요구사항명세(SRS)

Project : Eternal Guardians Data Script Generator

작성자 : 조 수 운

이 문서의 내용에 대한 모든 권리는 ㈜중원게임즈에게 있습니다.
이 문서의 일부 혹은 전체를 ㈜ 중원게임즈의 허가 없이 복사,
전재하거나 공중에 배포하는 행위는 저작권법에 저촉됩니다.

검수자 서명

1. 모든 이해 당사자들은 아래 명세의 내용을 충분히 이해하고, 면밀히 검토하였다.
2. 구현 결과물 및 테스트 결과물은 이 문서에 적힌 명세의 내용을 올바르게 반영하여야 한다.
3. 명세 내용을 변경할 경우, 변경한 내용이 프로젝트의 수행 비용에 미칠 영향력에 대해 평가할 것을 강력히 권고한다.
4. 명세에 대한 변경사항이 있는 경우, 그 변경 내용을 반드시 문서에 반영해야 한다.

관계	부서	직위	이름	서명

일러두기

◆ 내용에 있어 각 문장의 글씨 색깔이 의미하는 바는 대략 이렇다.

- **검은색**

: 일반적인 내용의 서술에 사용한다. 강조하는 의미로 **굵은 글씨**를 사용한 부분도 있다.

- **빨강색**

: 내용을 읽을 때, 특별한 주의점을 나타내거나 강한 강조를 위해, 주목을 끌 목적으로 사용한다. 주로 부정적인 강조일 때 사용한다.

- **파란색**

: 내용을 읽을 때, 특별한 주의점을 나타내거나 강한 강조를 위해, 주목을 끌 목적으로 사용한다. 주로 긍정적인 강조일 때 사용한다.

- **연한 파란색**

: 약한 강조를 위해 사용한다. 긍정 / 부정적인 의미는 크게 구분하지 않고, 알아두면 좋을 만한 사항에 주로 사용한다.

- **진한 빨강**

: 현재 명세에는 해당하는 사항이 없거나, 혹은 아직 명확하게 정의되지 않은 내용이 있음을 나타낸다.

- **보라색**

: 외부 문서에 대한 참조는 이 색깔로 표기한다.

- **분홍색**

: 이 문서의 어떤 명세 항목에서 문서 내 다른 명세 항목을 참조해야 하는 경우, 참조할 명세 번호를 이 색깔로 표기한다.

- **황갈색**

: 작성자, 작성 시점, 추가 / 삭제 / 재정의 관련 처리 내용을 표기한다.

- **파란 음영 + 짙은 파란색**

: 명세 항목의 세부적인 내용을 나타낸다.
명세 항목의 개요 부분은 음영을 넣지 않고, 세부 내용에만 파란 음영을 넣는다.

- **녹색 음영 + 녹색**
: **기술 노트.**
(프로그래밍의)기술적인 힌트나, 왜 이렇게 구현을 했는가에 대한 의도 등은 이런 형식으로 표시한다. 이 색깔로 표시된 문장은, 읽기 싫은 사람이나 기술적인 용어가 짜증나는 사람들은 안 읽어도 상관없다.

- **주황색 음영 + 갈색**
: **테스트 노트.**
명세 및 구현 내용에 대해 테스트해야 할 경우, 테스트와 관련된 상세한 내용을 나열한다.

- **보라색 음영 + 보라색**
: **마케팅 노트.**
운영 / 과금 / 라이선스 등의 정책과 관련하여 알아두면 좋을 상세 내용을 나열한다.

- 회색 음영
~~삭제된 기능은 이렇게 삭제 표시가 된 채, 회색 음영으로 내용을 감싼다. (음영은 경우에 따라 넣지 않을 수 있다.)~~
~~만일 내용 중에 이미 음영 처리가 된 부분이 있다면, 그 부분만 회색 음영으로 감싸서 제외되었음을 표시한다.~~
~~그러므로, 명세 기능을 삭제하는 경우가 아니라면 이 색깔의 음영을 사용하지 말 것.~~

※ 근데 막상 작성하다 보면, 이런 색상 규칙 같은 거 잘 안 지키게 된다.-_-
그냥 이런 게 있다 생각하고, 글자 색상과 의미가 반드시 일치하는지 따지지는 말 것...

◆ **항상 일러두기를 최신으로 유지하려고는 하는데, 간혹 내용이 틀릴 수도 있다. 그러니 일러두기의 형식을 지나치게 의식하는 것은 좋지 않다.**

그냥! 편하게 읽으면 된다.(...) 편하게...

목 차

A. 개 요 6

A-1. Concept 7
A-2. 용어 설명 10
A-3. 제품 사용 시나리오 22

B. 기능 요구사항 27

B-1. 데이터시트 28
B-2. 데이터 스크립트 43
B-3. 소스 파일 45

C. 기반 시스템 50

C-1. 하드웨어 & 플랫폼 51
C-2. 소프트웨어 구조 52
C-3. 사용자 인터페이스 54

A. 개　　요

A-1. Concept

◆ A-1-1. 달성 목표

- **A-1-1-1.** Eteranal Guardians 프로젝트의 게임 데이터들을 정의하는 데이터시트를, 게임의 서비스 모듈에 따라 적합한 방식의 데이터 스크립트로 변환하는 기능을 자동으로 수행해주는 저작 도구를 만든다.

- **A-1-1-2.** 주요 목표 대상이 되는 게임의 서비스 모듈은 **클라이언트**와 **서버**다.
  : 게임의 클라이언트인지 서버인지에 따라, 데이터 스크립트의 형식(Format)을 다르게 한다.

- **A-1-1-3.** 클라이언트와 서버가 서로 다른 파일 형태의 데이터 스크립트를 사용하고 있더라도, 이를 적절히 변환해줄 수 있게 한다.

- **A-1-1-4.** 사람의 손으로 편집해야 하는 데이터시트들은 하나의 세트로 구성한다.
  : 게임 서비스 모듈 별로 별도의 데이터시트를 편집하지 않게 하고, 하나의 형식으로 통합한다.

- **A-1-1-5.** 사람이 편집하는 데이터시트는 하나일지라도, 이것이 각 서비스 모듈에 맞는 데이터 스크립트로 변환하고 배치할 때는, 해당 서비스 모듈이 필요로 하는 정보로만 구성하도록 자동화한다.

※ 예를 들어, 모델 데이터가 들어 있는 상대적인 디렉토리 경로 위치라든가, GUI 텍스트의 내용과 같은 정보들은 게임 클라이언트 프로그램에서만 필요하고, 게임 서버 프로그램에서는 전혀 알 필요도 없는 리소스들이다.
그렇기 때문에, 데이터 스크립트로 변환하는 경우에, 위와 같은 내용들은 클라이언트 프로그램에 쓸 데이터 스크립트에만 들어가 있고, 서버 프로그램에서 쓸 데이터 스크립트에는 포함하지 않는다.

- **A-1-1-6.** 사람이 편집하는 부분은 사람에게 친숙한 방식으로 데이터를 설계하고, 컴퓨터가 빠르게 처리할 수 있는 방식이 이와 상반된다면, **사람이 보기에 적합한 방식과 컴퓨터에게 적합한 방식을 자동으로 전환**할 수 있게 한다.

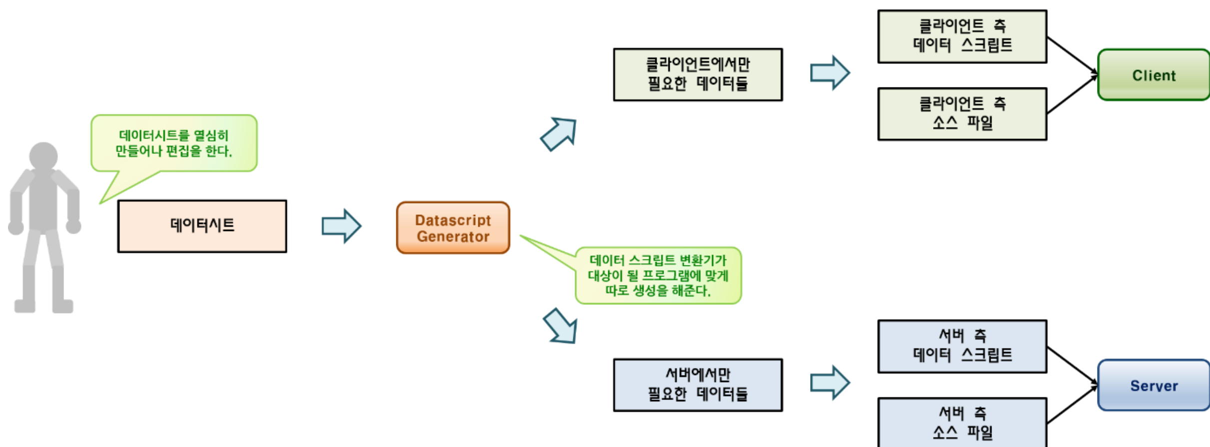
※ 사람이 순수하게 데이터를 볼 때는 행(Row)과 열(Column)로 되어 있는 표(Chart) 방식이 가장 눈에 잘 들어온다. 또한, 숫자만 나열되어 있는 것보다는, 의미를 가지고 있는 문자열로 파악하는 편을 더 좋아하고, 편안하게 느낀다.

하지만 문자열은 컴퓨터 프로그램에게는 친숙하지 않다. 컴퓨터 프로그램은 문자열보다 숫자를 훨씬 빠르게 다룬다. 뭔가를 검색할 때, 검색 키가 문자열인 경우보다 정수인 경우에 성능이 더욱 좋다.

여기서 추구하고자 하는 바는, 같은 데이터를 다루더라도, 사람이 편집하는 부분은 사람이 더 편안하게 느끼는 방식(문자열 등)으로 다루게 하고, 컴퓨터가 처리하는 부분은 컴퓨터에게 더 적합한 방식(숫자 등)으로 다룰 수 있게 둘 사이의 변환 과정을 자동으로 처리해주는 프로그램을 만들겠다는 뜻이다.

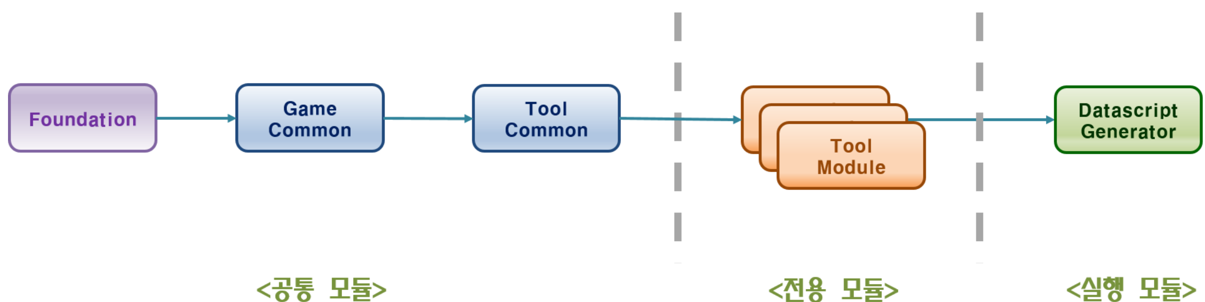
◆ A-1-2. 시스템 모델 관계

- **A-1-2-1.** 게임 데이터를 만들어내서 적용하기까지의 과정은 대략 다음과 같이 표현할 수 있다.



<게임 데이터를 만들고 그것이 각 서비스 모듈에 적용되는 과정의 표현>

- **A-1-2-2.** 데이터 스크립트 생성기 프로그램을 제작할 때의 시스템 내부의 코드 및 모듈의 구조는 다음과 같다.



<데이터 스크립트 생성기 프로그램의 내부 시스템 계층 구조>

◆ A-1-3. GUI 조감도

- **A-1-3-1.** 구조적으로도, 기술적으로도 최대한 단순하게 만드는 데 역점을 둔다.
  : 구현에 시간이 오래 걸리는 기능들은 최대한 배제한다.

- **A-1-3-2.** 요구하는 기능이 복잡하지 않기 때문에, 한 화면에 모든 기능을 다 집어넣을 수 있을 것으로 보인다.

Eternal Guardians Game Datascript Generator

모드 ☑ Client ☐ Server

데이터시트 불러오기

데이터시트 파일 형식 ☑ Excel2003XML ☐ Excel ☐ Libre Office Calc

데이터 스크립트 내보내기

데이터 스크립트 파일 형식 ☑ SCSV ☐ XML ☐ SQL

소스 파일 내보내기

소스 파일 형식 ☐ C++ ☑ C# ☐ Java

Action.xml
Character.xml
Item.xml
Skill.xml
Shop.xml
Stage.xml
Reward.xml

Export [Action.xml] -> [Action.txt]…
Success!
Export [Character.xml] -> [Character.txt]…
Success!
Export [Item.xml] -> [Item.txt]…
Success!
Export [Skill.xml] -> [Skill.txt]…
Success!
Export [Shop.xml] -> [Shop.txt]…
Success!

변환하기

<잘 하면 한 화면 안에 모든 기능이 다 들어갈 것 같다.>

※ GUI가 겹쳐야 하거나, 전환 과정이 없기 때문에 구현이 매우 간단해진다.
사용처가 제한적이기 때문에, 처음부터 큼직한 해상도를 사용해서 가급적 한 화면에 모두 구현하는 방향으로 한다.

A-2. 용어 설명

◆ A-2-1. 데이터(Data)

- **A-2-1-1.** 게임 프로젝트에서 사용하는 정보의 단위이다.
  : 사실, 게임 뿐 아니라 어떤 소프트웨어 프로그램에서도 이러한 정의가 통용된다.

- **A-2-1-2.** 일반적으로, 테이블 형태로 되어 있는 구조의 데이터시트나 데이터 스크립트에서 1줄(또는 1행)의 내용이 하나의 데이터를 표현한다.

- **A-2-1-3.** 어떤 데이터는 다른 데이터들 여러 개로 구성한 집합을 포함할 수도 있다.
  : 이런 경우, 테이블 형태로 되어 있는 구조의 데이터시트나 데이터 스크립트에서 여러 줄(또는 여러 행)의 내용들이 하나의 데이터를 표현한다.

- **A-2-1-4.** 소스 코드에서는 보통 하나의 구조체(struct) 혹은 클래스(class)로 표현한다.
  : 언어의 문법에 의해 구조체나 클래스가 또 다른 구조체나 클래스들을 하위 멤버 요소로써 가지고 있을 수 있으므로, 위 내용들은 소스 코드로 표현이 가능한 요소들이다.

◆ A-2-2. 게임 데이터(Game Data)

- **A-2-2-1.** 데이터들 중에서, 게임 플레이 내용을 조직하고, 각 단계 별 난이도와 능력, 보상 등을 구성하게 만드는 역할을 하는 데이터를 게임 데이터라고 한다.
  : 좀 더 정확하게 풀어서 쓰면, **'게임 구성 데이터(Game Configuration Data)'**라고 볼 수 있다.

※ 구성(Configuration)이라는 말 자체가 소프트웨어에서 뭔가 옵션을 조절하는 기능으로 자주 쓰이기 때문에, 오해를 피하기 위해 일부터 구성이라는 단어를 피해서 용어를 만들었다.
뭐, 옵션을 조절해야 하는 경우, '옵션 구성(Option Configuration)'이라는 말로 구분해줄 수 있긴 하지만...

- **A-2-2-2.** 게임 데이터는 일반적으로 사전에 정해진 규칙에 의해 정의하고 있는 숫자와 문자열들의 값으로 표현한다.

- **A-2-2-3.** 게임 데이터는 이진 파일 형식일수도, 텍스트 파일 형식일 수도 있다.

: 데이터의 내용 그 자체는 게임 데이터가 어떤 파일 형식을 가지는지 여부와 직접적인 관련이 없다.

