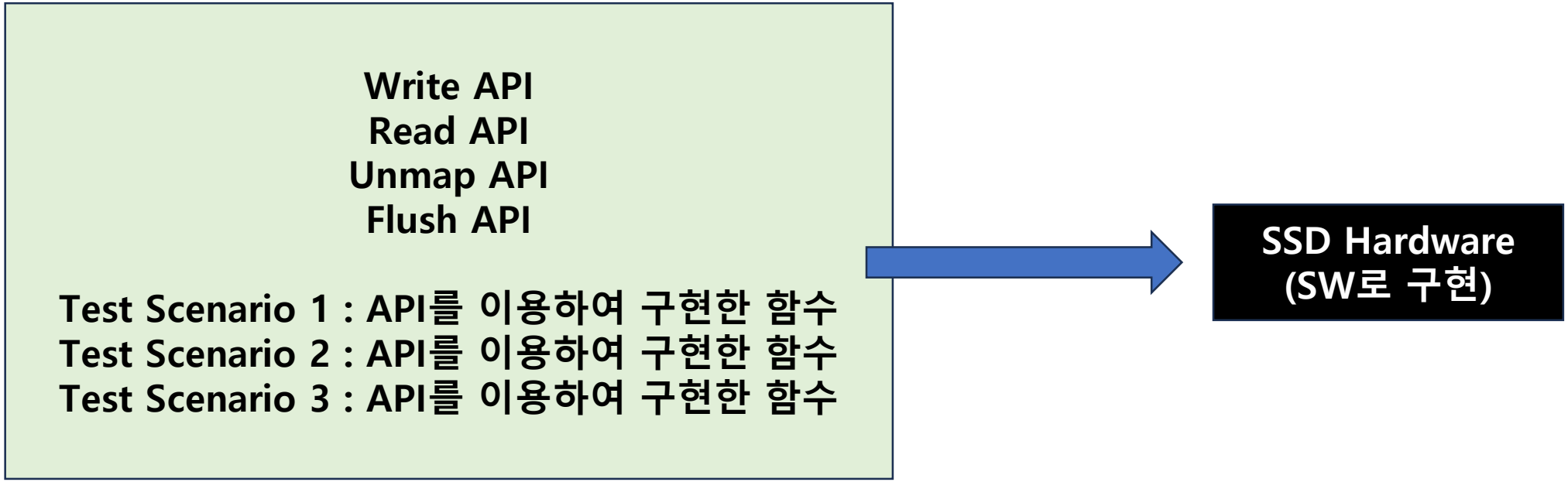


Runner 기능 추가

Test Script Name을 한 Text file에 모아, 순차적으로 수행하는 기능

현재 구조

각 API들을 사용하여, Test Scenario들이 만들어져있다.



Runner

테스트 시나리오들은
하나의 함수 형태로 만들어져 있으며,
Shell에서 하나의 시나리오 씩 수행할 수 있다.

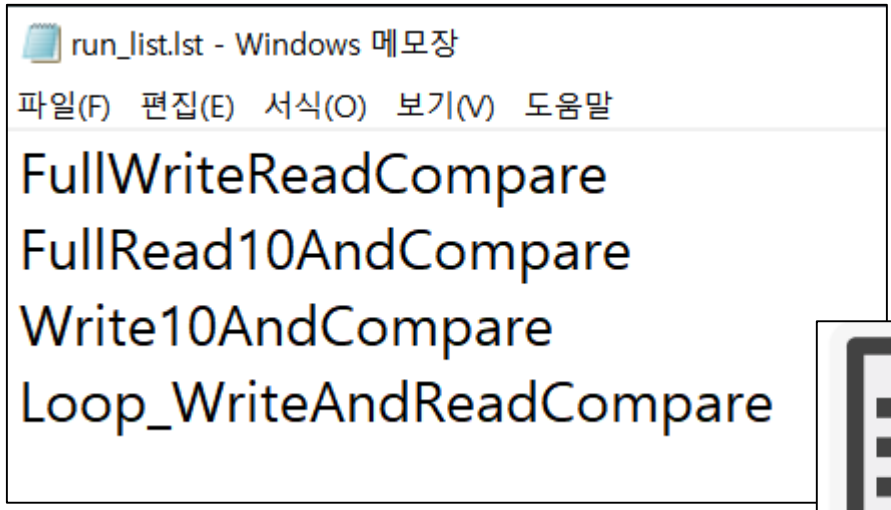
검증 팀에서는
미리 제작한 시나리오 이름을 txt파일로 저장한 후,
txt파일 이름만 입력하면, 기입된 시나리오들을
하나씩 순차적으로 실행하는 기능을 추가하고자 한다.

