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.

문체, 구조, 기법

BDD는 선언적 프로그래밍 영역에 해당한다. 선언적 스타일이 코드를 읽기 쉽게 만드는데 도움이 되는 영역이 상당히 있다. Fluent 스타일, 데이터 주도, 이벤트 주도 스타일과 함께 선언적 스타일을 적절하게 문체와 구조로 사용한다.

선언적 스타일은 STL, MPL에 모두 있다. 미리 시켜두면 나중에 알아서 뭔가 돌아가는 건 프로그래밍의 기본적인 마법이다. 이러니 이렇고 이러면 이렇다로 코딩하게 되면 복잡한 코드가 만들어진다. 대신에 이거 이거 이거 할 수 있고 이런 경우 이거 하고 저런 경우 저거하고 이게 아니면 이것들 못 하고 이렇게 설정해 두고 입력을 주면 동작하는 기계로 만들거나 코드 흐름을 작성하면 훨씬 깔끔해진다. 여기에는 문체, 구조, 기법이 모두 필요하다.

천재들이 주도한 프로그래밍은 알고리즘 위주로 발전하여 범인들을 위한 고려가 적다. 특히, c++과 같은 경우 전문가의 날카로운 도구로 쉬운 구현을 위한 노력이 다른 언어들에 비해 훨씬 적다. 이런 시대는 지나가야 한다. 여전히 가장 빠른 코드를 만들어 주며 컴파일 시간 타잎 계산이 완전한 언어로 얼마간의 노력만 더해지면 java나 javascript 만큼 대중적인 언어가 될 수 있다. 취향에 따라 갈리기는 하겠으나 가능해 보인다.

선언적

STL의 알고리즘 스타일이 대표적이다. 함수형 언어의 기법들을 더 많이 차용할 필요가 있다. when(cond).then(func1).else(func2); 와 같이 fluent 스타일과 합쳐서 코드 구조를 작성할 수 있다. 고차 함수를 사용하거나 함수 합성 등을 사용하면 호출을 선언적으로 구성할 수 있다.

이벤트

Mission, quest 등에 잘 맞는 구조이다. 외부와 일반적인 인터페이스를 만드는데 좋다. 이벤트가 아니라면 인터페이스 상속을 통해 맞춰야 하는데 이벤트로 하게 되면 OnEvent와 같은 함수 하나로 전체 이벤트 인터페이스를 만들 수 있다.

일반적이며 가장 커플링이 약한 구조이다. 유용한 부분이 꽤 있다.

데이터

가상 머신의 명령어처럼 실행을 주도하는 데이터를 구성한다. 스킬을 처리한다고 할 때 스킬 타잎과 타잎에 대한 데이터, 타이밍 정보 등을 보고 각 클래스나 함수를 호출하여 처리할 수 있도록 한다.

외부에서 설정을 하고 내부는 개별 함수의 정확성만 확보하면 되는 구조이다.

게임 서버의 문체

스토리 기반 흐름

- on
- scenario
- given
- when
- do
- then

on

메세지와 시간으로 시작된다. 문맥을 정리할 때 필요하다. 특정 함수에서 시작할 경우 on은 특별한 의미를 갖지는 않지만 시작한 문맥을 이해해야 하는 경우도 있다.

on의 하위 언어는 서버에서 message와 시간이다.

scenario

시나리오는 함수의 조건에 따른 분기를 나타낸다. given 에서 주어진 조건에 따라

given

given은 조건에 해당한다. 조건은 상태이다. 상태 중 필터링 해서 예외로 처리되는 것들이 있다. 이들을 filter라고 하자. filter는 에러 체크를 포함한다.

given은 if 문의 형태를 띄지만 나뉜다.

- 조건에 해당하는 경우
- filter에 해당하는 경우
- do에 해당하는 경우

filter는 별도의 매크로로 DSL을 만드는 것이 좋다. return_if() 나 verify()와 같은 매크로를 추가하여 함수의 처리 조건을 만족하지 않는 경우를 나누어 이후 논리 전개를 단순하게 만든다.

when

when은 함수 호출의 내용에 대한 설명이다. when은 if 문으로 나눠서 처리되는 경우가 대부분이다.

do

행위 자체이다. 함수 구현이다.

then

then은 ensure와 같이 post condition에 해당한다. do를 하고 난 후에 만족해야 하는 조건이다. do의 결과로서 상태를 만족해야 하는 조건이다.