※ 이진 파일 형식은 파일의 크기를 더 작게 만들 수 있고, 실행속도도 더 빠른 장점이 있다. 반면에, 프로그램이 해석하기 전까지는 사람이 그 내용을 확인하기 어렵다는 단점이 있다.
텍스트 파일 형식은 파일의 크기가 같은 내용의 이진 파일에 비해 훨씬 크고, 텍스트를 해석하는 과정이 더 느리지만, 사람이 내용을 파악하기 쉽고, 소스 관리 시스템을 이용한 협업에 더 적합한 장점이 있다.

- **A-2-2-4.** 게임 데이터가 가지는 내용은 대개 게임 플레이의 논리를 구성하기 위한 내용들이다.
: 이 내용들은 소프트웨어를 구동하는 과정과는 직접적인 관련이 없고, 게임 플레이 내용을 만드는 데 주요한 관심을 가지는 데이터들이다.

※ 경우에 따라, 게임 데이터들은 소프트웨어를 구동하기 위한 하드웨어 사양이라든가, 해상도 정보 등을 조절할 경우도 있지만, 절대적으로 많은 비율은 게임 플레이가 어떤 내용을 가져야 할지에 대한 내용을 지시하는 데 초점을 두고 있다.
그러므로, 게임 데이터들은 일반적으로 모든 게임 프로젝트마다 고유하며, 프로젝트 간 이식 가능성은 없다고 볼 수 있다.

◆ A-2-3. 데이터시트(Datasheet)

- **A-2-3-1.** 데이터시트는 사람이 자신이 생각하는 게임의 플레이 구성과 내용을, 게임 데이터들을 조직화하여 표현하기 위한 파일을 의미한다.

- **A-2-3-2.** 사람이 편집하고 알아보기 쉽게 하기 위해, 일반적으로 스프레드 시트(Spread Sheet)의 형식을 취한다.
: Microsoft Excel이 대표적이며, Libre Office Calc, 테이블 형태로 변환 가능한 XML 등도 모두 이 범주에 속한다고 볼 수 있다.

- **A-2-3-3.** 데이터시트에 고정된 파일 형식은 존재하지 않지만, 사람이 직접 편집할 수 있는 형식을 가진다.

※ Microsoft Excel 등 전용 프로그램이 필요한 스프레드 시트를 사용할 수도 있고, XML이나 Json 등 텍스트 에디터로 직접 편집이 가능한 텍스트 기반 구문을 가진 스크립트 형식을 사용할 수도 있다.
프로젝트 상황에 적합한 형식을 쓰도록 한다.

◆ A-2-4. 데이터 스크립트(Datascript)

- **A-2-4-1.** 게임의 각 서비스 모듈(클라이언트 프로그램, 서버 프로그램)들이 게임 데이터를 직접적으로 적재하기 위한 파일 형식을 데이터 스크립트라고 부른다.

- **A-2-4-2.** 데이터시트와 데이터 스크립트의 차이점은, 소프트웨어 프로그램이 게임 데이터를 가져오기 위해 어떤 파일을 직접 이용해서 불러오는지에 대한 여부다.
  : 소프트웨어 프로그램이 데이터를 불러오기 위해 직접 이용하는 파일은 데이터 스크립트다.

- **A-2-4-3.** 따라서, 데이터 스크립트는 사람이 보거나 편집하기 위한 편의성보다는, 컴퓨터 프로그램이 읽고 처리하는데 더 적합하도록 설계한다.

- **A-2-4-4.** : 그러나, 데이터 스크립트를 사람이 완전히 편집하지 못하는 방식으로만 제작하지는 않는다.
  : 단지, 데이터시트에 비해 사람이 다루기가 더 불편할 수도 있다는 뜻이다.

※ 이를테면, 사람이 편집해야 하는 데이터시트에는 값이 문자열로 표현되어 있지만, 데이터 스크립트는 컴퓨터가 읽어서 사용해야 하므로, 좀 더 능률적 처리하기에 적합하게 숫자로 변환해서 저장할 수도 있다.
비록, 데이터 스크립트가 텍스트 형식 기반이라고 하더라도, 수많은 데이터 필드가 숫자 값으로 뒤덮여 있다면, 사람이 편집한다는 건 가능은 하더라도 몹시 힘들 것이다.
당연한 말이지만, 사람과 컴퓨터 둘 다 능률적으로 다룰 수 있는 방식이 있다면, 당연히 그런 방식을 사용하는 게 좋다.

- **A-2-4-5.** 데이터 스크립트는 데이터시트의 내용을 완전하지 담고 있지 않을 수 있다.
  : 해당 데이터 스크립트를 이용하는 서비스 모듈의 프로그램 성격에 맞춰서, 꼭 필요한 정보들만으로 재구성해서 형성할 수 있다.

◆ A-2-5. 소스 파일, 소스 코드 파일(Source File, Source Code File)

- **A-2-5-1.** 게임 데이터의 내용을 소스 코드로 선언하고자 할 때, 원하는 프로그래밍 언어로 된 소스 파일을 데이터 스크립트와 함께 생성할 수 있다.

- **A-2-5-2.** 소스 파일을 생성하는 데 대한 프로그래밍 언어의 제약은 특별히 존재하지 않는다.
  : 현재는 Unity 3D 엔진으로 제작하고 있고, 이 엔진에서 C# 언어를 사용하고 있으므로, 소스 파일도 C# 파일(.cs 확장자를 가진다.)로 생성한다.

- **A-2-5-3.** 소스 파일을 데이터 스크립트와 함께 자동으로 생성하는 목적은, 해당 데이터시트의 데이터 모델이 변했을 때, 이를 소스 코드에도 자동으로 반영하기 쉽게 하기 위해서다.

※ 예를 들어, 캐릭터 능력치의 모델을 설계할 때, 처음에는 10개의 능력치로 디자인을 했다면, 그 당시의 프로그램에서는 캐릭터 데이터의 클래스는 10개의 능력치 변수로 구성되어 있어야 한다. 그래야 프로그램이 올바르게 동작하니까...
하지만 시간이 흐르고 난 뒤, 예전에 생각하지 못했던 5개의 능력치를 더 부여하면 게임이 좀 더 재미있고 내용이 풍부해질 것 같다는 판단을 했다고 치자. 그래서 데이터시트를 편집할 때, 기존의 10개 능력치가 있던 부분을 확장해서 15개의 능력치를 가지도록 만들었다.
그런데 이는 어디까지나 데이터시트를 편집했을 뿐이지, 프로그램의 내용을 바꾼 건 아니다.
새로운 프로그램이 이 바뀐 사항(10 종류의 능력치 -> 15 종류의 능력치)을 알아먹으려면 그 프로그램을 만들어낸 소스 코드 파일도 맞춰서 수정을 해줘야 한다.
데이터 변환기 프로그램의 목적 중 하나는, 이렇게 데이터시트의 값이 변경될 때 뿐 아니라, 데이터시트의 데이터 모델 자체가 변화하더라도, 그에 맞게 프로그램의 소스 코드도 맞춰주는 역할도 있다.

◆ A-2-6. 게임 자산, 에셋(Asset, Game Asset), 리소스(Resource)

- **A-2-6-1.** 여기서 말하는 자산(Asset)은, 게임을 구성하는 데 필요한, 사전에 저장매체에 저장한 파일들을 가리킨다.

- **A-2-6-2.** 저장매체로부터 불러들여서, 게임을 실행하기 위해 메모리에 올려두고, 이를 복제해서 같은 성질을 가진 여러 개의 독립 개체들을 만든 것들은 자산이 아니라 **객체(Object)** 혹은 **인스턴스(Instance)**라고 부른다.

- **A-2-6-3.** 데이터시트, 데이터 스크립트, 소스 파일도 게임 자산의 일종이다.
: 모두 디스크와 같은 저장 장치에 저장하므로 게임 자산으로 분류한다.

◆ A-2-7. 분할 시트(Partialsheet)

- **A-2-7-1.** 하나의 데이터시트 파일이 내부적으로 여러 개의 시트들로 구성되어 있을 때, 그 각각의 내부 시트들을 **분할 시트**라고 한다.
: 데이터시트는 일반적으로 사람이 편집하기 쉽도록 하기 위해 스프레드 시트 파일을 이용하여 작성하는데, 현대의 스프레드 시트들은 일반적으로 하나의 파일에 여러 개의 시트를 작성할 수 있도록 되어 있다. 이러한 분할 시트들은 보통 탭(Tab)으로 구분되어 있다.

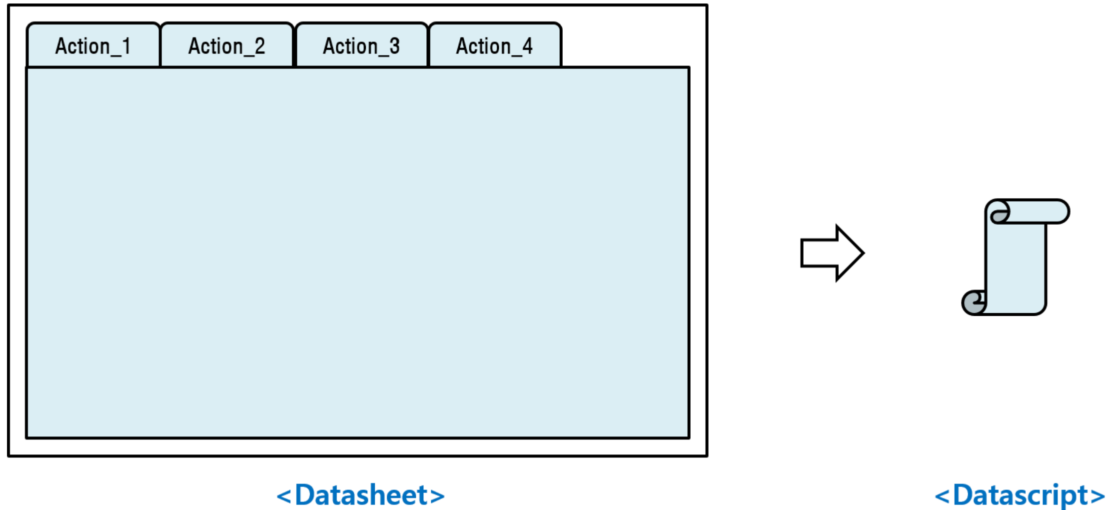
- **A-2-7-2.** 같은 데이터시트 파일을 이루는 각 분할 시트들은 완전히 같은 데이터 모델을 사용

하거나, 혹은 다른 데이터 모델을 섞어서 사용할 수 있다.

- **A-2-7-3. 동일 데이터 모델(Same Data Model)**

: 데이터시트의 각 분할 시트들은 완전히 동일한 데이터 모델(열의 개수와 성분이 모두 일치한다.)을 가진다.

이 경우, 데이터 스크립트로 변환할 때, 그 데이터시트의 각 분할 시트들은 모두 하나의 데이터 스크립트에 합한다.

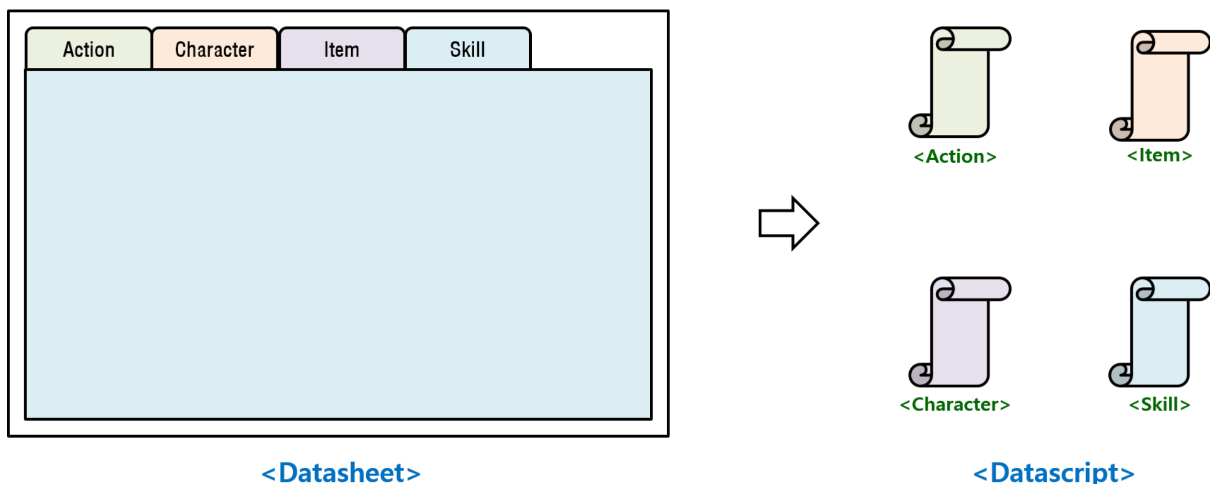


<Datasheet> <Datascript>

- **A-2-7-4. 분리 데이터 모델(Separated Data Model)**

: 데이터시트의 각 분할 시트들은 각자 데이터 모델이 다르다. (만약, 설령 외형이 같다 하더라도 이는 우연이며, 데이터 모델 간에는 상관 관계가 없다.)

이 경우, 데이터 스크립트로 변환할 때, 그 데이터시트의 각 분할 시트들은 각각 별도의 데이터 스크립트로 변환한다.



<Datasheet> <Datascript>

- **A-2-7-5.** 하나의 데이터 스크립트에 들어갈 내용이지만, 데이터시트에서는 데이터를 여러 분할

시트에 쪼개서 유지하고 싶으면, 반드시 동일 데이터 모델로 데이터시트를 표현해야만 한다.
: 왜냐하면, 데이터시트에서는 여러 개의 분할 시트로 표현했다 할지라도, 데이터 스크립트로 변환할 경우에, 데이터 스크립트에서는 그런 시트 탭 기능이 없어서 하나의 테이블 데이터로 묶어야 할 수 있기 때문이다.
이럴 때 분할 시트마다 데이터 모델이 다른 경우라면 **아예 데이터 스크립트도 별도의 파일로 저장해야** 할 것이다.

- **A-2-7-6.** 마찬가지로, 분리 데이터 모델을 채택하고 있는 경우에는, 같은 데이터 스크립트에 들어갈 내용을 여러 개의 분할 시트로 나눠서 구현할 수 없다.
: 이 때는 무조건 하나의 데이터 모델은 하나의 분할 시트 안에서만 표현해야 한다.

- **A-2-7-7.** 동일 데이터 모델과 분리 데이터 모델 간의 분할 시트 역할은 **반드시 양자택일해야만 하며, 양립할 수 없다.**

※ 지금 여기서 말하고 있는 분할의 개념이, 디스크 등의 저장 장치에 저장되는 파일 단위로 분할한다는 개념이 아니라는 사실에 주의해야 한다.
스프레드 시트가 내부적으로 여러 개의 시트 탭으로 이루어질 수 있다고 위에서 말했다. 여기서의 시트 분할이란, 하나의 스프레드 시트 파일 내에서 한 개의 시트 탭 만을 사용하는 게 아니라, 여러 개의 시트 탭으로 하나의 데이터시트를 표현하는 상황을 의미한다.

※ 데이터시트의 사용 방식은 사용자의 성향에 따라 꽤 다를 것이다.
어떤 사용자는 하나의 데이터시트 파일 안에 여러 개의 다른 데이터 모델의 데이터시트들을 섞어서 쓰는 방식을 좋아한다. 그렇게 하면 관리해야 할 데이터시트 파일의 개수를 효과적으로 줄일 수 있기 때문이다.

하지만 또 다른 사용자는 그런 방식을 좋아하지 않는다. 하나의 데이터시트에는 하나의 데이터 모델만 있어야 한다는 주의를 가지고 있다. 그런 사람들은 데이터시트 파일이 수십 개, 수백 개가 되는 상황을 기꺼이 감수한다. 다수의 작업자들이 데이터시트를 건드려야 할 경우, 파일 자체가 쪼개어진 편이 같은 파일을 수정해서 벌어지는 충돌상황을 더 효과적으로 관리할 수 있다는 논리다.
. 왜냐하면, 데이터시트를 누군가 한 사람이 편집하고 있다면, 다른 사람들은 데이터시트를 편집하는 사람이 작업을 마치고 저장소에 등록할 때까지는 갱신 내용이 충돌할까 염려하여 편집을 하기 어렵기 때문이다.