예시

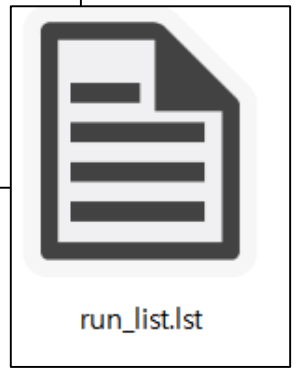
run_list 파일에 실행시키고 싶은 시나리오 이름들을 기입한다.
그리고 cmd 창에서

shell.exe run_list.lst

이라고 수행하면
테스트가 하나씩 수행된다.



Test Scenario 4개 기입 예시



결과 화면

1. 화면에는 동작 Text만을 출력한다.
동작Text는 우측 예시를 참고
스크립트 이름 --- Run ... 이 먼저 출력된다.

2. 테스트가 끝나고 Pass가 되면 Pass 출력 후,
다음 스크립트가 수행된다.

3. 테스트가 Fail로 끝나면 FAIL! 출력 후
즉시 프로그램이 종료되며
더이상 스크립트를 수행하지 않는다.

1

```
FullWriteReadCompare    ---  Run...Pass
FullRead10AndCompare    ---   Run...Pass
Write10AndCompare --- Run...
```

2

```
FullWriteReadCompare    ---  Run...Pass
FullRead10AndCompare    ---   Run...Pass
Write10AndCompare --- Run...Pass
Loop_WriteAndReadCompare --- Run...
```

3

```
FullWriteReadCompare    ---  Run...Pass
FullRead10AndCompare    ---   Run...Pass
Write10AndCompare --- Run...Pass
Loop_WriteAndReadCompare --- Run...FAIL!
```

SSD - Command Buffer 기능 추가

Command Buffer 및 Flush 명령어 기능 추가

Command Buffer 를 활용한 최적화

Command Buffer의 도입 목적

- 불필요한 Command 개수를 줄여, 성능을 향상

- Command들을 즉시 처리하지 않고, 모아둠으로써
  수행을 할 필요가 없는 명령어를 찾아내어 제거한다.

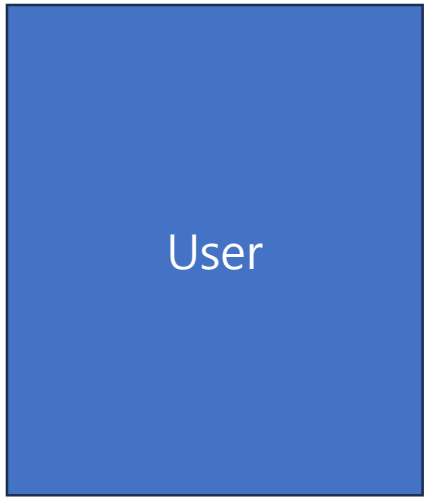
Command Buffer 개념 1

User가 가상 SSD에 3회의 Write를 수행한다고 가정한다.

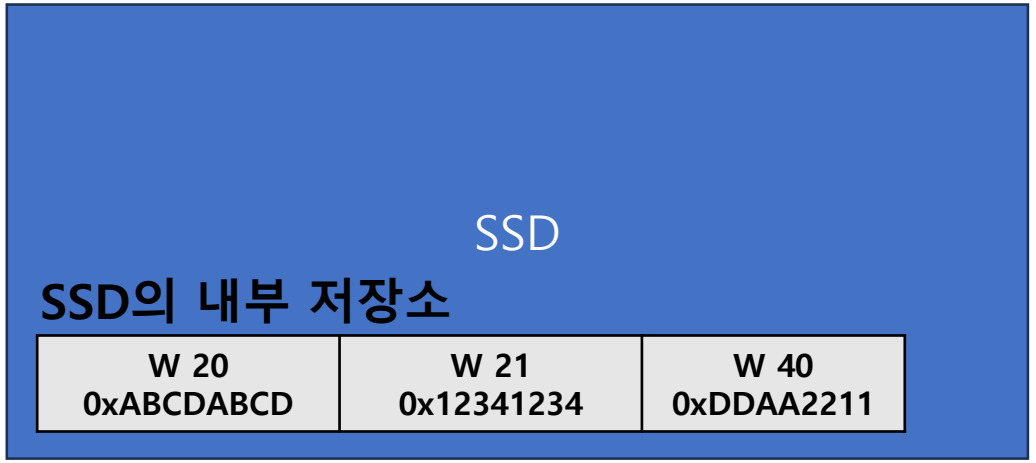
SSD!
내 3개의 Write 명령어를 받아 수행하렴.

