!!! 개발 (D-I-T) 과정 !!!

목적 : 효율적인 개발

1. 나 자신과, 다른 사람과의 협업을 원활히 하기 => 직관적 / 가독성 높은 구조 및 코드
2. 코드 재사용 => 모듈성 / 사용성 효용성 높은 아키텍쳐를 띈 코드 개발
    - 독립적으로 존재하는 모듈.
    - 모듈의 역할 - 역할에 따른 Input / Output 이 직관적으로 명세 되어 있어 사용이 쉬움.

실질적 이행사항

Design (개발)

1. **Product Backlog** (피상적 기능 - End User가 쓰는 기능) 선정
2. **Sprint Backlog** (세세하게 개발해야 하는 기능) 분화 (+ Usecase)
    - 이까지는 개발 전 어느 정도 끝내는 부분.
3. **Modulization** : 중심적인 내용, 큰 내용을 뽑아 분산 및 독립 시키기 (+ Activity + a)
    - Sprint Backlog 의 전체 내용을 다 모듈화 시킬 필요는 없음. **모듈로서 독립시켰을 때 효과적일 기능**들, 단독으로 독립될 수 있는 기능들에 대해 => 분산하여 독립.
    - 개발 맡을 때나 개발 들어갈 때 하면 좋은 부분인 듯.

여기부턴 각 부분에 대한 직접적인 개발

4. **Module 설계** :

- 해당 Module 이 수행할 직관적인 역할에 대해 명세 (Backlog와 별개 취급할 수도 있고, 필요 결과물 및 기획을 많이 참고할 수도 있음.)
- 직관적 역할별 Input / Output / 내부 흐름 정리
- 해당 직관적인 역할에 대한 파악을 바탕으로 Class Diagram 작성. (**실질적 설계**)
- State Diagram, Sequence Diagram 도 필요에 따라 작성.
- 실질적 설계 단계에서 Design Pattern, Architecture 가 고려 / 사용될 수 있음.

- 해당 기록은 프로젝트 문서 내에 쌓아둘 수 있도록 함.
- 현재의 경우 Backlogs & Sprint 의 각 Backlog 내 개발 항목.

Implementation

5. **Module 제작** : Module 설계 기반 제작.
6. **Interface 제작** : Module 을 사용하기 위한 요소, 혹은 Unity 등 엔진에 맞물려, 해당 Module 이 돌아가게 만들어 주는 요소 제작.

- 복잡할 경우, 설계 및 제작된 Module 을 추상화 시킨 모형과, 엔진 등 맞물려 떨어지는 내용이 어떻게 연결되어 있는지 - 연결관계를 간이 Diagram으로 작성하여 표현할 수 있음. (추상화 - 계층적 구조 와도 연결되어 있는 생각.)

개발 과정이 어느 정도 끝나고 난 후 이행

Test

7. 모든 시나리오 (테스트 케이스) 를 기반에 둔 테스트.
8. 이전 작성한 Diagram 등을 결과물에 따라 수정.

개발 과정 종료 후 이행

After Testing

> *성장, 협업 (사용법 공유 및 모듈화 익숙), 자원 (이후에 다시 써먹을 수 있는)을 위한 정리 단계.*

- **성장**을 위한 정리 : 해당 개발을 하면서 얻은 새롭고, 활용성, 효용성이 높은 지식을 Obsidian 에 기록. (이건 중간 중간에 진행하기도 할 것.)
- **협업**을 위한 정리 :
  - Unity 의 경우, 다른 개발자들이 만든 결과물을 손 댈 때 손대는 것은 **Prefab** 일 것. 이에 따른 방안.
    - 해당 Prefab이 수행하는 직관적인 역할(들)을 명세.
    - 각 역할을 수행하기 위해서 Prefab에 뭘 집어넣고(Inspector 상에), 어떻게 동작시켜야 하는지, **사용법**을 기록하여 공유하기.
    - (결국 개발 협업에서 개발 결과물은, 이걸 쓰는 사람이 쓰기 쉽고 - 이해하기 쉽게 만드는 것. 이게 가장 중요한 것 같긴 함.)
  - 제작 모듈에 관한 설명은 다음과 같이 기록할 것 :
    - 구조 : 그간 작성한 Diagram 을 정리하여 Backlogs and Sprint 에 첨부.
    - 코드 : 평소 개발할 때, /// 를 이용한 주석 처리. 직관적인 코드와, 나 스스로도 읽었을 때 읽기 쉬운 코드 구조를 구축하는 것으로 충분치 않을까 함. 마지막엔 다시 읽으며, 미진한 부분 정리.
- **자원**을 위한 정리 :
  - 이건 솔직히 Project 가 끝난 다음 마련해도 됨.
  - 제작 모듈은 별도의 UnityPackage 등으로 빼두기.
    - 이때 "제작 모듈에 관한 설명" (위의 기록 사항) 을 정리하거나 링크 첨부하여 함께 보관해두기.
  - 만약 진짜 API Document 를 만들어야 한다면 "000. Surface / 001. Depth"로 나누기.
    - Surface 의 경우 해당 모듈의 표면.
    - Depth 는 해당 모듈의 내부 구성이나 Customizing 등에 활용할 수 있는 정보.

- 모듈 안에 다른 모듈이 있다면, 내부 소주제 구성 (!! 앞으로의 기록 방침 > 의제 1. 분류) 을 모방하여 기록.