아무튼, 양쪽 다 장점과 단점이 있으며, 어떤 게 정답이라고 꼭 꼬집어 말하기는 어렵다. 그때그때의 상황과 선호도에 맞출 수 밖에 없다.

- **A-2-7-8.** 데이터시트의 분할은 **행(Row) 단위로만 가능**하다.
: 열(Column) 단위로는 분할할 수 없다.

※ 일반적인 용례에 비추어 보더라도, 행 단위로 분할하는 게 더 적합하다.
열 단위로 분할한다는 개념은, 말 그대로 기술적으로 행 단위를 분할할 수 있으니까 열 단위로도 가능할 수도 있다는 의미지, 실제로 작업할 때 이런 식으로 분할해야겠다고 마음먹는 경우는 아주 드물다. ~~(보통 그런 생각하는 사람을 4차원 인간으로 볼 것이다.)~~
물론, 분할을 할지 말지는 전적으로 데이터시트를 디자인하는 사람의 마음이다. 분할을 전혀 쓰지 않고도 데이터를 표현하는 데는 아무 지장이 없다.

◆ A-2-8. 테이블 데이터(Table Data)

- **A-2-8-1.** 행(Row)과 열(Column)으로 구성된, 표 방식으로 표현하는 데이터를 테이블 데이터라고 부른다.

- **A-2-8-2.** 이 프로젝트의 게임 데이터들은 모두 테이블 데이터 형식으로 표현해야 한다.

※ 기술적으로 반드시 테이블 데이터가 더 우수해서 그런 건 아니다.
오히려, 기술적인 면에서 봤을 때, 테이블 방식의 데이터는 복잡한 계통을 가지는 게임 데이터들을 표현하기에 좋지 않은 경우가 더 많다.
트리(Tree)의 단계 형식으로 표현하기에 적절한 게임 데이터들을 테이블 형식으로 표현하려면, 여러 개의 테이블 데이터들을 관계 지어서 표현해야 한다. 이 테이블 간의 관계라는 놈은 순전히 관념적인데다, 언뜻 보기에도 한 눈에 들어오지도 않는다. (이게 쉬운 거였으면 DBA를 개나 소나 했겠지...)
테이블 데이터를 많이 쓰는 이유는 사람이 사용하기가 가장 쉽고 편하게 느끼는 방식이라는 점이 더 크다. 그게 아니고 트리 방식으로 구현한 데이터를 사람이 사용하기 쉽게 하려면, 편집이 쉽도록 전용 도구를 만들어야 할 것이다. 하지만 대부분 비용 문제와 구현 난이도 문제 때문에, 해석기를 만들기 쉽고 구현도 쉬운 테이블 방식의 데이터를 선호하게 된다.

- **A-2-8-3.** 테이블 데이터는 사람이 편집할 때, 표(Chart) 형식으로 보이게 할 수 있어야 한다.
  : 스프레드 시트를 데이터시트로 주로 쓰는 이유는, 표 형식으로 보고 편집할 수 있는 가장 대중적인 프로그램이기 때문이다.

◆ A-2-9. 데이터 행(Data Row)

- **A-2-9-1.** 스프레드 시트에서 가로로 1줄을 데이터 행이라고 한다.

- **A-2-9-2.** 일반적으로, 데이터 행 1개는 1가지 종류의 게임 데이터를 구성한다.
  : 그 데이터가 캐릭터 데이터라면 캐릭터 1종류에 대한 데이터를, 아이템 데이터라면 아이템 1

종류에 대한 데이터를 표현한다.

Read	Comment	General_Type_Code	GTC_Key	Chance
Common	Common	Common	Common	Common
bool	string	EodGTC : primitive	EodGTC : key_define	percent
TRUE	1 번째 아이템 보상 데이터	1001	ItemReward001	74.5
TRUE	2 번째 아이템 보상 데이터	1002	ItemReward002	74.05
TRUE	3 번째 아이템 보상 데이터	1003	ItemReward003	73.6
TRUE	4 번째 아이템 보상 데이터	1004	ItemReward004	73.15
TRUE	5 번째 아이템 보상 데이터	1005	ItemReward005	72.7

< (일반적으로) 한 줄이 하나의 데이터를 표현한다. >

- **A-2-9-3.** 데이터 행 여러 개가 1가지 종류의 게임 데이터를 구성해야 할 경우도 있다.

: 보통 어떤 게임 데이터가 여러 개의 하위 게임 데이터로 구성되어야 할 경우에 그렇다고 할 수 있다.

※ 어떤 미션의 완료 보상을 정의한다고 하자.

미션의 완료 보상에 들어가는 아이템 종류가 3개라면, 어떤 식으로 표현하는 게 좋을까?

가장 손쉬운 방법은 <RewardItem-1>, <RewardItem-2>, <RewardItem-3>... 하는 식으로 데이터 열을 추가하는 방법이다. 하지만, 만약 미션의 완료 보상이 어떤 경우에는 1개고 어떤 경우에는 10개라면, 방금 말한 방법은 난관에 부딪친다. 보상 아이템이 1개인 미션 보상 데이터는 나머지 9개의 열은 아무짝에도 쓸모 없는 데이터가 되기 때문이다. (심지어 프로그램이나 데이터베이스에서는 이런 빈 칸도 용량을 차지해야 하기 때문에 더욱 문제가 된다.)

더구나, 만약 앞으로 기획에 따라 미션 보상 아이템이 최대 20개까지 가능해야 한다면...?

이런 문제를 해결하기 위해, 다른 방법으로 접근할 수 있다.

먼저 <Mission> 열과 <Index>, <Reward> 데이터 열을 정의한다. 그리고 **각 데이터 행은 특정 미션의 보상 아이템 1종류를 나타내는 것**으로 데이터 모델을 구성한다. 만약 보상이 여러 개라면, 데이터 행을 여러 개 넣는다. 대신, 그러한 데이터 행들은 <Mission> 열에 들어있는 값이 모두 같은 값이다. <Index>는 현재 데이터 행이 <Mission>의 몇 번째 보상인지 나타낸다.

이와 같은 방식이라면 미션에 따라 보상이 1개 밖에 없는 경우에는 데이터 행 1줄로, 5개면 데이터 행 5줄로, 보상이 10개면 데이터 행 10줄로 표현할 수 있다.

만약 기획에 따라 미션 보상 아이템을 최대 20개까지 줄 수 있어야 하는 경우에도, **전혀 테이블 데이터의 모델을 고칠 필요가 없다.** 보상 아이템이 20개면, 그냥 데이터 행 20개를 추가하면 된다!

◆ A-2-10. 데이터 열(Data Column)

- **A-2-10-1.** 스프레드 시트에서 세로로 1줄을 데이터 열이라고 한다.

- **A-2-10-2.** 각 데이터 열은 하나의 데이터를 구성하고 있는 각각의 데이터 필드(Data Field)들의 총합을 나타내는 표현이다.

- **A-2-10-3.** 줄(행, Row)이 다르고 열이 같은 데이터들은 같은 성질을 가지는 값들이다.
  : 그러나, 줄이 같아도 열이 다른 데이터들은 다른 성질을 가지는 값들이다.

Read	Comment	General_Type_Code	GTC_Key	Chance
Common	Common	Common	Common	Common
bool	string	EodGTC : primitive	EodGTC : key_define	percent
TRUE	1 번째 아이템 보상 데이터	1001	ItemReward001	74.5
TRUE	2 번째 아이템 보상 데이터	1002	ItemReward002	74.05
TRUE	3 번째 아이템 보상 데이터	1003	ItemReward003	73.6
TRUE	4 번째 아이템 보상 데이터	1004	ItemReward004	73.15
TRUE	5 번째 아이템 보상 데이터	1005	ItemReward005	72.7

< 특정 열에 속하는 다른 줄의 데이터들은 같은 성질을 가진다. >

- **A-2-10-4.** 데이터 열들은 **서로 순서를 바꿀 수 없고, 값도 교환할 수 없다.**
  : 데이터 열의 순서나 값이 바뀔 경우, 데이터 모델의 규칙을 깨는 행위가 된다.
  반면, 데이터 행의 순서나 값을 바꾸는 경우는 단지 읽는 순서만 바뀔 뿐, 데이터 모델의 규칙을 깨는 행위가 아니다.

※ 어떤 테이블이 [1열 – int], [2열 – float], [3열 – string]의 규칙으로 구성되어 있다고 하자.
그런데 2열의 값을 1열과 바꾸고 싶다고 가정해보도록 하자.

이럴 경우, 원래의 2열은 본래 소수점을 허용하는 값을 받아들이는 열이고, 1열은 정수형만 허용하는 열이기 때문에 문제가 된다.
결과적으로, 데이터시트를 해석할 때 오류가 나지 않으면 다행이고, 오류가 나지 않는다고 해도 소수점 부분은 모조리 잘리게 된다.
반면, 데이터 행 단위로는 어떻게 변경을 하더라도 [1열 – int], [2열 – float], [3열 – string]의 데이터 모델 규칙을 깨지 않는다. 이 때는 단지 먼저 읽어 들일지, 아니면 나중에 읽어 들일지 차이 밖에 없다.

◆ A-2-11. 데이터 필드(Data Field)

- **A-2-11-1.** 데이터 필드는 프로그램 소스 코드에서, 구조체나 클래스의 멤버 변수 1종을 나타낸다.

- **A-2-11-2.** 데이터시트에서는 특정 데이터 행에 위치한 특정 데이터 열이 데이터 필드 하나를 나타낸다.
  : 이는 마치 2차원 지도에서 x, y 좌표로 어떤 지점을 표현하는 방식에 비유할 수 있다. 좌표에 의해 가리키고 있는 지점이 바로 데이터 필드에 해당하는 지점이다.

- **A-2-11-3.** 간혹 데이터 열과 혼용해서 사용하기도 한다.
  : 이 때는 대개 데이터 열의 개념에 가깝게 사용한다.
  값의 표현 형식을 말하는 것이냐(데이터 열), 구체적인 값이냐(데이터 필드)의 뉘앙스 차이가 있다고 보면 된다.

◆ A-2-12. 목록 컨테이너

- **A-2-12-1.** 같은 성질을 가진 값들을 일렬로 나열한 데이터 필드의 값들을 목록 컨테이너(List Container)로 부른다.

- **A-2-12-2.** 프로그래밍 자료 구조의 관점에서는 정적 배열(Array)과 동적 배열(Dynamic Array, List<T>) 둘 다 포함하는 관념이다.
  : 그러니까, 여기서의 표현 방식으로는, 원소를 일렬로 표현하기만 하다면, 프로그램 내부적으로 이게 메모리를 정적으로 쓰는지 동적으로 쓰는지는 알 바가 아니다.

- **A-2-12-3.** 이 게임 프로젝트의 데이터시트에서는 값의 목록 컨테이너를 데이터 필드의 값으로 지정할 수 있다.
  : 그라니까, 표 1칸 안에 1, 2, 3, 4, 5… 와 같은 값 목록을 넣는 기능을 지원한다는 의미이다.

◆ A-2-13. 사전 컨테이너(Dictionary Container)

- **A-2-13-1.** 검색 키(Key) – 값(Value) 쌍으로 이루어진 값들의 목록을 사전 컨테이너라고 한다.

- **A-2-13-2.** 프로그래밍 자료 구조에서의 사전 컨테이너와는 다르게, '정렬(Sorting)' 부분에 대해서는 전혀 고려하지 않는다.
  : 즉, 데이터시트 단계에서는 자료 값들을 사전 방식으로 정렬하는 데이터 컨테이너가 있다는

것만 알지, 그게 어떻게 정렬되는지는 전혀 신경 쓰지 않는다.

- **A-2-13-3.** 데이터 필드의 값으로 표현할 수 없다.
: 데이터 필드의 값은 목록 컨테이너만 가능하다.

- **A-2-13-4.** 데이터시트에서 사전 컨테이너를 직접 데이터 필드의 값으로 넣어서 사용하지는 못한다.

- **A-2-13-5.** 다만, 이를 표현하는 부분은 다름 아닌, 각 데이터 행이 어떤 데이터를 나타내는지 가리키는 내용이 들어갈 경우다.
: 식별 코드가 100인 아이템은 100으로 검색하면 반드시 식별 코드가 100으로 정해진 아이템 데이터가 나오고, 식별 코드가 200인 아이템은 200으로 검색하면 반드시 식별 코드가 200으로 정해진 아이템 데이터가 검색된다.

◆ A-2-14. 일반 타입 식별 코드(General Type Code, GTC)

- **A-2-14-1.** 게임 플레이를 구성하는 데이터들을 의미에 따라 종류를 구분하기 위해서 사용하는 번호 체계이다.

- **A-2-14-2.** 정수 방식의 숫자로 구성한다.

- **A-2-14-3.** GTC에 대한 상세한 내용은 프로젝트의 공식 명세서(Specification - Eternal Guardians)에서 관련 내용을 참조한다.

※ 명세서는 다양한 문서 포맷을 지원한다.
: Microsoft Word 형식(*.docx), Libre Office Writer 형식(*.odf), PDF 형식(*.pdf)을 지원한다.

◆ A-2-15. GTC 키(GTC Key)

- **A-2-15-1.** GTC 키는 GTC 값과 연결되어 있는 문자열 값을 말한다.

- **A-2-15-2.** 정수 형식으로 표현하는 GTC를 사람이 다루기 편하도록 하기 위해 GTC를 특정한 문자열로 연결한 개념이다.

- **A-2-15-3.** GTC가 각각의 GTC 값끼리 유일함을 유지하는 것처럼, GTC 키 역시 각각의 GTC에 대해 고유한 문자열로 정의해야 한다.

- **A-2-15-4.** GTC 키는 데이터시트와, 데이터시트를 해석해서 데이터 스크립트로 변환하는 도구에서는 확실히 사용한다. 하지만, 데이터 스크립트나, 이것으로부터 파생된 소스 파일에는 포함되지 않을 수 있다.
  : 왜냐하면 GTC 키는 근본적으로 사람이 보고 편집할 때의 편리함을 위해 만들어진 개념이기 때문이다. 컴퓨터 프로그램은 문자열보다 숫자 값이 훨씬 처리하기에 능률적이다.

A-3. 제품 사용 시나리오

◆ A-3-1. 데이터시트의 작성

- **A-3-1-1.** 모든 과정에서 최초로 만들어야 하는 데이터시트는, 사람이 직접 수동으로 데이터를 모델링하고 작성해야 한다.
  : 데이터 모델링 그 자체는 사람이 해야만 하고 아직은 기계가 해줄 수는 없다. (이 과정에 사람의 개입이 필요 없다면, 결과적으로 프로그래머도 존재할 필요가 없을 것이다.)

- **A-3-1-2.** 데이터시트는 일반적인 스프레드 시트 프로그램을 이용해서 작성한다고 가정한다.
  또한, 이러한 스프레드 시트 프로그램을 이용해서 작성한 데이터시트를 해석할 수 있는 프로그램 기술과 관련 기능들을 보유했다고 가정한다.

- **A-3-1-3.** 게임에 적용할 데이터시트들은 하나의 디렉토리에 모아놓고 작성한다.
  : 그래야 나중에 데이터 스크립트 생성기가 변환할 데이터시트들을 찾기가 쉽다. 경우에 따라, 데이터 스크립트 생성기가 **데이터시트들을 하나의 디렉토리에 모아놓도록 강제해도 무방**하다.