Command Buffer 개념 2

SSD는 내부 저장소를 가지고 있어, Command들을 보관해둔다.
처리 속도를 더 높이는 알고리즘을 적용하기 위해,
한꺼번에 모아서 처리하려는 목적이다.



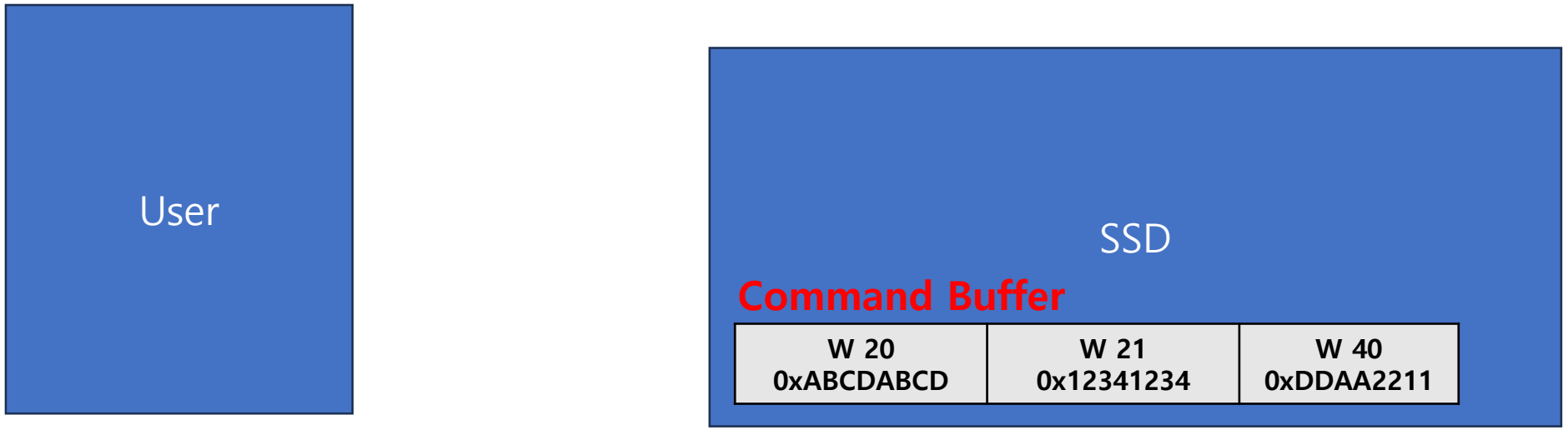
SSD는
내 명령을 다 수행했겠지?



User의 명령들은
일단 보관했다가 한꺼번에 처리해야지.

Command Buffer 개념 3

SSD의 내부 저장소를 **Command Buffer** 라고 한다.