개념 수준

모든 서술은 특정 개념의 수준에 따른다. 흐름, 알고리즘, 함수 내 처리에서 의미가 부여된다. 개념 수준을 정하고 그에 따른 타잎을 갖추며 함수를 통해 의미를 만들고 시나리오를 적용해야 한다.

예시. GetRewardFromVolume(volumeIndex, rewardType, rewardValue)

Story: RPG에서 맵에 설치된 특정 볼륨에서 보상을 해당 볼륨 안에 있는 플레이어들에게 제공한다.

On:

특정 사용자가 퀘스트 완료나 미션 완료를 했을 때

Scenario :

볼륨 안에 플레이어가 있을 경우 타잎별 보상을 제공한다.

시나리오에 맞는 코드를 작성한다.

```
// 볼륨 안에 있는 플레이어에 대해
GetPlayersInVolume( ..., players);

// 타잎별 보상을 제공한다.
switch ( rewardType )
{
    case RewardType::QuestCompletion:
        {
                std::for_each( players,
```

```
                            [rewardValue] ( auto player )
                            { RewardQuestCompletion(player, rewardValue); }
                           );
                break;
        }
    case RewardType::... :
        {
                ...
        }
        break;
    default:
        // error log
        break;
   }
}
```

하위 시나리오인 RewardQuestCompletion()은 다음과 같이 구현된다.

```
RewardQuestCompletion( EntityPlayer* player, questId )
{
    // Given : 진행 중인 questId의 퀘스트가 있다면
    player->GetActionQuest().CompleteQuest( questId );
    // Then : questId의 퀘스트가 완료됨
}
```

코드가 짧으니 람다에 모두 구현해도 된다.

```
std::for_each(
    players,
    [rewardValue] ( auto player ) { player->GetActionQuest().CompleteQuest( rewardValue ); }
);
```

이와 같이 분리하고 Scenario (목표), Given (조건), When (요청), Then (결과) 로 나누어 생각하면 흐름이 명확해진다.

위의 시나리오 구현 구조는 개별 플레이어에게 먼저 전달된 후 Component로 전달되게 하면 EntityPlayer가 매우 뚱뚱해진다 (엄청나게 많은 함수가 추가됨). 따라서, 적절한 균형점이 필요한데 컴포넌트를 얻는 방식은 괜찮아 보인다.

개념 수준의 분류

- on의 절차적 시나리오
  - 메세지와 조건을 검증하고 분기 하는 함수
- 대상의 개수
  - 여러 개 또는 단일

다른 차원의 분류가 가능하며 서버 분리된 (orthogonal) 개념들의 차원을 정리하는 노력을 지속한다.

비동기 처리

별도로 논의되어야 할 항목이나 문체와 관련 있기에 여기에 둔다.

타잎

개념을 갖고 있는 타잎들을 통해 구성되도록 설계하고 코딩하는 것이 중요하다. 타잎은 상태와 동작을 갖고 있는데 동작 자체가 하나의 개념이나 동작 만을 많아 처리하도록 하고 상위 동작은 이런 하위 동작들로 구성해야 검증 가능하고 이해하기 쉬운 코드가 나온다.

BDD 스타일의 의도와 결과에 기반한 흐름이 타잎과 타잎들의 동작으로 구성되도록 해야 한다. 개념화는 모든 곳에서 가능하고 필요하다.

대수적 구현

명확한 구현을 위해 대수적인 구조를 만드는 노력을 해야 한다. 대수적 시스템이란 더하기, 빼기와 같이 하나의 연산처럼 동작하는 구조를 만든다는 뜻이다. 대상 개념의 수준을 올리면 자연스럽게 복잡한 동작도 가능하게 되도록 해야 한다.

이 부분은 중요한 연구 과제로 보인다.