- **A-3-1-4.** 데이터시트를 작성할 때, 데이터 스크립트 생성기가 데이터시트의 내용을 해석할 수 있게 도와주는 역할을 하는 몇 가지 필수적인 공통요소들을 준수해서 작성한다.
  : 구체적인 내용은 관련 항목에서 더 자세히 서술한다. 단지, 여기서 중요한 점은 모든 데이터시트들은 공통의 템플릿으로 작성해야 한다는 점이다.

- **A-3-1-5.** 데이터 스크립트 생성기가 여러 파일 형식의 데이터시트를 해석할 수 있더라도, 하나의 프로젝트에서는 **하나의 파일 형식으로만 데이터 스크립트를 작성**해야 한다.

◆ A-3-2. 데이터시트 내용을 데이터 스크립트로 변환할 환경 설정

- **A-3-2-1.** 데이터 스크립트 생성기 프로그램을 켠다.

- **A-3-2-2.** 데이터 스크립트 생성기는 이전에 작업했던 데이터시트 디렉토리 경로에 대한 정보를 가지고 있다면, 그 디렉토리 경로의 파일 정보를 자동으로 가져온다.
  만약 데이터 스크립트 생성기 프로그램을 처음 시작하거나, 이전에 작업했던 데이터시트 디렉토리 경로 정보가 없다면, 프로그램이 들어 있는 디렉토리 경로의 파일 정보를 보여준다.

- **A-3-2-3.** 디렉토리 및 파일 정보를 보여줄 때에는, 오직 데이터시트와 디렉토리만 사용자에게 보여준다.
 : 다른 파일들은 어차피 이 프로그램에서 다룰 수도 없고, 다룰 이유도 없기 때문에 사용자에게 보여줘야 할 필요가 없다.

- **A-3-2-4.** 사용자에게 보여줄 파일 중에서, 유니티 엔진의 메타 파일들(*.meta 형식)은 보여줄 대상에서 제외한다.

- **A-3-2-5.** 데이터시트를 어떤 서비스 모듈을 타겟으로 하여 변환할 것인지 설정한다.
 : 클라이언트 용으로 변환할지, 서버 용으로 변환할지 설정한다.

- **A-3-2-6.** 데이터시트가 어떤 형식으로 되어 있는지 지정한다.
 : 스프레드 시트도 어떤 제품을 사용했는지에 따라 해석을 처리하는 라이브러리나 관련 코드가 달라야 한다.

- **A-3-2-7.** 변환할 데이터 스크립트의 파일 형식을 지정한다.
 : 데이터 스크립트를 어떤 파일 형식으로 사용할지는 상황에 따라 다를 수 있다.

※ 파일 형식은 상황에 목적에 따라 적합한 형식이 다를 수 있다. 어떤 경우에는 XML을 쓸 수도 있고, 어떤 경우에는 Json을, 또는 그냥 일반 텍스트 파일(*.txt)을 사용할 수도 있다. 아니면 심지어 SQL로 작성해서 데이터베이스 형태로 데이터 스크립트의 데이터를 적용할 수도 있다.
 또는 데이터시트와 데이터 스크립트가 완전히 같은 파일 형식을 사용하는 경우도 가능하다.
 (이때는 양쪽 파일들은 내용의 차이점만 존재할 것이다.)

- **A-3-2-8.** 변환할 데이터 스크립트의 파일 형식은 현재의 모든 데이터시트 변환에 공통적으로 사용해야 한다.
 : 이 파일은 XML로 변환하고, 저 파일은 Json으로 변환하는 방식은 허용하지 않는다.

- **A-3-2-9.** 데이터시트를 데이터 스크립트로 변환할 때, 소스 파일도 같이 생성해줄 것인지 여부를 설정한다.

- **A-3-2-10.** 데이터시트를 데이터 스크립트로 변환할 때 생성하는 소스 파일의 프로그래밍 언어를 지정한다.

- **A-3-2-11.** 변환할 때 생성하는 소스 파일의 프로그래밍 언어는 현재의 모든 데이터시트 변환에 공통적으로 적용하여야 한다.
 : 이 데이터시트를 변환할 때는 소스 파일을 C++ 파일로 생성하고, 저 데이터시트를 변환할 때는 소스 파일을 자바 파일로 생성하는 등의 일은 할 수 없다.

◆ A-3-3. 데이터 스크립트 변환

- **A-3-3-1.** 데이터 스크립트 변환 명령을 내리면, (버튼을 터치하든, 명령을 호출하든) 지정한 디렉토리에 있는 모든 데이터시트들이 데이터 스크립트와 소스 파일로 일괄적으로 변환을 시작한다.

- **A-3-3-2.** 일부 데이터시트만 변환하는 기능은 제공하지 않는다.
  : **무조건 모든 데이터시트를 한꺼번에 변환하여야 한다.**

※ 일부 데이터시트 변환 기능을 구현하지 않는 게 꼭 기술적인 한계나 편의 때문만은 아니다.
현재 데이터시트는 테이블 데이터 형식으로 표현하고 있고, 그렇기 때문에 많은 수의 테이블 파일들이 간접적인 관계를 이루며 맺어져 있다.
이런 상태에서, **일부 데이터시트만 내용을 고친다고 해서, 다른 데이터시트에 영향이 없으리라는 보장이 없다!** 어떤 데이터시트가 내가 고친 데이터의 내용이나 검색 키 필드를 참조하고 있을지 모르기 때문이다. 특히, GTC나 GTC 키를 바꿨다거나, 해당 GTC의 데이터를 지우거나 새로 추가한 경우는 더욱 그렇다. 그러니, 차라리 데이터시트에 뭔가 변경이 가해졌다면, **모든 데이터시트의 내용이 변경되었다고 가정하고 처리하게 하는 편이 낫다.**

- **A-3-3-3.** 데이터 스크립트의 변환 과정은, 원하는 대상 서비스 모듈 별로 따로 진행해야 한다.
  : 클라이언트 대상 변환과 서버 대상 변환은 한꺼번에 수행할 수 없다는 뜻이다.
  반드시 클라이언트 용 변환 과정을 따로 수행해야 하고, 서버 용 변환 과정을 따로 수행하도록 해야 한다.

※ 이 부분은 기술적인 편의 때문이 맞다.
클라이언트 모듈과 서버 모듈에서 필요로 하는 데이터 스크립트의 파일 형식과 소스 파일 형식 등이 서로 다를 경우, 데이터시트를 해석하면서 한 번에 두 가지 파일 형식을 만들어내는 과정이 쉽지 않기 때문이다.
기술적으로 가능하다고 해도, 그 난이도에 비해 얻는 이득이 그다지 크지도 않다.
이를 프로그래밍 언어의 컴파일러에 비유하자면, 마치 컴파일 버튼 한 번 누르면 x86 CPU 명령어를 쓰는 파일과, x64 CPU 명령어를 쓰는 파일을 컴파일 한 번에 동시에 나오게 만들어야 하는 것과 같은, 그런 방식이기 때문이다. 하지만, 본인이 아는 한, 어떤 컴파일러도 이런 식으로 동작하는 경우는 없다. x86 컴파일 따로, x64 컴파일 따로 순차적으로 진행한다.
누가 어떤 프로그램 도구를 만드는 경우라도, 이럴 때는 순차적으로 한 번에 하나씩 처리하게 만들 것이다.

◆ A-3-4. 프로젝트에 적용하기

- **A-3-4-1.** 데이터시트를 데이터 스크립트로 변환하기 전에, 변환이 끝난 데이터 스크립트들과 소스 파일들을 어느 디렉토리 안에 생성할지 지정할 수 있다.

- **A-3-4-2.** 데이터 스크립트를 생성할 디렉토리와, 소스 파일들을 생성할 디렉토리를 따로 지정할 수 있다.

- **A-3-4-3.** 모든 데이터시트들은 자신들이 변환한 결과물(데이터 스크립트 + 소스 파일)을 생성할 디렉토리 경로를 공유한다.
  : 데이터 스크립트를 생성하는 디렉토리 경로를 지정하면, 모든 데이터시트들은 그 디렉토리에 데이터 스크립트를 생성한다. 소스 파일을 생성하는 디렉토리 경로 역시 마찬가지로 모든 데이터시트들이 지정한 디렉토리에 소스 파일을 생성한다.

※ 데이터시트마다 변환한 결과물을 지정하는 디렉토리를 따로 지정하게 만들 수도 있겠지만, 그런 방식의 기능은 만드는 노력에 비해 사용 빈도가 높을 것 같지 않다.
 왜냐하면, 대부분의 게임 프로젝트에서는 데이터시트를 하나의 디렉토리에 두고 쓰는 게 일반적이다. 그게 가장 사용하기에 편하기 때문이다. 어떤 데이터 스크립트를 찾는 데 있어, 아이템 데이터 스크립트가 있는 폴더가 따로 있고, 캐릭터 데이터 스크립트가 있는 폴더가 따로 있고, 스테이지 데이터 스크립트가 있는 폴더가 따로 있게 만드는 경우는 별로 많지 않다.
 이런 점은 소스 파일의 경우에도 마찬가지다. 큰 범주에서 관련이 있는 소스 파일끼리 묶어서 하나의 폴더에 들어가도록 보관하는 방식이 가장 일반적이고 인지하기 쉽다.
 위와 같은 점을 고려하면, 데이터시트마다 하나하나 변환 결과물이 생성될 경로를 지정하게 하는 것보다, 모든 데이터시트가 하나의 경로에 변환 결과물들을 생성하게 만드는 게 더 좋다.
 그러는 편이 개발 난이도에 있어서도 이득이고, 실제 사용에 불편함도 없을 것이다.

- **A-3-4-4.** 데이터시트의 변환 결과물들은 프로젝트에 적용하기 위한 별도의 과정이나 명령 메뉴는 존재하지 않는다.

- **A-3-4-5.** 단지, 그저 해당 프로젝트에서 데이터시트로부터 변환한 결과물들을 집어넣어야 하는 디렉토리를 지정하는 행위가, 프로젝트에 결과물을 적용하는 방법의 전부이다.
  : 즉, 데이터 스크립트 생성기 프로그램은, 데이터시트의 변환 결과물을 필요로 하는 프로젝트에 대해서 '알지는 못한다.'
  그냥, 데이터가 위치해야 하는 적절한 경로만 지정하면, 그것으로 프로젝트에 변환 결과물을 적용하는 준비가 끝나는 셈이다.

※ 데이터시트의 변환 결과물을 생성한다는 건, **그저 운영체제 상에 새로운 파일을 생성하는 과정일 뿐**이다.
 그 때문에, 아무데나 원하는 디렉토리에 변환 결과물을 생성하게 하고, 나중에 그 변환 결과물을 수동으로 복사하거나 잘라서 적용한다 할지라도 결과는 완전히 똑같다.

◆ A-3-5. 빌드 자동화에 사용할 때

- **A-3-5-1.** 데이터 스크립트 변환기 프로그램은 **명령행 인터페이스(Command Line Interface, CLI) 방식으로 호출해서 사용할 수 있다.**
  : 명령행으로 사용할 수 있어야, 빌드 자동화 도구를 통해 통합하기가 쉽다.

- **A-3-5-2.** 게임 데이터 스크립트 생성기의 **핵심 기능들은 전부 라이브러리 화** 되어 있다.
  : 따라서, 필요한 플랫폼, 필요한 GUI 프로그램에 통합해서 사용할 수도 있게 제작할 것이다.

※ 핵심 기능 전체를 라이브러리로 동작하게 하려고 했던 이유는, 이게 설계상으로 더 우수하다고 생각했던 면도 있지만, GUI 프로그램을 무엇으로 사용해야 할지 명확하게 정하지 못했기 때문이기도 하다.
현재, 툴의 GUI는 유니티 엔진과 QT Designer, wxWidget, GTK# 등을 놓고 고민 중이다.
만약 상황에 따라, 이도 저도 아닌 경우에는, 라이브러리를 아예 프로젝트의 제품 기능 중 일부로 통합해서 호출하게 만들 수도 있으니, 핵심 기능 전부를 라이브러리로 제작하는 데는 이점이 더 많다 할 수 있다.

B. 기능 요구사항

B-1. 데이터시트

◆ B-1-1. 파일 관련

- **B-1-1-1.** 일차적으로 Excel 2003 XML 형식(*.xml)의 데이터시트를 읽어오는 기능을 지원한다.
  : 현재 기존에 구현되어 있는 방식이 데이터시트를 Excel 2003 XML로 만들고, 이것을 읽어오는 방식으로 되어 있다. 따라서 가장 안정적으로 구현할 수 있다.

※ 다만, Excel 2003 XML이라는 파일 형식 자체가 문제가 좀 많다.
이름만 봐서도 알 수 있듯, Microsoft Excel 2003 시절에 만들어진 것으로 보인다. 참고로, 지금은 Excel 2016이 출시된 시점이다. (...)

사실, 이것 자체는 별 문제는 아니다. *.xls이나 *.xlxs 파일 형식도 몇 년씩 써먹었으니까...
진정한 문제는, 저 파일 형식이 분명 Excel의 형식이기는 하지만, **요즘 Excel 프로그램에서는 호환이 잘 안 되고 있다**는 점에 있다.
즉, 만든 지 오래 된 물건인데다, 하위 호환에 이제는 크게 신경을 안 쓰는, **버려진 물건으로 보인다.**
현재 주력으로 쓰고 있는 Microsoft Excel 2013에서도 열 때마다 경고 창이 뜨는 경우가 많으며, OSX용 Microsoft Excel 2011에서는 **일부 파일은 아예 열리지도 않는 등** 장기적으로 사용하기가 어렵다고 판단한다.

비록 파일 형식이 XML 기반이기 때문에 텍스트 편집기로도 다룰 수 있다는 장점이 있지만, 실제로는 이런 장점을 써먹을 경우는 사실상 없다고 봐도 좋다.
왜냐하면, 이런 게임 데이터들은 한 번에 대량으로 수정하는데다, '최신 버전'이 뭔지는 작업자들만 알기 때문이다. 그러다 보니, 사실상 허울만 좋은 텍스트 형식을 뿐, 기계적인 병합은 불가능한 게 현실이다. (보통 한 쪽은 버리고 다른 한 쪽 내용을 취하는 방식으로 일하게 된다.)

여담으로, 게임 클라이언트를 제작하는데 사용하고 있는 Unity 3D 엔진에서도 비슷한 현상이 일어나고 있다.
*.unity 파일이라든가 프리팹 파일들은 엔진 도구에서 전용 이진 파일 형식과, yaml 기반의 텍스트 형식의 저장을 둘 다 지원한다. 그런데, 100% 텍스트 방식으로 저장한다고 해도, 사실상 이들 파일들은 텍스트 편집기로 편집하거나 병합하기는 대단히 어렵다.
심하면 수만 줄에 달하는 텍스트들 각각이 툴에서 어떤 식으로 작동하는지 정확하게 알기는 어렵고, 잘못 건드리면 파일 자체가 망가져 버릴 수도 있다. 툴에서는 사소한 변경임에도 불구하고, yaml 텍스트 상에서는 수십 줄 ~ 수백 줄의 변경을 가하는 경우도 흔하다.

아무튼, 이런 식이다 보니, Excel 2003 XML이라는 형식은 Excel의 본래 파일 형식인 *.xls나 *.xlsx보다 딱히 나은 점이 없는 상황이 되고 말았다.

- **B-1-1-2.** 이차적으로 Excel 확장 형식(*.xlsx)의 데이터시트를 읽어오는 기능을 지원한다.
 : Microsoft Excel의 현재 공식적인 파일 형식이다.

※ Microsoft Excel의 대표적인 공식 파일 형식이기 때문에(사실, Excel로 만들어내거나 열 수 있는 스프레드 시트 관련 파일 형식은 한두 개가 아니다.), 가장 안정적으로 작동한다. OSX에서도 깨끗하게 문제 없이 열고 편집이 가능하다.

- **B-1-1-3.** (선택적 기능)가능하다면, Libre Office의 Calc 형식(*.ods)의 데이터시트를 읽어오는 기능을 지원한다.