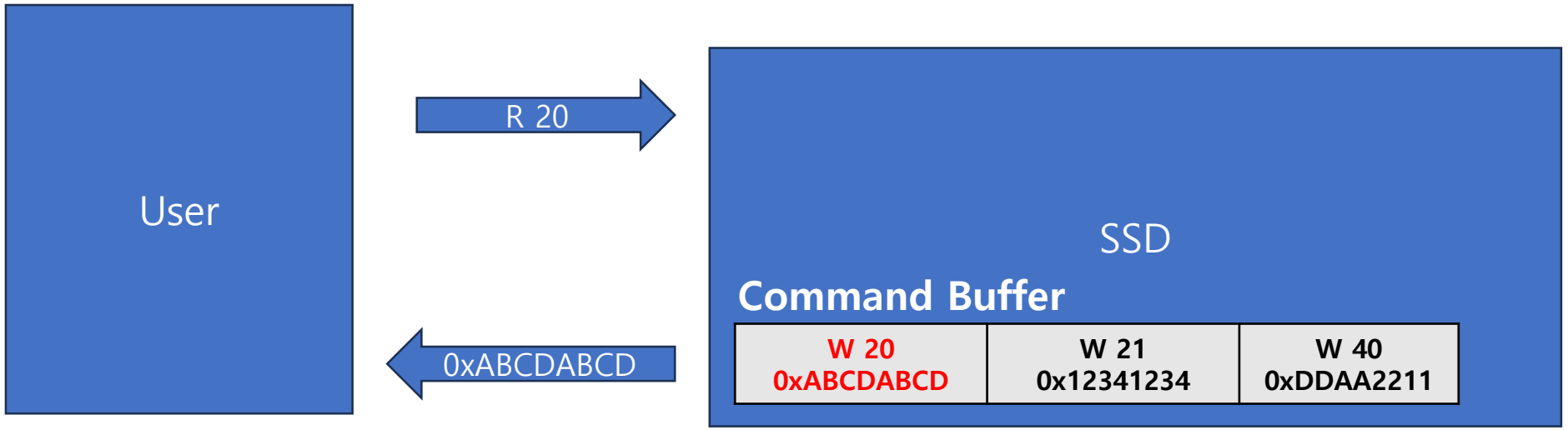


Command Buffer 개념 4

이 상황에서 User가 Read 20 LBA를 요청하면

nand를 확인하지 않고, buffer에 있는 값을 찾아 리턴해야한다.

- nand 보다 buffer에 있는 내용이 더 최신 정보이기 때문



SSD – Command Buffer 세부사항

Command Buffer 세부사항

- Write와 Erase명령어들을 Buffer에 저장한 후, 한꺼번에 명령어들을 수행한다.
- Buffer에는 최대 10개의 명령어를 저장할 수 있다.
- Buffer가 꽉찬 상태에서 새로운 명령어가 들어오면,
  즉시 Flush 수행 후, 새로운 명령어를 Buffering한다.
- nand.txt처럼, buffer.txt 파일을 사용하여 구현한다.

SSD – Flush 기능 (명령어 추가 필요)

- Command Buffer에 있는 모든 명령어들을 수행하여 Buffer 전체 비운다.
- SSD 명령어 이름 : F
- Shell 명령어 이름 : flush

Command Buffer를 이용한 성능 최적화 예시 1

Ignore Write 알고리즘 1

- 같은 LBA에 Write하는 명령어가 여러개 들어왔다면,
  Last 명령어만 남겨두고, 기존 명령어는 Buffer에서 삭제한다.

Command Buffer를 이용한 성능 최적화 예시 2

Ignore Write 알고리즘 2

- 같은 LBA에 대해 Write 명령어와 Erase 명령어가 순차적으로 들어왔다면, Write 명령을 수행할 필요가 없기에 Buffer에서 제거한다.

Command Buffer를 이용한 성능 최적화 예시 3

Merge Erase 알고리즘

- Erase 명령어가 연속적으로 2회 들어왔고,
  2개의 Erase 명령어가 연속적인 LBA Range를 갖는다면,
  두 명령어를 하나의 명령어로 교체한다.

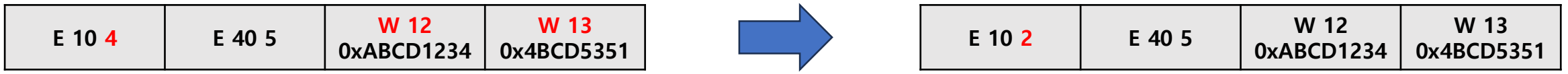
- Erase 명령어의 Size는 10보다 클 수 없음에 유의한다.

Command Buffer를 이용한 성능 최적화 예시 4

Narrow range of Erase 알고리즘

- Erase Range 의 시작 LBA 또는 마지막 LBA에 Write를 수행하는 경우 Erase 범위를 축소시킬 수 있다.