확장판 (더 자세한 내용)

Design (개발)

1. **Product Backlog** (피상적 기능 - End User가 쓰는 기능) 선정
   - Sprint Backlog 작성 시 Product Backlog 기반 A-Z 필요한 기능들을 발상해 볼 것이므로 (이유), **Product Backlog 에 대한 이해도를 높이는 과정**이 필요할 수 있음. 다음은 Product Backlog 의 이해도를 높일 때 사용할 수 있는 방법
     - 스토리 보드 : 유저가 해당 Product Backlog 를 사용할 때 어떤 흐름으로, 어떻게 사용할지에 관한 흐름을 파악 (Activity Diagram을 기반으로 할 수 있음.)
     - Kano Model : 해당 Product Backlog 의 기본 요구사항, 성능 요구사항, 기대 요구사항, 숨겨진 요구사항, 파괴적 요구사항을 분류하여 우선순위를 설정하는 방법
     - 혹은 그냥 Usecase 를 통해 해당 Product Backlog 를 사용하는 User 가 어떤 기능을 더 자세히 사용할 지 파악할 수 있으며, 혹은 필요 기능 - Sprint Backlog 자체를 떠올려도 가능함.
     - 누가 무엇을 이용해서 어떻게 어떤 시점에 왜 사용하는지, 육하원칙을 사용하는 것도 효과적인 방법으로 보임.
2. **Sprint Backlog** (세세하게 개발해야 하는 기능) 분화
   - 개발할 때 뭐가 필요할 지 (**기능**) 대략적이나마 **자세하게** 작성하는 과정
   - 세분화를 담당한다고 생각하면 된다. 즉, 해당 주제를 더욱 세세하게 나누고 단계를 나눔으로써 난이도를 떨어트린다 (격하).
   - 요구 사항에 대한 명세이므로, 이때 부터 In, Out, 기조 확립, 모듈화 등에 대해 자세히 생각할 필요 없음. 오히려 그러면 기능이나 요구사항에 대해 자세히 생각하지 못하게 되므로 답답해짐. **기능, 필요한 게 뭐가 있을지, 등등 정도나 생각.**
   - 자세하게 작성 (분화) 하는 방법
     - Product Backlog 기반 A - Z 까지 필요한 기능을 다 생각해본다는 느낌으로 작성해보기.
     - Usecase (기능 목록) / Activity Diagram (흐름) 을 사용하는 것이 용이할 수 있음.

(생략될 수 있음.
3. Sprint Backlog 를 기반으로 필요한 객체 및 그들 사이의 연결관계에 대해 떠올리기
- 이전 작성해보곤 했던, 추상화된 (필드 / 메서드가 없는) 클래스 다이어그램

4. 중심 격에 해당하는 내용 파악
)

5. **Modulization** : 중심적인 내용, 큰 내용을 뽑아 분산 및 독립 시키기
    - Sprint Backlog 의 전체 내용을 다 모듈화 시킬 필요는 없음. 모듈로서 독립시켰을 때 효과적일 기능들, 단독으로 독립될 수 있는 기능들에 대해 분산하여 독립
    - 분산 및 독립 시에는 연결관계에 대한 고려를 끊어내고, 오히려 직관적 역할이 어떻게 구분되느냐에 신경쓸 수 있도록 한다. (이후, 해당 역할을 명세할 것이고, 연결관계가 해당 모듈의 역할 내의 In / Out 으로 대체될 것.)
6. **Module 설계** :
    - 해당 Module 이 수행할 직관적인 역할에 대해 명세 (Backlog와 별개 취급하는 게 용이)
    - 직관적 역할별 Input / Output / 내부 흐름 정리
    - 해당 직관적인 역할에 대한 파악을 바탕으로 Class Diagram 작성. (**실질적 설계**)
        - State Diagram, Sequence Diagram등 추가 OOAD도 필요에 따라 작성.
    - 실질적 설계 단계에서 Design Pattern, Architecture 가 고려 / 사용될 수 있음.

Implementation

7. **Module 제작** : Module 설계 기반 제작.
8. **Interface 제작** : Module 을 사용하기 위한 요소, 혹은 Unity 등 엔진에 맞물려, 해당 Module 이 돌아가게 만들어 주는 요소 제작.
    - 복잡할 경우, 설계 및 제작된 Module 을 추상화 시킨 모형과, 엔진 등 맞물려 떨어지는 내용이 어떻게 연결되어 있는지 - 연결관계를 간이 Diagram으로 작성하여 표현할 수 있음. (추상화 - 계층적 구조 와도 연결되어 있는 생각.)

- Implementation 중, *파악해두었던 구조가 바뀌게 되는 경우*도 분명히! 있을 수 있다.
  모듈의 **구조가 전체적으로** 바뀌거나, **직관적인 역할 - Input / Output - 이 직접적 맞닿은 부분**이 바뀌지 않는 한, 큰 상관은 없으므로, 쭉 진행한다. (바뀐 내용이나 구조에 대해선, Design 에 반영할 것.)

- 실질적인 코드 작성에 있어서 Copilot X 를 활용한 AI 프로그래밍 (추상 및 전체) 의 내용을 응용 및 참고할 것.

Test

9. 모든 시나리오 (테스트 케이스) 를 기반에 둔 테스트.