※ Libre Office의 가장 좋은 점은 **공짜**라는 점이다.(...)
 이미 리눅스 진영의 대표적인 오피스 프로그램인데다, 전세계 사용자들이 가장 많이 쓰는 운영체제들인 Windows, OSX, Linux에서 모두 사용이 가능하다는 장점이 있다.
 단점은 현재 프로젝트의 주력 언어인 C# 언어와 .NET Framework 기반에서 Libre Office 문서를 처리하는 안정적인 개발 도구가 좀 부족하다는 점이다. (아마 찾아보면 없지는 않을 거다...)

- **B-1-1-4.** 분할 시트를 사용할 경우, 각 분할 시트의 데이터 열 형식들은 분할 시트마다 완전히 일치해야 한다.
 : 하나의 시트로 나타내기에 데이터가 세로로 너무 길어질 경우에, 적당히 나눠서 편집하는 용도로 사용해야 한다.

※ 컴퓨터 프로그램은 데이터시트를 분할하든 아니든 읽는 데 아무런 상관이 없다.
 반면, 사람은 문서가 지나치게 한쪽 방향으로 길어지면 내용을 파악하는데 지장이 있다. 분할 시트는 이런 단점을 상당히 보완해줄 수 있는 편리한 기능이다. 그리고, 딱 그 용도까지지만 사용하도록 제한하기 위해 분할 시트의 형식 제한을 두고 있다.
 만약 분할 시트의 각 데이터 열 형식이 모두 다를 수 있다면, 극단적으로 말해 모든 게임 데이터시트가 하나의 데이터시트 파일 안에서 분할 시트로 다루어질 수도 있음을 의미한다. 이는 기술적으로 가능은 하지만, 본래 분할 시트를 사용하는 취지에는 맞지 않다.

◆ B-1-2. 데이터 형식

- **B-1-2-1.** 데이터시트에서 필드 값으로 사용할 수 있어야 하는 데이터 형식들은 다음과 같다.

데이터 형식	코 드	설 명

부울형	bool	• True(= 1), False(= 0) 값만 존재한다.
정수형	sbyte, short, int, long 등	• 0, 100, -200, 238123과 같은 숫자를 사용할 수 있다..
실수형	float, double	• 0.1, 112.432, -10.543과 같은 소수점이 있는 숫자를 사용할 수 있다.
퍼센트형	percent	• 실수형의 한 변형이다. • 표기 단위를 퍼센트로 하여 값을 넣는다. • 데이터 스크립트로 저장할 때는 0.01을 곱하여 비율 값으로 저장한다. • 값은 float과 같다. • 편집하는 사람의 편의를 위해 추가한 데이터형이다.
문자열	string	• 말 그대로 문자열을 사용할 수 있다.
열거형	enum : ? : ?	• 프로그램 소스에서 미리 정해진 항목 중에서 값을 선택하는 방식이다. • 예를 들어, 프로그램에서 미리 Apple, Grape, Melon과 같은 값을 정했다면, 그것을 데이터시트에서 사용할 수 있다.
목록 방식	list<?>	• 위 항목의 데이터들을 하나의 필드에서 여러 개 묶어서 사용할 수 있다. • 이를 테면, '1, 2, 3, 4, 5', 'a, b, c, d, e'... 처럼 말이다.

- **B-1-2-2.** 데이터시트에 기록하는 **모든 값들은 원칙적으로 모두 문자열 값**이다.
  : 게임 데이터 스크립트 해석기 프로그램에서는 모든 값들을 일차적으로 문자열로 읽어 온 뒤, 경우에 맞게 알맞은 값 타입으로 해석한다.

- **B-1-2-3.** 정수형과 실수형, 열거형들은 프로그램 코드에서 64비트를 초과하는 형식을 사용할 수 없다.
  : C#에는 decimal이라는 128비트 형식 숫자가 있는데, 이런 형식은 사용할 수 없다.

- **B-1-2-4.** 데이터에서 음수를 제외하고 싶은 경우에는 다음 키워드를 사용한다.
  : 단, 이는 정수형에만 해당하며, 실수형에는 해당 사항이 없다.

데이터 형식	양수형	크기
sbyte	byte	1바이트(8비트)
short	unsigned short 또는 ushort	2바이트(16비트)
int	unsigned int 또는 uint	4바이트(32비트)
long	지원하지 않음	8바이트(64비트)

※ long 형의 경우, 양수만 가능한 unsigned 형을 지원하지 않는다.
이렇게 하는 이유는, 일반적으로 데이터시트로 사용하는 **스프레드시트 형식들의 내부적으로 프로그램 한계** 때문이다.

이를 설명하기 위해, 가장 널리 사용하는 스프레드시트인 ©Microsoft Excel을 예로 들겠다.
Excel의 각 셀(Cell)에는 데이터를 입력할 수 있는데, 이 셀은 프로그래밍 언어와는 다르게 숫자 값에 대해 정수형이니 실수형이니 몇 바이트니 하는 식으로 타입을 세밀하게 구분하지는 않는다. 이들은 그저 **'숫자형(Number)'**으로 구분할 뿐이다.

그러다 보니, 프로그램 내부적으로는 각 셀에 들어 있는 숫자 값들을 가장 폭넓게 구현할 수 있는 형식, 즉 double 형식으로 구현하는 것으로 보인다.
double은 64비트의 크기를 가지는 부동소수점 형식이다. 이 값들은 극도로 큰 수나 작은 수를 표현할 수 있는 장점이 있지만, 이 극도로 크거나 작은 수에 가까이 갈수록 입력한 값과 실제 처리하는 값에 오차가 발생하는 문제가 있다. 이걸 부동소수점 수의 정밀도(Precision) 문제라고 한다.

Microsoft Excel의 경우, IEEE 754 규정에 따라 부동소수점을 처리하고 있으며, 그에 따라 **최대 소수점 15자리까지의 정밀도만을 보장**하고 있다. (단위로는 백조 단위이다.)
(이에 대한 관련 기술 문서는 https://support.microsoft.com/ko-kr/kb/78113 를 참조한다.)
기술 문서에 따르면, 타 스프레드시트 제품들 역시 유사한 문제를 보인다고 한다.

이와 같은 이유 때문에, 스프레드시트에서 정확하게 처리할 수 있는 정수형 값의 한계는 ±999,999,999,999,999(약 999조)의 범위 이내다.
long 형식의 정수형 처리 범위는 대략 ±922경 정도이기 때문에, 위 한계 값 범위를 모두 포함할 수 있다. 따라서 굳이 unsigned long 형식이 필요가 없다.

- **B-1-2-5.** 정수형은 컴퓨터의 처리능력과 관계 없이 **±999,999,999,999,999**의 범위까지만 사용할 수 있다.
  : 일반적으로 스프레드시트의 각 셀이 처리할 수 있는 정수 숫자의 정확도는 여기까지만 보장하도록 되어 있기 때문이다.

- **B-1-2-6.** **32비트 실수형(float)은 소수점 3자리**까지, **64비트 실수형(double)은 소수점 6자리**까지 사용하는 것을 권장한다.
  : 둘 다 부동소수점 수 형식이기 때문에, 정밀도(precision)에 관한 문제가 있다.

※ 물론, 경우에 따라 float임에도 불구하고 소수점 4자리 쓰는 경우도 있을 수 있다. 계산하다 보면 0.0125같은 값이 나오지 않으리라는 보장이 없으니까... 보통 정밀도의 문제는 극단적으로 큰 수나 극단적으로 작은 수를 부동소수점 수로 표현하고자 할 때 발생한다.

일반적으로 float은 소수점 6자리까지는 크게 문제 없이 사용할 수 있는 것으로 알려져 있다.
double의 경우에는 (우주탐사선 프로그램이나 회계 프로그램 같은 게 아니라면) 일반적인 사용에서 정밀도 때문에 문제가 되는 경우는 거의 없다.

- **B-1-2-7.** 실수를 표현할 때, 고정소수점 방식은 지원하지 않는다.
: 아까도 나왔던 C#의 decimal이 바로 고정소수점 수를 표현한다.

※ 고정소수점 수는 주로 속도보다 정밀도 이슈가 더 큰, 통화를 다루는 재무계산 용도 등에서 사용한다.
게임에서도 숫자 값의 계산이 정밀하면 좋기야 하겠지만, 게임 소프트웨어라는 게 그렇게 극단적으로 값의 정밀성을 요구하는 환경이 아니다. 그러므로, 처리 속도를 더 중시해서 부동소수점 수만 사용하기로 한다.
통화(Currency)는 게임에서도 (살짝) 처리하는 요소 중 하나이기는 하지만, 어차피 현실의 화폐가 아니기 때문에, 심각하게 다루지는 않는다. 단지, 게임에서 다루는 화폐들은 중간 계산 값들을 제외하고는, 항상 정수 형식으로만 다루어서 부동소수점 문제를 회피한다.

※ 위 문제 말고도, 데이터시트에서 사용하는 스프레드시트 자체의 정밀도 한계 문제도 있다.
대부분의 스프레드시트들은 IEEE 754 규정에 따라 부동소수점을 처리하기 때문에 최대 소수점 15자리까지만 정밀도를 보장하고 있다.
더구나, 스프레드시트들은 부동소수점은 텍스트로 표현할 경우에 지수 형식(a.aaaaa × Ebb같은 형식)으로 저장한다. 그런데, C# 언어의 decimal과 같은 고정소수점 타입은 이런 지수 형식의 텍스트들을 해석하지 못한다. (사실 이건 부동소수점 방식과 고정소수점 방식의 차이를 고려한다면 당연한 일이기도 하다.)

- **B-1-2-8.** 문자열은 **서명 없는 UTF-8(ISO-65001, No Byte Order Mark)** 인코딩 형식으로 처리하는 것을 원칙으로 한다.
: 데이터시트에 입력하는 문자열 값들은 프로그램 내에서 사용하는 이진 타입이어야 할 수도 있지만, GUI 등으로 노출해야 하는 텍스트의 값일 수도 있다. 이걸 데이터시트 단계에서 사전에 구분할 방법은 없다.
따라서 모든 문자열 값들은 반드시 유니코드 이슈를 고려해야 한다.

◆ B-1-3. 일반적인 문법 형식

- **B-1-3-1.** 데이터 형식의 키워드들은 **대소문자를 구분하지 않는다.**
: sbyte, int, float, long 등은 SBYTE, INT, FLOAT, LONG으로 사용해도 무방하다.

- **B-1-3-2.** 숫자 값이 아닌 경우에도 enum, string, list의 키워드는 대소문자를 구분하지 않고 사용할 수 있다.

※ 가만 보면, 이름이 항상 고정되어 있어서 변할 일이 전혀 없는 키워드들만 대소문자 구분 없이 사용할 수 있음을 알 수 있다.

- **B-1-3-3.** **프로그램의 이름공간(namespace), 사용자 타입, 클래스 이름을 사용하는 경우에는 이름의 대소문자를 구분**해야 한다.
  : 예를 들어, enum : sbyte : Eod.Define.Character.eClass는 enum과 sbyte는 ENUM, SBYTE로 대소문자 구분 없이 사용할 수 있지만, Eod.Define.Character.eClass 부분은 반드시 대소문자를 지켜서 사용해야 한다.

- **B-1-3-4.** 부울 형식의 값은 대소문자 구분 없이 사용할 수 있다.
  : true, false, True, False, TRUE, FALSE 모두 유효한 값이다.

- **B-1-3-5.** 부울 형식의 값은 숫자로 사용할 수도 있다.
  : 이 때는 0(False), 1(True)의 값만 사용할 수 있다.
  그러나, 정수형 데이터와 쉽게 혼동하므로 사용은 그다지 권하지 않는다.

- **B-1-3-6.** 부울 형식이 아닌 값들은 대소문자를 지켜서 입력해야 한다.
  : 특히 열거형 부분이 그러하다.

◆ B-1-4. 목록 방식의 자료형

- **B-1-4-1.** 하나의 필드에 여러 개의 자료를 넣어야 하는 경우가 종종 있다. 특히, **'매개변수가 몇 개가 필요한지 그때그때 달라야 하는 경우'**에 주로 매개변수들을 목록으로 만들어서 넣는 기능이 특히 유용하다.

※ 데이터의 모델은 같은데 매개변수가 어떤 경우에는 3개가 필요하고 어떤 경우는 5개가 필요한 경우가 있을 것이다. 그리고 그러한 경우의 가짓수가 데이터시트 상에서 매우 많다면 테이블 모델을 일일이 분리해서 데이터시트를 만들기도 부담스럽지 않을 수 없다.
이럴 때는 필드 하나에 여러 개의 데이터를 묶어서 보낼 수 있게 하는 편이 더 낫다.

- **B-1-4-2.** 목록 자료형의 선언 방식
  : 목록 자료형을 사용하는 형식은 다음과 같다.

```
list<$(자료형의 이름)>
```

- **B-1-4-3.** 특히, 매개변수들을 보내는 개수 뿐 아니라, 형식조차도 각 데이터마다 달라질 수 있는 경우에는 매개변수들을 모두 문자열 형식으로 전달하는 기법(list<string>)을 쓸 수 있다.

: 이 경우, 매개변수들을 적절하게 해석하는 일은 데이터시트의 데이터들을 받아들이는 객체에게 그 책임이 있다.

- **B-1-4-4.** 일반 타입 식별 코드(General Type Code, GTC)를 목록으로 보내야 하는 경우에는, 그 GTC가 원시 타입(숫자 형)인지 참조키 타입(문자열 형)인지 콜론(':')으로 구분해야 한다.
: 구체적인 방식은 다음과 같다.

• 원시 타입일 경우
list<EodGTC> : primitive

• 참조키 타입일 경우
list<EodGTC> : key_reference : $(데이터 객체의 이름)

- **B-1-4-5.** 만약 아무런 추가 정보 없이 list<EodGTC>라고만 사용할 경우에는 list<EodGTC> : primitive와 동일하다.

- **B-1-4-6.** 목록 자료형으로 값 넣기
: 어떤 필드에 값을 목록으로 넣으려고 할 경우, 여러 개의 값을 쉼표(',')로 구분해서 기입한다.

- **B-1-4-7.** 예를 들어, 1 ~ 5를 목록으로 필드에 값을 할당하려고 할 경우, 다음처럼 표기한다.

```
1, 2, 3, 4, 5
```

- **B-1-4-8.** 5개의 문자열을 목록으로 구성해서 필드에 값을 넣는 경우, 다음처럼 표기한다.

```
apple, peach, melon, grape, tomato
```

- **B-1-4-9.** 쉼표와 쉼표 사이에는 공백이나 줄바꿈을 할 수 있다.
: 아래 예시는 둘 다 규약에 맞는 표현이며 해석 결과도 같다.

```
apple, peach, melon, grape, tomato


apple,
peach,
melon,
grape,
tomato
```

- **B-1-4-10.** 콜론(':')을 이용해서 목록의 개별적인 원소 값에 참조할 수 있는 이름을 부여할 수

있다.
: 참조 이름은 데이터시트 편집자의 편의를 위한 기능이며, 데이터 스크립트나 소스 파일로 변환할 때는 전혀 이용하지 않는다.

- **B-1-4-11.** 목록 필드에서의 참조 이름은 다음과 같은 형식으로 부여한다.

```
$(값) : $(이름), $(값) : $(이름), $(값) : $(이름)...
```

예시는 다음과 같다.

```
1 : apple, 2 : peach, 3 : melon, 4 : grape, 5 : tomato

또는 다음과 같이 표기해도 된다.

1 : apple,
2 : peach,
3 : melon,
4 : grape,
5 : tomato
```

※ 위의 예에서, apple, peach 등의 문자열 값들은 데이터 스크립트나 소스 파일에서는 전혀 나타나지 않는다.
변환 과정을 거친 데이터 스크립트에서는 그냥 1, 2, 3, 4, 5의 값으로만 보인다.