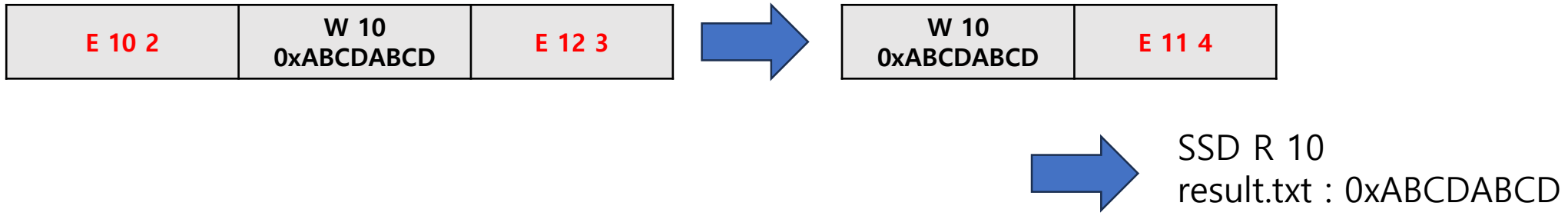
- Erase 할 범위가 남아있지 않다면, 명령어를 제거한다. (Ignore Erase)



Command Buffer를 이용한 성능 최적화 예시 5

Fast Read

- Read를 수행할 때, Command Buffer 부터 확인을 한다.
- Buffer에 Read할 값이 적혀 있다면,
  실제 NAND를 읽지 않고도, Buffer로 부터 값을 찾아 리턴할 수 있다.



[정리] Command Buffer 기능 추가

Command Buffer

- SSD의 명령어를 모은 Buffer.
- 꽉 차면 Flush를 수행한다.

Command Buffer 성능 최적화 알고리즘

- 불필요한 명령어를 삭제하여 SSD 성능을 올린다.

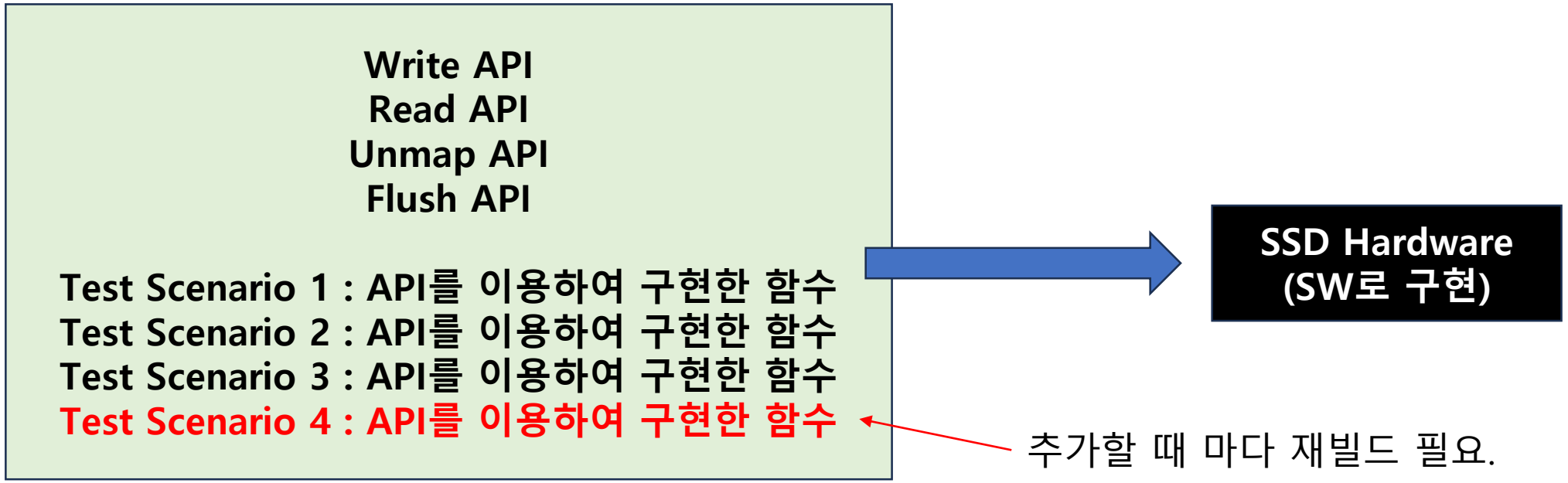
소개드리는 성능 최적화 예시 외,
다양한 아이디어를 적용하셔도 좋습니다.

시나리오 추가시, 재빌드 이슈 해결

난이도 HARD, Test Shell의 Script 관리 방법을 변경하자.

Test Scenario를 추가할 때마다 전체 재빌드 필요.

매번 시나리오를 추가할 때 마다,
쉘을 재빌드하는 문제가 있다.



재빌드하는데 걸리는 시간이 너무 크다.

Test Scenario 함수들은
매우 많으며,
소스파일은 300개가 넘는다.

이 때문에
한번 빌드하는데 걸리는 시간은
5분 걸린다고 가정하자.

