Story Based Writing

개요

게임 서버 구현에서 스크립트 성격의 코딩은 피하기 어렵다. 많은 상태와 조건에 따른 분기, 클라이언트 메세지 전송으로 구성된 처리 코드가 필요할 수 밖에 없다. 단위 테스트의 발전 중 하나인 BDD (Behavior Driven Development)는 글쓰기처럼 행동을 정의한다.

예를 들면 다음과 같다.

```
Story: Returns go to stock

As a store owner
In order to keep track of stock
I want to add items back to stock when they're returned.

Scenario 1: Refunded items should be returned to stock
Given that a customer previously bought a black sweater from me
And I have three black sweaters in stock.
When he returns the black sweater for a refund
Then I should have four black sweaters in stock.

Scenario 2: Replaced items should be returned to stock
Given that a customer previously bought a blue garment from me
And I have two blue garments in stock
And three black garments in stock.
When he returns the blue garment for a replacement in black
Then I should have three blue garments in stock
And two black garments in stock.
```

위의 방식을 완전히 고정된 틀로 적용하기는 어렵지만 게임 서버 코드의 컨텐츠 스크립트 작성에 활용 가능해 보인다. 적어도 주석과 코드 실행 흐름을 정리하는 용도로는 가능하다.

모색

실게임 예

```
void     MazeInDarknessProcessor::onCheckInUser(EntityUnit* pUnit, EntityUnit* target, UInt32 v1, UInt32
v2)
{
          UNREFERENCED_PARAMETER(target);
          VERIFY_RETURN(pUnit && pUnit->IsValid() && pUnit->IsPlayer(), );

          RETURN_IF(m_sector == nullptr);

          if(m_currentState == STATE_NONE)
          {
               changeState(STATE_WAIT);
          }
}
```

먼저 UInt32 v1, UInt32 v2의 의미를 알기 어렵다. UInt32 대신 alias 타잎이라도 이름이 있다면 훨씬 파악하기 쉽다. VERIFY_RETURN은 그 자체로 확립된 패턴이 될 수 있는데 pre-condition과 invarint, post-condition을 분리하여 EXPECT, VERIFY, ENSURE로 나누면 스토리 파악이 더 쉬워진다.

RETURN_IF(...)는 VERIFY_RETURN()이나 EXPECT 형태로 변경하여 처리해야 일관성이 있고 깔끔하다. UNREFERENCED_PARAMETER(target)이 이름이 좀 길다. UNUSED() 정도로 alias를 만들면 더 깔끔해질 듯 하다.

if (m_currentState == STATE_NONE) 일 경우 상태를 변경하는데 else 문에 대한 처리가 없다. else 일 경우 버그가 나온 것이다.

onCheckInUser()는 어떤 인터페이스의 요구 때문에 강제된 것으로 (왜냐하면 target을 사용하지 않기 때문에) 결과값을 받아 에러 처리를 하는 함수이어야 한다. 이름이 의미하는 바는 미로나 미궁에 사용자를 넣는 매우 **중요한** 과정이기 때문이다.

- 타잎이름의 추가는 의미 파악에 중요
    - 값이라고 하더라도 필요한 경우가 있음
    - 불가능하면 변수 명에 의미를 부여
- pre-condition, invariant, post-condition을 위한 일관된 매크로 사용
- if 문이 있으면 else 문에 대한 고려가 반드시 필요
    - 인터페이스 차원에서 에러 처리를 고려해야 함
- 의도의 명시
    - 헤더 파일 함수 주석이 가장 적합한 위치

```
using result = result<bool, std::string>; // bool 의미를 갖는 클래스 또는 구조체.
result   MazeInDarknessProcessor::onCheckInUser(EntityUnit* pUnit, EntityUnit* target, UInt32 v1, UInt32
v2)
{
          UNUSED(target);
          VERIFY_RETURN(
           pUnit && pUnit->IsValid() && pUnit->IsPlayer(),
           result(false, "checkin user ptr is invalid");
         );

          VERIFY_RETURN(
           m_sector != nullptr,
           result(false, "sector ptr is null");
```

```
        );

        VERIFY_RETURN(
          m_currentState == STATE_NONE,
          result(false, "maze state is invalid");
        );

        changeState(STATE_WAIT);

       return result(true, "checked in and wating");
}
```

onCheckInUser를 호출하는 쪽에서 result를 보고 처리가 가능하면 처리를 한다. 에러를 리턴하는 것은 역할 분담을 조정할 필요가 있을 경우에도 필요하지만 코드를 읽을 때 의미 파악을 위해서도 필요하다. 좋은 result, exception 클래스를 갖고 있으면 디버깅도 편하고 읽기도 쉽다.

습관 / 규칙 (Convention)

완전한 BDD 형태로 시나리오를 C++로 구현하면 자유도가 너무 떨어진다. 따라서, 모색 과정에서 느낀 점들을 정리하면 다음과 같다.

- 헤더 파일의 함수 주석에 의도(In order to)를 명시한다.
- 입력 변수의 의미 파악이 가능한 타잎과 변수 명을 사용한다
    - 사용 안 하는 변수들은 명시해 둔다.
    - UNUSED와 같은 간결한 형태가 더 낫다.
- bool 변환이 되는 result 구조체로 문자열로 의미 부여가 가능하게 한다.
    - 성능이 중요한 경우 enum을 갖는 result 구조체를 사용한다.
    - 이를 통해 실패한 경우의 의미, 성공한 경우의 의미를 부여할 수 있다.
- pre-condition, invariant, post-condition을 체크하는 매크로를 둔다.
    - VERIFY_RETURN이나 VERIFY_DO 와 같은 하나의 매크로로도 가능하다.
    - expect, ensure, check와 같은 형식도 가능하다.
- 포인터 유효성 체크가 필요한 경우를 최대한 배제하는 것이 좋다.
    - 타잎 체크가 코드 의미를 많이 가린다.
- 실행 중 의미 파악을 위해 trace나 debug 로그를 추가한다.

정리

시작할 때 약간 거창한 포부가 있었는데 적다 보니 많이 작아졌다. 그래도 약간은 의미가 있어 보인다. 데이터로 흐름을 구성하고 이를 실행하는 방식의 BDD 스타일의 스토리 기반 개발은 계속 살펴볼 가치가 있으므로 상태나 행동, callback의 구성을 통해 진행될 수 있는 지 기회가 될 때 더 살펴본다.