◆ B-1-5. 열거형 처리

- **B-1-5-1.** 열거형은 미리 정의되어 있는 몇 개의 값 중 하나를 선택하는 방식이다.
: 그래서 프로그램 내부에서 실제로는 숫자값으로 작동하지만, 데이터시트에 기입할 때는 사전에 값과 연결하도록 정의한 문자열을 쓴다.

- **B-1-5-2.** 열거형 세부 규약은 프로그램 소스 코드 단계에서 사전에 정의해야 하기 때문에, 새로운 열거형이 필요한 경우에는 개발자 간에 이 부분을 먼저 조율하고, **새 열거형 타입을 소스 코드에 반영하는 작업이 우선되어야 데이터시트에서도 사용할 수 있다.**

- **B-1-5-3.** 데이터시트에서 열거형 타입을 선언하는 형식은 다음과 같다.

```
enum : $(정수 타입) : $(열거형의 이름)
```

- **B-1-5-4.** **$(정수 타입)**
: 해당 열거형이 정수 값으로 어느 정도의 범위를 사용하는지 적는다. 대개는 sbyte, short 둘 중 하나이다.

- **B-1-5-5.** **$(열거형의 이름)**
: 열거형의 이름을 쓴다.

◆ B-1-6. 일반 타입 식별 코드(General Type Code, GTC)의 처리

- **B-1-6-1.** GTC를 데이터시트의 필드에 사용할 때는, GTC를 사용할 열에서 값 타입을 정의하는 부분(데이터시트의 4번째 줄)을 다음 중 하나로 지정한다.

- **B-1-6-2.** **EodGTC : primitive**
: GTC를 정의할 때 사용한다. GTC는 본래 int의 값 범위를 가지는 숫자이다. 따라서, EodGTC : primitive를 사용하는 필드의 값은 숫자를 적어야 한다.

- **B-1-6-3.** **EodGTC : key_define**
: EodGTC : primitive로 정의한 타입 식별 코드 숫자를 특정한 문자열 값과 결합할 때 사용한다.
따라서 EodGTC : primitive로 GTC를 정의한 열과 다른 열에 사용해야 하고, 항상 EodGTC : primitive로 정의한 열과 같이 사용해야 한다.

※ 실제로, 모든 데이터시트를 살펴봐도, GTC : primitive를 쓰는 열이 있으면, 반드시 그 옆에 EodGTC : key_define을 쓰는 열이 같이 있다.

- **B-1-6-4.** **EodGTC : key_reference : $(Datasheet Name)**
: 데이터시트의 한 항목 열을 다른 데이터시트에 정의한 데이터와 관계짓고 싶은 경우, 그 열의 데이터는 참조할 다른 데이터시트의 GTC를 값을 필드에 넣도록 설계를 하게 된다.
이 때, 그 GTC를 GTC 본래의 값인 int 숫자 값으로 받는 대신, EodGTC : key_define으로 정의한 GTC 문자열 키를 값으로 받게 하기 위해서 이와 같은 방식으로 값 형식을 선언한다.
$(Datasheet Name)는 어떤 데이터시트에서 정의한 GTC인지를 지정하기 위해서 필요하다. GTC는 가급적 전역적으로 유일한 값을 가지도록 권장하지만, 이를 강제하지는 않기 때문에, 반드시 어느 데이터시트의 GTC인지를 알아야 한다.
데이터시트는 사람이 수동으로 편집하는 경우가 대부분이기 때문에, 숫자보다는 문자열로 GTC를 구분하게 하는 편이 편의성이나, 사용성에 있어 훨씬 더 좋다.

※ 다만, 프로그램에서는 GTC 값이 숫자인 편이 용량이나 속도 면에서 더 낫기 때문에, 툴에서 데이터시트를 데이터 스크립트로 변환할 때는, 문자로 참조한 GTC들은 전부 GTC의 본래 숫자 값으로 변환하여 저장한다.

◆ B-1-7. 여러 줄의 내용을 하나의 데이터로 구성하기

- **B-1-7-1.** 행과 열을 가지는 테이블 방식의 데이터는 좀 더 복잡한 계통 단계를 가지는 데이터들을 효과적으로 표현하는 데 한계를 가지고 있다.

- **B-1-7-2.** 그래서, 행과 열의 모양을 가지는 한계 안에서 더 복잡한 계통 단계를 표현하기 위해, **여러 줄이 하나의 데이터를 표현하도록 하는 방식**을 고안하곤 한다.

광전사 스킬 4	berserker_skill_4	0	PlaySound
광전사 스킬 4	berserker_skill_4	1	ShakeCamera
광전사 스킬 4	berserker_skill_4	2	PlayEffect
광전사 스킬 4	berserker_skill_4	3	PlayEffect
광전사 스킬 4	berserker_skill_4	4	PlayEffect
광전사 스킬 4	berserker_skill_4	5	PlayEffect
광전사 스킬 4	berserker_skill_4	6	PlayEffect
광전사 스킬 4	berserker_skill_4	7	PlayEffect
광전사 스킬 4	berserker_skill_4	8	PlayEffect
광전사 스킬 4	berserker_skill_4	9	PlayEffect
광전사 스킬 4	berserker_skill_4	10	PlayEffect
광전사 스킬 4	berserker_skill_4	11	PlayEffect
광전사 스킬 4	berserker_skill_4	12	PlayEffect
광전사 스킬 4	berserker_skill_4	13	PlayEffect
광전사 스킬 4	berserker_skill_4	14	PlayEffect
광전사 스킬 4	berserker_skill_4	15	PlayEffect
광전사 스킬 4	berserker_skill_4	16	PlayEffect
광전사 스킬 4	berserker_skill_4	17	PlayEffect
광전사 스킬 4	berserker_skill_4	18	PlayEffect
광전사 스킬 4	berserker_skill_4	19	PlayEffect

<위 20 줄은 하나의 액션을 연출하기 위한 이벤트의 목록이다.>

- **B-1-7-3.** 이런 방식의 데이터를 앞으로 **다중 행 데이터(Multi Line Data, MLD)**로 부른다.
  : 반대 급부로, 한 줄이 데이터 하나를 나타내는 경우는 **단일 행 데이터(Single Line, Data, SLD)**라고 부르면 된다.

- **B-1-7-4.** 이 프로젝트의 데이터시트에서, 여러 줄로써 하나의 데이터를 표현하게 만들기 위해서는, **특정한 열의 값을 기준으로 삼아야 한다.**

: 그래야 어떤 값을 기준으로 하여 여러 줄이 하나의 데이터를 나타내는지 알 수가 있다.

※ 위 예제에서는 <berserker_skill_4>라는 필드 열의 값에 의해, 20개의 줄이 하나의 데이터를 표현하도록 규정하였다.

- **B-1-7-5.** 다중 열 데이터를 표현하는 기준이 되는 열(Column)은, 다중 열 데이터의 **기준 열 (Pivot Column)**이라고 부른다.

- **B-1-7-6.** 기준 열의 데이터 타입 키워드는, 뒤에 **[] 기호**를 붙힌다.
: 그래야 데이터 스크립트 생성기 프로그램이 해당 열이 다중 열 데이터임을 알 수 있다.

※ 즉, 기준이 되는 값을 가진 열의 데이터 타입 키워드가 int인 경우, int[]로 표시해야 한다.
만약 string이라면 string[]이 된다.
GTC인 경우에는 EodGTC[] : primitive, EodGTC[] : reference : $(Defined File Name), EodGTC[] : key_reference : $(Defined File Name) 처럼 표현하면 된다.

- **B-1-7-7. list<> 키워드와 [] 키워드를 혼동하면 안 된다.**
: list<> 키워드는 해당 열의 값 하나가 배열 컨테이너의 형식으로 표현된다는 뜻이다.
반면, [] 키워드는 해당 열의 값을 기준으로 하여 여러 개의 줄이 데이터 하나를 구성한다는 뜻이다.

- **B-1-7-8.** 다중 행 데이터에 들어갈 각 행들은 인위적으로 컨테이너에 들어갈 수 있는 순서를 조정할 수 있다.
: 이를 위해서는, 기준 열의 값 뒤에 **'[인덱스 번호]'**의 방식으로 인덱스를 직접 입력해준다.

Action_Event	Fixed_Damage	AdFactor	ApFactor	Add_Option_Code	Cooldown_Time
Client	Client	Client	Client	Client	Client
EodGTC[] : key_reference : Skill	List<float>	float	float	float	float
171100001[0]	100, 100	0.5	0.5	0	3
171100001[1]	100, 100	0.5	0.5	0	3
171100003[0]	100, 100	0.5	0.5	0	3
171100003[1]	100, 100	0.5	0.5	0	3
171100003[2]	100, 100	0.5	0.5	0	3

<가장 처음 열이 기준 열이다. 인덱스 입력 방식이 보일 것이다.>

- **B-1-7-9.** 기준 열의 각 값에 인덱스 번호를 부여할 경우, 인덱스는 **0번이 시작 번호이고, 1씩 증가하는 양의 정수**로 순서를 매겨야 한다.

※ 이는 일반적인 C 계열의 문법을 따르는 프로그래밍 언어에서 배열 컨테이너의 인덱스를 매기는 방법과 완전히 같다.

- **B-1-7-10.** 인덱스 값은 암묵적으로 4바이트 정수 값(int)으로 간주한다.
: 따라서, 인덱스의 최대 한계는 약 21.47억이다.

※ 실질적으로 인덱스가 이 정도 한계에 다다라야 할 일은 없다고 봐도 좋지만, 인덱스가 실제로는 무한대가 아니라는 점은 알고 있어야 한다.
여기서 뿐만 아니라, 대부분의 경우에 뭔가 배열 컨테이너의 인덱스를 취급한다고 하면, 4바이트 정수 값의 한계를 가지고 있다고 보면 된다.

- **B-1-7-11.** 기준 열의 값은 인덱스를 매기지 않아도 된다.
: 단, 이 경우에는 데이터시트에 입력한 순서가 곧 인덱스가 된다.

- **B-1-7-12.** 다중 행 데이터는 내부적으로 배열 혹은 이와 유사한 컨테이너 객체로 표현한다.
: 일반적으로 가장 효과적인 수단은 동적 배열 컨테이너 객체이다. (C++ 언어의 std::vector<>, 또는 .NET Freamework의 List<> 객체 등)

- **B-1-7-13.** 다중 행 데이터의 컨테이너를 구성하는 데이터들은, 기준이 되는 열에서 순서 상으로 나머지 뒷부분 열에 있는 데이터들 전부이다.

※ 어떤 데이터시트가 총 10개의 열로 구성이 되어 있다고 가정하자. 이 중에서, 만약 3번째 열이 다중 행 데이터를 구성하는 기준 열로 설정되어 있다고 치자.
그러면, 다중 행 데이터는 4번째 ~ 10번째열까지의 데이터로 구성이 된다.
즉, 4열 ~ 10열까지의 데이터들은 3열에서 같은 값을 가지는 여러 줄의 데이터들은 모두 하나의 데이터를 표현하는 배열 컨테이너의 원소들이라는 말이다.

- **B-1-7-14.** 다중 행 데이터는 또 다른 다중 행 데이터를 멤버로 가질 수 있다.

※ 단, 이 구현은 시급한 사항은 아니다.
대부분의 상황에서, 다중 행 데이터는 데이터시트에서 한 가지만 필요할 때가 많다. 따라서 이 기능을 위한 구현을 위한 시간이 부족한 경우에는, 다중 행 데이터가 데이터시트에서 하나만 있다고 가정하고 관련 프로그램을 구현해도 좋다.

- **B-1-7-15.** 다중 행 데이터가 데이터시트에 여러 개 존재할 때, 다중 행 데이터끼리의 소유 관계는 **다중 행 데이터 기준 열의 순서**에 달려 있다.
: 즉, 다중 행 데이터를 설정하는 기준 열이 여러 개일 때, 먼저 등장하는 기준 열이 나중에 등장하는 기준 열을 멤버로 가지는 관계다.

※ 프로그래밍 언어 단계에서 생각해보면, 이 상황이 마치 컨테이너 객체가 컨테이너 객체를 소유하고 있는 경우에 비유할 수 있다.
A라는 상위 객체가 B라는 하위 객체를 소유하는 개념 그 자체는 개발자에게 있어 아주 익숙한 개념이다.

Account	Password	CharacterID	Nickname	JobClass	ReservedItem	ItemCode
Common	Common	Common	Common	Common	Common	Common
string[]	string	Long[]	string	enum : sbyte : eJobClass	Int[]	int
god01[0]	12345	324[0]	상남자	Worrior	0[0]	100
god01[0]	12345	324[0]	상남자	Worrior	0[1]	200
god01[0]	12345	324[1]	상남자	Worrior	1[0]	200
god01[0]	12345	324[1]	상남자	Worrior	1[1]	100
god01[1]	12345	325[0]	원혼	Archer	0[0]	110
god01[1]	12345	325[0]	원혼	Archer	0[1]	210
god01[1]	12345	325[1]	원혼	Archer	1[0]	210
god01[1]	12345	325[1]	원혼	Archer	1[1]	110
god01[2]	12345	326[0]	You'reFired	Wizard	0[0]	120
god01[2]	12345	326[0]	You'reFired	Wizard	0[1]	220
god01[2]	12345	326[1]	You'reFired	Wizard	1[0]	220
god01[2]	12345	326[1]	You'reFired	Wizard	1[1]	120
god01[2]	12345	326[2]	You'reFired	Wizard	2[0]	220
god01[2]	12345	326[2]	You'reFired	Wizard	2[1]	221

<하나의 데이터 시트에 여러 다중 행 데이터가 들어 있을 경우의 모습>

※ 상단의 데이터시트 예제는 하나의 데이터시트에 다중 행 데이터가 여러 단계로 나타날 경우를 가정한 ~~(다분히 의도적이고 무리한 데이터 구조를 가진)~~ 예제이다.
붉은색 테두리를 가지고 있는 열은 모두 다중 행 데이터의 기준 열이다.
즉, <Account> 열의 'god01' 값을 가진 데이터들은 <Password> ~ <ItemCode> 열의 4줄에 걸친 데이터들로 'god01' 계정 정보를 구성한다는 뜻이다.
그러나, 실제로는 캐릭터 별 정보를 <CharacterID> 열 기준으로, 아이템 구성에 대한 예약 정보를 <ReservedItem> 열 기준으로 구성하였으므로, 구조적으로는 다음과 같은 계통을 가지게 된다.

1) <ReservedItem> 열을 기준으로 하여 <ItemCode> 열의 각 1줄마다 나눈 데이터
2) 1)에서 구성한 데이터들을 <CharacterID> 열을 기준으로 하여, <ReservedItem> 열의 값에 따라 각자 데이터를 소유한다.

즉, <CharacterID> 기준 열은 <Nicknames> ~ <ItemCode> 간 데이터를 멤버로 가진다.

3) 2)의 단계에서 구성한 데이터들을 <Account> 열을 기준으로 하여, <CharacterID> 열의 값에 따라 각자 데이터를 소유한다.

즉, <Account> 기준 열은 <Password> ~ <ItemCode> 간 데이터를 멤버로 가진다.

말로 풀어서 설명하기에는 쉽지 않지만, 어쨌든 기준 열이 3가지 이고, 이 3가지의 기준 열은 데이터시트에서 (좌측으로부터의) 순서가 어떤지에 따라 계통 서열이 정해진다.
이 데이터를 코드로 읽어온다면 대략 다음과 같은 모습이 될 것이다.