- 사소한 버그 낼때마다 재빌드 (5분 소요)
- 한번 테스트할때마다 재빌드 (5분 소요)

이로인해 개발이 힘들다.

Write API
Read API
Unmap API
Flush API

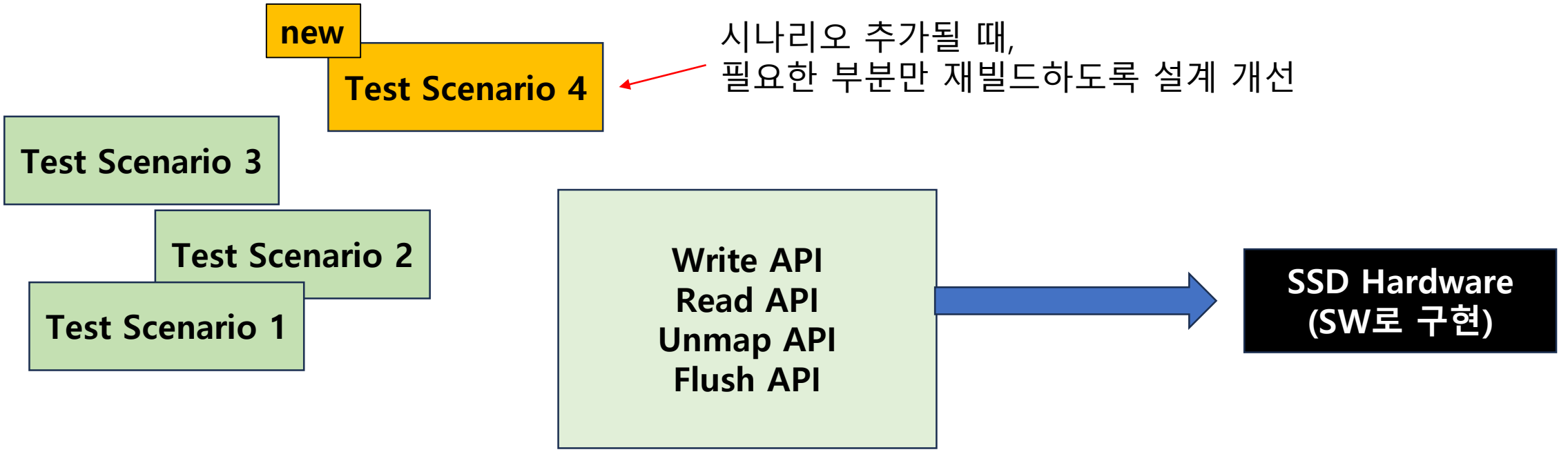
Test Scenario 1 : API를 이용하여 구현한 함수
Test Scenario 2 : API를 이용하여 구현한 함수
Test Scenario 3 : API를 이용하여 구현한 함수
Test Scenario 4 : API를 이용하여 구현한 함수

…

Test Scenario 999 : API를 이용하여 구현한 함수
Test Scenario 1000 : API를 이용하여 구현한 함수

Test Scenario를 추가할 때마다 전체 재빌드 필요.

재빌드 해야하는 규모를 최소화할 수 있는
설계로 개선하는 리팩토링을 수행한다.



기능 추가 미션, 유의사항

미션 수행시, 유의사항

유의사항 1

1. 협업이 잘 되도록 역할분담을 해주세요.

2. 팀장님을 제외한 코드리뷰어를 1명 이상 꼭 지정해주세요.
팀원들이 개발한 코드는 코드리뷰어가 리뷰해주시면 됩니다.
코드리뷰어가 개발한 코드는 팀장님이 리뷰해주세요.

3. 시간 간격을 협의하셔서, 정해진 시간별로 코드리뷰 담당자를 변경해주세요.

유의사항 2

4. 3일차 부터는 TDD로 개발하지 않아도 됩니다.

5. 일부러 지저분하게 작성 후 Commit을 해주어도 좋습니다.
개발과 리팩토링을 동시에 하는 것을 자제해주세요.

6. 개발할 때는 마음 편하게
변수명, 함수명 네이밍 등을 사용해도 좋습니다

7. 개발 후 동작이 잘 되면 Commit을 하고,
클린코드가 되게 끔 리팩토링을 하여 Commit을 해주세요.
전 후 비교되게끔 해주시면, 발표자료를 만들 때 사용하기 좋습니다.