• <Account> 열을 기준으로 하는 최상위 객체

```
class Account
{
      string account;
      list<Character> characters;
}
```

• <CharacterID> 열을 기준으로 하는 차상위 객체

```
class Character
{
      long id;
      string nickName;
      eJobClass jobClass;
      list<ReservedItem> reservedItems;
}
```

• <ReservedItem> 열을 기준으로 하는 최하위 객체

```
class ReservedItem
{
      int reservedIndex;
      list<int> itemCodes;
}
```

- **B-1-7-16.** 다중 행 데이터의 일부 행을 데이터 스크립트에 포함하지 않도록 주석처리 했다면 다음과 같이 동작한다.
: 주석 처리란, 모든 데이터시트의 가장 첫 번째 열에 오도록 되어 있는 <Read> 필드의 값이 False(대 / 소문자 구분은 없다.) 또는 0으로 설정되어 있는 경우를 말한다.

• 다중 행 데이터마다 명시적으로 인덱스를 지정해놓았을 경우, 그 인덱스가 그대로 유지되지 않는다.

: 주석 처리를 하였으므로, 명시한 인덱스보다 실제 데이터 스크립트에 탑재하고 이용하는 인덱스는 더 작은 값을 가진다.

- 다만, 명시적으로 지정한 인덱스의 **'순서'**만큼은 유지된다.

※ 다중 행 데이터로 캐릭터의 스킬 이벤트 연출 과정을 표현했다고 가정해보자.
연출 과정은 원래는 총 10개의 행으로 구성되어 하나의 스킬에 대한 이벤트를 연출했었다.
이 때, 각 스킬 이벤트들은 <SkillCode> 필드에 100[0], 100[1], 100[2]... 100[9]까지 다중 행 데이터의 인덱스를 부여하였다. 데이터 타입은 int[EventInfo]로 지정되어 있었다.
그런데 제작하던 중간에 마음이 바뀌어서, 혹은 테스트를 위해서인지 중간에 100[4](다섯 번째 데이터)와 100[5](여섯 번째 데이터)의 이벤트를 주석 처리하기로 마음먹었다. 그렇다면, 주석 처리가 이루어진 이후에, 데이터시트를 데이터 스크립트로 변환하고 나면 <SkillCode> 값이 100인 스킬 이벤트의 다중 행 데이터 값들은 다음과 같이 취급한다.

우선, 100[0] ~ 100[3]까지는 데이터시트에 기재한 인덱스의 순서와 같다.
하지만 100[4], 100[5] 데이터는 주석 처리하였으므로 데이터 스크립트에는 포함하지 않는 데이터가 되어 버렸다. 그러면 원래의 100[4], 100[5]가 있어야 하는 자리에는 그 다음 데이터, 즉, 원래 데이터시트에서는 100[6], 100[7]로 인덱스를 부여한 데이터가 차지한다. 물론, 100[6], 100[7]의 데이터들이 주석 처리한 행이 아니기 때문에 가능하다. 이것마저 데이터시트에서 주석 처리한 행이라면 또 그 다음 인덱스 순번을 가진 데이터들에게로 자리가 넘어간다.
즉, 데이터시트에서 주석 처리로 인해 비어 있게 되는 인덱스들은, 그 자리를 채우도록 뒤의 순위 인덱스를 가진 데이터들이 앞으로 채워지는 것이다.

그래서 최종적으로는 데이터 스크립트 변환 과정 이후에는 <SkillCode> 값이 100인 스킬의 이벤트 데이터는 총 8개가 된다. 하지만, 이벤트들은 중간에 빈 자리가 없도록 앞으로 이동했을 뿐이고, 주석 처리하지 않고 포함한 이벤트 행들은 그 순서가 뒤바뀌지 않았다.

B-2. 데이터 스크립트

◆ B-2-1. 파일 형식

- **B-2-1-1.** 클라이언트 측에서는 SCSV 형식과 XML 형식으로 데이터 스크립트 파일을 생성할 수 있게 한다.
  : SCSV 형식의 구현이 우선이다.

※ SCSV 형식은 '세미 콜론으로 구분된 변수들(**S**emi**C**olon **S**eperate **V**ariables)'이라는 파일 형식의 축약형이다.
이 파일은 비 표준 파일 형식이기 때문에 어디서 이런 파일 형식을 찾으려고 하지 말 것.
스프레드 시트를 텍스트 에디터로 편집 가능한 파일 형식으로 내보낼 때 자주 사용하는 CSV(Comma Separate Variables) 파일 형식과 매우 흡사하다.
CSV 파일과 다른 점은 CSV 파일이 쉼표(,)로 각 필드를 구분하는 데 비해, **세미콜론(;)으로 각 필드를 구분한다**는 점에 있다.

- **B-2-1-2.** 서버 측에서는 MySQL과 호환되는 SQL 형식과 XML 형식으로 데이터 스크립트 파일을 생성할 수 있게 한다.
  : SQL 형식의 구현이 우선이다.

- **B-2-1-3.** 추출하는 데이터 스크립트 파일의 인코딩 형식은 **서명 없는 UTF-8(ISO 65001, No Byte Order Mark)** 인코딩 형식이어야 한다.

◆ B-2-2. 변환 관련 주의 사항

- **B-2-2-1.** **데이터시트 파일 하나당, 데이터 스크립트 파일 하나**로 추출하는 것을 원칙으로 한다.
  : 분할 시트의 경우, 같은 데이터시트 내에서 분할 시트의 데이터 열 형식이 모두 같으므로, 하나의 데이터 스크립트로 합하여 추출할 수 있다.

- **B-2-2-2.** 데이터 스크립트를 SQL 형식으로 추출할 경우, **데이터베이스 테이블 전체를 교체하는 방식으로 SQL을 작성**하도록 해야 한다.

※ 만약 기존 내용을 수정하는 SQL로 작성할 경우, 모든 수정 이력에 관한 SQL들을 전부 보관하

고 순차적으로 적용하게 만들어야 한다.
이렇게 만들기도 힘들 뿐 아니라, 데이터 스크립트는 사용자 데이터가 아니기 때문에 기존 내용과 호환성을 보장하거나, 기존 내용을 보존해야 할 필요가 없다. 그러니, 데이터 스크립트로 사용하는 데이터베이스 테이블들은 교체할 때마다 해당하는 DB 전체를 삭제하고 새로운 내용으로 다시 생성하는 게 더 적합하다.

- **B-2-2-3.** 데이터 스크립트로 사용하는 데이터베이스 테이블의 필드에는 **외래 키 제약(Foreign Key Constraint)을 걸지 말아야 한다**.
: 외래 키 제약이 걸려 있을 경우, 데이터 스크립트로 사용할 데이터베이스 테이블을 교체할 때, 삭제하는 순서를 반드시 지켜야 하기 때문에, 작업을 자동화하기 어렵게 만든다.

※ 데이터 스크립트는 그 용도상 전체를 한꺼번에 메모리에 불러온 뒤에 사용한다. 또한, 도중에 그 내용을 읽어오기만 할 뿐, 갱신해야 할 필요가 없다.
이런 특성 때문에, 데이터 스크립트로 사용하는 데이터베이스 테이블은, 데이터베이스에서의 외래 키 제약으로부터 아무런 이점도 얻을 수 없다.

- **B-2-2-4.** 서버의 경우, 사용자 데이터를 담고 있는 데이터베이스들이 데이터 스크립트 용 데이터베이스의 키를 외래 키로 참조하지 못하게 해야 한다.
: 역시 마찬가지 이유다.
외래 키 제약(Foreign Key Constraint)을 걸어 버리면, 데이터 스크립트 용 DB 테이블을 교체하려고 할 때, 사용자 테이블의 데이터를 삭제해야만 하는 사태가 발생할 수 있다!

※ 외래 키 제약은 사용자 데이터가 있는 DB 테이블끼리 걸어야 한다.
최우선 키(Primary Key)나 인덱스는 걸어도 상관은 없는데, 어차피 전체 내용을 메모리로 불러온 뒤에 처리하는 방식이기 때문에 별 의미가 없는 건 마찬가지다.

B-3. 소스 파일

◆ B-3-1. 파일 형식

- **B-3-1-1.** 현재로써는 C# 파일 형식(*.cs)만 지원한다.
  : 게임 서비스 모듈들이 전부 C#으로 구현되고 있기 때문이다.

- **B-3-1-2.** 소스 파일의 텍스트를 인코딩하는 형식은 **서명 있는 UTF-8(ISO 65001, Byte Order Mark)** 방식이어야 한다.

※ 사실, 유사 유닉스(Unix-like) 계열 운영체제(Linux, BSD, OSX 등)에서는 서명 없는 UTF-8 인코딩이 대세이긴 하다.
 그러나 이 부분은 단지 프로젝트의 엔진으로 사용하고 있는 유니티 3D 엔진 때문에 예외로 하고 있다.
 OSX 버전의 유니티 3D 엔진에서 사용하는 컴파일러가, 서명 없는 UTF-8 인코딩으로 작성한 소스 파일들 안에 영어가 아닌 문자가 섞였을 때, 소스 해석에 문제를 일으키기 때문이다.

- **B-3-1-3.** 소스 파일에 사용할 이름은 대 / 소문자를 구분해도 되지만, 대 / 소문자만 다르고 단어는 완전히 같은 이름의 파일을 만들면 안 된다.
  : Windows는 파일 경로와 이름의 대 / 소문자를 구분하지 않는다.

- **B-3-1-4.** 소스 파일의 코드 내용을 자동으로 생성할 때, 주석을 제외하고 컴파일러가 해석해야 하는 모든 내용들은 반드시 영문으로 출력하게 해야 한다.
  : 명명하는 클래스 이름, 멤버 변수 이름 등이 모두 해당한다.

◆ B-3-2. 변환 관련 주의사항

- **B-3-2-1.** **데이터시트 하나 당 데이터 클래스 하나**를 만들어내도록 한다.

※ 데이터시트 당 1개의 클래스 구현은 가장 혼동이 적고 직관적인 방식이다.
 데이터 주도적인 부분의 코드를 작성할 때는, 객체의 추상화에 지나치게 집착하지 않는 설계가 전반적으로 관리하기에 좀 더 낫다.

- **B-3-2-2.** GTC와 GTC Key가 있는 데이터시트는, 데이터시트 내에 있는 모든 GTC 값을 상수로 추출해야 한다.

- **B-3-2-3.** GTC 값을 상수로 추출할 경우, GTC Key의 값은 소스 코드 상 그 상수의 이름이 된다.

- **B-3-2-4.** GTC 값을 상수로 추출할 때, 별도의 클래스에 추출할 수 있다.
  : 단, GTC 상수 값을 정의하는 클래스는 해당 데이터시트의 클래스와 연관이 있는 이름이거나, 분할 클래스(partial class)여야 한다. (분할 클래스는 C#처럼 관련 프로그래밍 언어가 지원하는 경우에 이용한다.)

◆ B-3-3. 데이터시트의 다중 행 데이터를 소스로 변환하기

- **B-3-3-1.** 다중 행 데이터들을 소스 코드의 클래스로 구현할 때는, 데이터시트마다 할당되어 있는 클래스의 내부 클래스로써 구현한다.
  : 데이터시트 당 1개의 클래스를 생성하는 방식으로 규칙을 정했기 때문이다.

※ 분명히 내부 클래스의 개념이 없는 프로그래밍 언어도 존재한다. (C언어처럼...) 하지만, 실제 이 프로젝트에서 사용하는 언어들은 모두 내부 클래스를 작성할 수 있도록 되어 있기 때문에, 그렇지 못한 ~~구세대~~ 언어들은 고려 대상에서 일단 제외한다.
 그런 언어에서 대안으로 제시할 방법이 없지는 않지만, 그런 부분까지 명세하기에는 시간도 좀 아깝고, 별로 필요하지도 않아서 그런 것이다. (정말로 필요할 때가 되면 당연히 명세를 그에 맞게 개정할 셈이다.)

- **B-3-3-2.** 이렇게 작성하는 내부 클래스의 이름은, 해당 기준 열의 데이터 타입 키워드의 [] 기호 안에 문자열로 넣어서 지정할 수 있다.

※ 즉, 이제까지는 데이터 타입 키워드에 내부 클래스 이름을 넣지 않고 사용한 셈이다.
 기존에는 다중 행 데이터의 기준 열 타입 선언을 int[], string[], EodGTC[] : key_reference : ??? 식으로 해왔다.
 만일 이를 내부 클래스 이름까지 지정해서 만들고 싶다면 int[ClassName], string[ClassName], EodGTC[ClassName] : key_reference : ??? 같은 식으로 선언해주면 된다는 뜻이다.

- **B-3-3-3.** 데이터 타입 키워드의 [] 기호 안에 내부 클래스로 사용할 이름을 따로 정하지 않을 수도 있다.
  : 이 경우에는, 해당 기준 열에서 설정한 필드 이름을 사용한다. 어쨌든 내부 클래스에 쓸 이름은 필요하기 때문이다.

※ 다만, 이 때 주의해야 할 점이 있다.

기준 열의 필드 이름 역시 소스 코드에서 멤버 변수로 변환해야 하는 대상이기 때문이다.
이 이름을 그대로 내부 클래스의 이름으로 사용해버리면 소스 코드를 프로그래밍 언어로 해석할 때, 문제가 될 수 있다. 같은 이름으로 된 요소들이 여럿 존재하면, 컴파일러가 언어를 해석하는 방식에 따라, 특정 이름이 어느 요소를 가리켜야 하는지 알 수 없어서 컴파일 오류를 내버릴 수도 있기 때문이다.
그러므로, 기준 열의 필드 이름을 가져다 쓰더라도 적절하게 이름이 겹치지 않도록 변형해주는 규칙을 줄 필요가 있다. 이건 크게 대단한 기술이 필요한 부분이 아니다. 그저 '필드 이름 뒤에 Class 문자열을 공통적으로 더해준다' 정도의 규칙만 정해두더라도 이름 충돌은 완벽하게 방지할 수 있다.

- **B-3-3-4.** 다중 행 데이터의 내부 클래스 목록과, 기준 열의 멤버 변수는 각각 별도로 존재해야 한다.
: 기준 열을 설정함으로써 기준 열 데이터가 사라지고, 만들어진 내부 클래스로 대체되지 않는다는 점에 주의해야 한다.
기준 열 데이터는 말 그대로 **다중 행의 데이터들을 조직하기 위한 기준을 제공**할 뿐이다. 기준 열에 있는 각 행들의 데이터들 역시, 데이터시트의 관점에서는 분명하게 멤버 데이터로 들어가야 하는 요소들이다.

Account	CharacterName	JobClass	CharacterID	EquippedItem
Common	Common	Common	Common	Common
string[Character]	string	enum : sbyte : eJobClass	long[EquipItem]	EodGTC : key_reference : Item
god01[0]	상남자	Worrior	324[0]	Weapon_1
god01[0]	상남자	Worrior	324[1]	Weapon_2
god01[1]	원혼	Archer	325[0]	Weapon_10
god01[1]	원혼	Archer	325[1]	Weapon_12
god01[2]	YoureFired	Wizard	326[0]	Weapon_23
god01[2]	YoureFired	Wizard	326[1]	Weapon_24
god01[2]	YoureFired	Wizard	326[2]	Weapon_27

<살짝 변형해서 재탕(?)한 다중 열 데이터 구성>

※ 문서 작성할 때 가로 길이가 모자라서 잘라낸(...) 열 몇 개를 제외하고 이미 한 번 봤던 예제 데이터시트다. (아니, 사실 줄 수도 많이 줄었다. 다중 행 데이터의 단계가 하나 더 줄어들었기 때문이다.)
잘 보면 녹색 바탕으로 된 세 번째 줄, 즉, 데이터 타입을 선언하는 부분에서, 기준 열의 데이터 타입 선언이 조금 바뀌었음을 알 수 있다. (string[Account], long[Character])
[] 안에 들어 있는 문자열들이 바로 다중 행 데이터의 내부 클래스를 만들 때, 그 이름으로 사용할 문자열이다.

string[Account]는 내부 클래스로 class Account { … }를 만들 것이다. 마찬가지로, long[Character]는 내부 클래스로 class Character { … }를 만들 것이다.

하지만 주의해야 할 점은, 여전히 데이터시트의 관점에서 보자면 <Account> 열의 데이터와 <CharacterID> 열의 데이터들이 멤버로써 필요하다는 점이다.
전체 데이터시트 클래스의 관점에서 보자면 위 데이터시트의 관계는 다음의 객체 선언으로 표현할 수 있다.

```
class DatasheetData
{
    class Character
    {
        long characterID;
        eJobClass jobClass;
        list<EquipItem> equipItems;
    }

    /* 데이터시트에서 단 하나의 열만 가지고서 내부 클래스를 만들어야 할 경우에는, 예외적으로 내부 클래스를 생략할 수도 있다.
       내부 클래스를 생략한다면, Characer 클래스의 equipItems는 list<EodGTC> 타입으로 표현되고, 아래 클래스는 생성하지 않는다.*/
    class EquipItem
    {
        EodGTC equippedItem;
    }

    string account;
    list<Account> accounts;
}
```

※ 위 코드 예제에서, EquipItem 클래스와 Character 클래스는 실제 내부적인 개념과는 다르게, EquipItem 클래스가 Character 클래스의 내부 클래스로 구현되지 않았다.
사실 객체 계층이 반드시 클래스 선언의 계층으로 그대로 연결되어야 하는 당위성은 없다. 예제로 제시한 테이블에서의 관계 계층의 단계는 말 그대로 **각자 객체들끼리 어떻게 관계 맺는지에 따라 결정되는 사항**이다. EquipItem이라는 객체 그 자체는 반드시 Character 객체 안에서 선언되어야만 하는 이유는 없는 셈이다. EquipItem이 Character 객체 안에서만 반드시 사용되어야 한다는 법은 없기 때문이다.
그리고, 내부 클래스 단계를 과도하게 두지 않는 편이, 소스 코드가 지나치게 길어지지 않는 장점도 있다.

- **B-3-3-5.** 다중 열 데이터의 내부 클래스를 구성해야 하는 열이 단 한 개라면, 이를 따로 내부 클래스로 만들지 않고 생략할 수 있다.
  : 이를 사용해야 하는 쪽에서는, 내부 클래스의 배열 컨테이너 타입을 선언하는 대신, 해당 열의 데이터 타입 배열 컨테이너로 선언한다.

※ 바로 직전의 예제가 아주 좋은 예시이다.
보면 알겠지만, <CharacterID> 열에서 다중 열 데이터를 선언했으나, 그 이후에 나오는 열은 <EquippedItem> 단 하나일 뿐이다.
이런 경우라면, 굳이 <EquippedItem>의 내용을 별도의 클래스로 만들어야 할 필요가 없다.
어차피 멤버 변수가 하나 뿐이기 때문이다.그래서 이럴 때는 예외적으로 내부 클래스를 생략할 수도 있게 한다.
위 예제에서, 내부 클래스를 생략한다면, Character 클래스의 equipItems 변수는 list<EodGTC> 타입으로 표현하면 된다. EquipItem 클래스 부분은 생성하지 않는다.

C. 기반 시스템

C-1. 하드웨어 & 플랫폼

◆ C-1-1. 지원사항

- **C-1-1-1.** **Windows와 OSX** 운영체제에서 실행할 수 있어야 한다.

- **C-1-1-2.** Linux는 가급적 지원할 수 있으면 좋으나, 필수적이지는 않다.
  : Android 개발은 Windows, OSX, Linux 어느 운영체제에서도 가능하기 때문에, OSX에서만 빌드가 가능한 iOS에 대한 문제만 없다면, 나머지는 상관없다.

※ 기실, OSX를 지원 사항에 꼭 포함시킨 이유는 순전히 iOS가 OSX에서 밖에 빌드할 수 없다는 이유 때문이다.
이것 때문에 개발 컴퓨터들이 OSX일 수도 있기 때문이다. 뭐, 주 개발 엔진인 유니티 3D가 OSX로도 많이 이용하기 때문인 점도 있다.

- **C-1-1-3.** **IBM 호환 PC와 Mac에서의 사용만을 전제**로 제작한다.
  : 특히, 모바일 운영체제에서의 사용에 대해서는 지원하지 않는다. 다만, 제작 과정에서 자연스럽게 모바일 기기를 지원하게 되는 경우(Windows Modern Style GUI 등)는 예외로 한다.

- **C-1-1-4.** 화면 터치, 진동이나 중력 센서, 카메라와 같은 기능은 고려하지 않는다.

◆ C-1-2. 해상도

- **C-1-2-1.** 해상도는 고정해도 되지만, 최소한 가로 1280, 세로 960 픽셀 이상의 해상도를 가져야 한다.

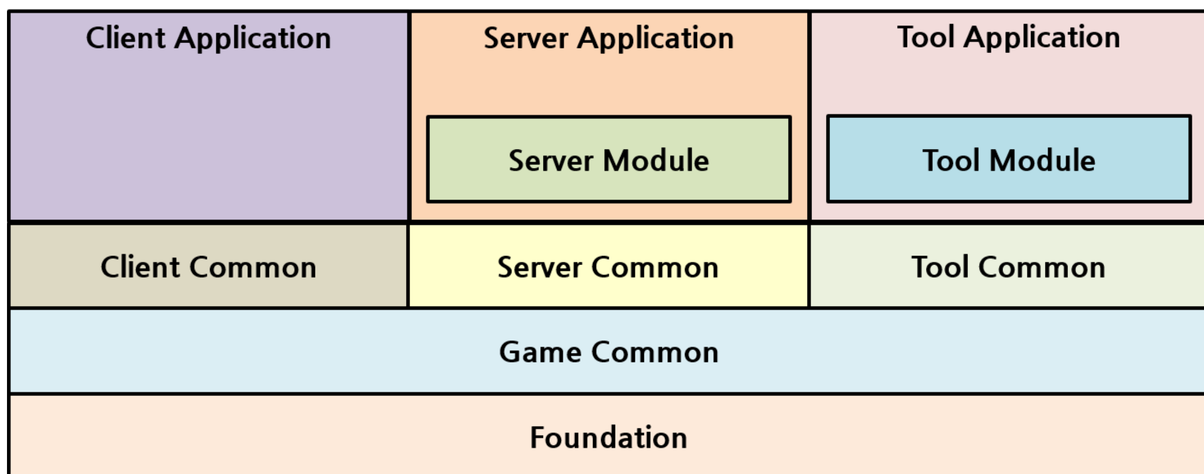
- **C-1-2-2.** 반응형(Responsive)으로 해상도를 조절해야 할 필요는 없다.
  : 해상도를 중간에 변경하는 기능도 고려하지 않는다.

- **C-1-2-3.** 특별히 요구하는 해상도 비율은 없다.
  : 최근 디스플레이들이 주로 가로 16 : 세로 9의 비율로 출시하므로, 잠정적으로 이를 따를 것이다.

C-2. 소프트웨어 구조

◆ C-2-1. 기본 구조

- **C-2-1-1.** 프로젝트 Eternal Guardians의 소프트웨어 구조는 다음과 같다.



<이 프로젝트의 소프트웨어 구조>

- **C-2-1-2.** 게임 데이터 스크립트 생성기 프로젝트는 위 소프트웨어 스택에서 **Tool Module과 Tool Application에 걸치는 프로젝트**다.

◆ C-2-2. 제한 사항

- **C-2-2-1.** 데이터시트를 데이터 스크립트로 변환하는 **핵심 기능들은 모두 라이브러리에 들어갈 수 있어야 한다.**
  : 이는 아키텍처에서 Tool Module 부분에 해당한다.

※ 이와 같은 방식에는 여러 가지 상당한 이점이 있다.
우선, 핵심 기능이 라이브러리 프로젝트로 묶이기 때문에, GUI와 직접 의존성을 가지지 않으므로 설계 구조가 '예쁘게' 나온다.
그리고, 핵심 기능들이 GUI에 의존적이지 않기 때문에, CLI 방식과 GUI 방식을 각각 지원하게 만들기가 쉬워진다. CLI 방식은 이 프로그램을 빌드 자동화 구축의 한 단계로 설정하기에 좋다.
마지막 장점은, GUI 구현 도구가 마음에 들지 않을 경우, 다른 제품 구현으로 교체하기가 쉽다

는 점이다. GUI와 핵심 동작 부분이 분리되어 있으니, GUI 구현이 마음에 안 들면 GUI만 갈아치우면 되기 때문이다. (물론, 실제는 말처럼 쉽지만은 않을 테지만...)

- C-2-2-2. 경로를 표기할 때는 반드시 슬래시('/') 만을 사용한다.
: 역슬래시('₩')는 Windows에서 기본 디렉토리 구분자이지만, 다른 운영체제에서는 그렇지 않다. 모든 운영체제는 공통적으로 '/' 문자열을 디렉토리 구분자로 쓸 수 있으므로, '/'를 표준으로 간주하고 사용한다.

※ .NET Framwork의 기본 클래스 라이브러리(BCS)에는 파일 및 디렉토리 경로를 다루는 구현이 다수 내장되어 있다. 혹시 잘 모르면 System.IO.Path 쪽 함수 인터페이스들을 찾아보기 바란다. 이런 거 괜히 독자적으로 만들 생각하지 말고, 잘 만들어진 기존 도구들을 이용한다. 플랫폼마다 달라지는 미묘한 처리들을 문제 없이 하나의 프로그램으로 구현하는 건 생각보다 쉬운 일이 아니다.

- C-2-2-3. 모든 데이터시트들을 일괄적으로 변환한다.
: 여기에는 다음과 같은 의미를 내포하고 있다.

- 데이터 스크립트 변환기는 지정한 디렉토리에 있는 모든 데이터시트들을 한꺼번에 읽어와야 한다.
- 데이터 스크립트 변환기는 변환 명령이 있을 때, 읽어온 모든 데이터시트들을 순차적으로 처음부터 끝까지 변환해야 한다.

- C-2-2-4. GTC Key를 GTC로 변환하는 기능을 구현해야 한다.
: 이게 가능하기 위한 전제 조건은 데이터 스크립트 변환기 프로그램이 항상 모든 데이터시트의 내용을 읽어와야 한다는 점이다. 그렇지 않다면, GTC Key로써 참조한 GTC를 찾아낼 방법이 없기 때문이다.

- C-2-2-5. GTC를 데이터 스크립트에 저장할 경우에는 GTC Key 문자열 값이 아닌, 본래의 GTC 숫자 값으로 저장해야 한다.
: 데이터시트는 사람의 편의를 위해 GTC 참조를 GTC Key로 대신하도록 지정할 수 있다. 이는 사람의 편의를 위한 기능이다. 하지만, 데이터 스크립트는 프로그램에서 사용해야 하므로 컴퓨터 프로그램에 더 능률적인 타입으로 저장해두어야 한다.

C-3. 사용자 인터페이스

◆ C-3-1. 그래픽 사용자 인터페이스(Graphic User Interface, GUI)

- **C-3-1-1.** 핵심 기능들은 모두 C# 라이브러리로 작성할 것이므로, GUI 도구를 이용해 저작도구 프로그램을 만들 때는, 시간과 비용 대비 가장 적절한 도구를 선택해서 제작한다.
  : 현재로써는 어떤 방식으로든 **C# 언어를 기반**으로 **다양한 플랫폼**에서 동작할 수 있는 도구를 기반으로 작성할 수 있는 방법을 찾고 있다.

※ 여기에는 다음과 같은 방안이 있다.

먼저, 게임 클라이언트 프로젝트와 같이 Unity 3D 엔진으로 게임 데이터 스크립트 생성기를 만드는 방법이다. 이 방법은 현재 프로젝트 참여하는 모든 개발자들이 익숙한 방법이기 때문에 제작에 대한 교육이 불필요한 장점이 있다.
하지만, 유니티 엔진은 본래 게임 엔진이라서, 본래 GUI 기능이 부실하다. 전문적인 GUI 제품에서 제공하는 강력한 GUI 기능들을 쓰기를 어렵다. 그래도 유니티 엔진의 특성상, Windows와 OSX를 지원할 수는 있다.

두 번째는 QT나 wxWidget과 같은 C++ 전문 라이브러리를 이용하는 방법이다.
전문 GUI 제품이기 때문에 매우 수준 높은 툴을 만들 수 있는 장점이 있다. 또한, 위 제품들은 Windows, OSX, Linux 모두 지원 가능하다.
하지만 C++ 기반으로 작성해야 하므로, C# 기반인 게임 프로젝트와 언어 기반이 다르고, 도구에 대한 교육 비용이 들어가는 게 단점이다. (사실 이런 방법을 사용하는 것은 지금으로써는 거의 고려하기 어렵다.)

마지막 방법은 Mono.NET 기반의 GTK#으로 툴의 GUI를 작성하는 방법이다.
Microsoft .NET은 .NET을 가장 자연스럽고 완벽하게 지원하는 플랫폼이지만, '아직은' Windows 외의 다른 플랫폼에서 사용할 수 없다. 따라서 OSX와 Linux에서 사용할 수 있는 .NET 플랫폼은 Mono.NET 밖에 없다. 일단, Mono.NET으로 제작할 수 있는 .NET 기반 GUI 라이브러리는 GTK#이 유일하다.

- **C-3-1-2.** GUI 저작 도구를 제작할 경우, Windows와 OSX 운영체제를 확실히 지원해야 한다.
  : Linux 지원은 필수적이지는 않다.

※ 멀티 플랫폼을 강조하는 이유는 빌드 환경을 덜 복잡하게 구성해야 하기 때문이다.

모바일 운영체제의 양대 산맥은 Android와 iOS다. 이 중에서 iOS는 오직 OSX에서만 빌드 할 수 있다. Android는 Linux 기반 운영체제이기는 하지만, Windows, OSX, Linux 모두 원칙적으로 개발 및 빌드가 가능하다.

이런 상황에서, 저작도구가 평소 흔히 하는 생각대로 오직 Windows만 지원한다면 빌드 자동화를 구성할 때 골치가 아파진다. 빌드는 가급적 하나의 기계에서 전부 처리하게 만드는 게 오류를 줄이기 쉽고, 빌드 구성도 간단하기 때문이다.

실제로, 현재도 프로젝트의 공식 빌드 컴퓨터는 Mac을 상정하고 있으며, 이 때문에 자체 개발 도구들도 가급적이면 OSX는 지원하게 하는 편이 좋다고 본다.

P.S : 이 부분은 단지 모바일 플랫폼 개발에만 국한한 내용이 아니다. 이미 PC 게임들도 Steam 플랫폼의 활성화와 더불어 멀티 플랫폼 지원이 대세가 되어 가고 있다.

- **C-3-1-3.** 해상도는 고정적으로 사용해도 좋다.
: 단, PC 기준으로 충분한 해상도를 갖춰야 한다. 적어도 1280 × 960 이상의 해상도여야 한다.

- **C-3-1-4.** 반응형(Responsive) GUI를 지원하지 않아도 좋다.
: 해상도 변화가 극심한 환경에서 사용할 것이 아니기 때문이다. 특히, 모바일 환경에서는 전혀 사용을 고려하고 있지 않으므로 더욱 그렇다.

◆ C-3-2. 명령행 사용자 인터페이스(Command Line Interface, CLI)

- **C-3-2-1.** 명령행 사용자 인터페이스는 Windows와 OSX를 필수적으로 지원해야 한다.
: Linux는 필수적이지 않다.

※ 저작도구 자체가 Windows와 OSX 지원이 필수이기 때문에 CLI 역시 그 요구사항을 따른다.
저작도구가 Linux에서의 실행도 아무런 문제 없이 지원한다면, CLI도 Linux를 지원하지 못할 이유가 없다.

- **C-3-2-2.** 명령행에서 인식할 **키워드는 대 / 소문자를 구분하지 않게 제작한다.**
: Windows의 기본 명령 도구(cmd.exe)는 대 / 소문자를 구분하지 않기 때문이다. (Windows의 파일 시스템도 대 / 소문자 구분 없이 쓰니까...)