**소프트웨어공학**

**프로젝트 보고서**

**- Title : Yoot Game Project Document -**

|  |  |
| --- | --- |
| **Class** | **소프트웨어공학, Class 01** |
| **Professor** | **이찬근** |
| **Team number** | **Team 18** |
| **Team Members** | **김찬중 (20213780)**  **박민용 (20213113)**  **엄태형 (20202029)**  **이정우 (20202021)**  **태아카 (20234483)** |
| **Author** | **김찬중** |

**목차**

1. Introduction - 02 -

**A. 프로젝트 개요**. - 02 -

2. Body……… - 12 -

**B. 유스 케이스 모델**. - 12 -

1. Use Case Diagram - 12 -

2. Use Case Text - 13 -

**C. 기타 요구 사항** - 17 –

1. Domain Model - 17 -

2. Activity Diagram - 18 –

3. System Sequence Diagram - 22 –

4. Operation Contract - 23 -

**D. 설계 및 구현** - 27 –

1. 설계 (Class Diagram, Sequence Diagram, MVC, 의존성 그래프) - 27 –

2. 구현 (프로그램 코드 및 코드의 동작, UI 코드 변경점) - 38 -

**E. Junit Test** - 62—

**F. Github Report (Git branch 전략 및 Github contribution)** - 74--

**3. Conclusion…………………………………………………………………….…………….…………- 84 -**

G. 회고 및 기타. - 84 -

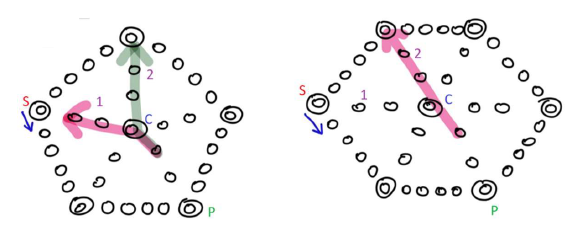
1. Introduction

A) 윷놀이 프로젝트의 개요

**a – 1 - 1) 윷놀이 규칙 (말, 참여자, 윷놀이 이동)**

* 윷놀이에 참여하는 인원은 최소 2명, 최대 4명이다.
* 윷놀이에 배치되는 참여 인원 당 배정되는 말의 개수는 최소 2개, 최대 5개이다.
* 윷놀이에 사용되는 이동 규칙은 [백도, 도, 개, 걸, 윷, 모]가 있고 각각 [-1, 1, 2, 3, 4, 5] 칸을 이동해야 한다.
* 윷을 던졌을 때 ‘윷’ 혹은 ‘모’가 나온 경우, 말을 이동하기 전에 다시 던져 추가로 이동할 거리를 저장해야 한다.
* 말을 이동했을 때, 그 위치에 아군의 말이 위치하고 있으면 말을 엎어야 한다.
* 말을 이동했을 때, 그 위치에 적군의 말이 위치하고 있으면 말을 잡고 해당 말의 소유자는 다시 윷을 던질 권리를 가지게 된다.
* 모든 말을 골인시키면 게임에서 승리한다.
* 승리 후 게임을 다시 시작하거나 종료할 수 있다.

**a – 1 - 2) 윷놀이 규칙 (판 위에서의 규칙)**

* 각 판의 꼭짓점에서 말이 움직이기 시작한다면 가운데로 이동해야 한다.
* 중앙의 이중 원 위에 위치하지 않고 지나칠 경우 말은 2번째 최단 경로에 해당하는 곳으로 이동해야 한다.
* 

a – 2 - 1) 게임 초기화면의 구상

텍스트, 스크린샷, 도표, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 초기 화면에선 사용자 수, 말의 개수, 게임판의 모양을 받을 수 있게 위와 같이 UI를 구상하였다.
* 사용자 수, 말 개수, 게임 판을 중복해서 선택할 수 없게 Radio Button으로 구현하는 것이 좋을 것으로 예상된다.
* 사용자 수, 말 개수, 게임 판에서 선택되지 않은 버튼이 있을 것을 대비하여 Default 값을 따로 지정하는 것이 좋을 것 같다.
* 각 옵션을 선택 후 ‘게임 시작’ 버튼을 누르면 게임 화면으로 넘어갈 수 있게 설계하면 좋을 것 같다.

a – 2 - 2) 게임 종료화면의 구상

텍스트, 폰트, 직사각형, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 위의 그림과 같이 특정 게임 참여자가 승리하였을 시 재시작과 종료 둘 중하나를 고를 수 있게 버튼을 배치하면 좋을 것 같음.
* 재시작 버튼을 누르게 된다면 B-1-1)의 초기 화면으로 다시 돌아가 게임을 다시 설정할 수 있게 해야 함.
* 재시작 버튼을 누르게 된다면 현재 동적으로 할당되어 있는 모든 값을 free하고 다시 시작해야 함. Java의 Garbage Collection의 경우 전역으로 선언되어 있는 class의 값을 수거하지 않는 경우가 있으므로 이를 고려하지 않을 시 재시작을 할 때마다 메인 메모리의 공간을 조금씩 갉아먹게 된다.
* 종료 버튼을 클릭하게 된다면 프로그램 창을 완전히 종료해야 한다.
* (게임 화면에 대해 설명할 내용이 상대적으로 많기에 다음 페이지로 넘어가서 설명한다.)

a – 2 - 3) 게임화면 구상

텍스트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 위의 그림은 B-1-1)에서 사용자가 게임의 환경을 설정하고 ‘게임 시작’ 버튼 눌렀을 때 나오는 화면이다.
* 게임 화면의 좌측 상단에는 사각형, 오각형, 육각형 형태의 판이 출력 된다.
* 판 위의 원은 버튼으로 매핑하여 색깔과 숫자를 통해 참여자와 말의 엎은 개수를 표현한다.
* 우측 상단의 ‘P1 차례’는 해당 참여자의 순서를 나타낸다. 위의 사진에서의 예시는 참여자가 4명이므로 P4까지 출력될 것이다.
* 바로 하단에 위치한 거대한 사각형 박스에는 각 참여자의 남은 말의 개수를 표현하는 상자이다. 말이 출발에서 나와 다시 출발 지를 지나 나가게 된다면 남은 말의 개수가 하나씩 줄어들 것이다.
* [백도, 도, 개, 걸, 윷, 모]가 mapping 되어 있는 상자는 debug 용 이동 규칙 버튼이다. ‘도’를 누르면 사용자의 말이 한 칸 움직이게 된다.
* 그 아래에 있는 ‘랜덤’ 버튼은 우리가 윷을 던지는 것처럼 난수를 이용해 가변적인 이동 규칙을 적용하는 버튼이다. 해당 출력 결과는 ‘윷 결과’ 상자에 나타날 것이다.
* 우측 하단에 있는 ‘내보내기’ 버튼은 윷을 던지고 난 후 새로운 윷을 내보내기 위한 버튼이다. ‘윷’이 나오고 ‘내보내기’ 버튼을 누르게 된다면 4번째 칸에 해당 참여자의 색깔과 숫자 1이 표시될 것이다. (아래의 붉은 원과 숫자 1이 그 예시이다.)
* 원이(가) 표시된 사진

  AI가 생성한 콘텐츠는 부정확할 수 있습니다.
* ‘판에서 선택’ 버튼은 판에 위치한 말을 클릭해서 해당 말에 참여자가 선택한 이동 규칙을 적용시키는 버튼이다.
* 우측 하단의 ‘윷 스택’은 이동 규칙을 저장하는 리스트라고 생각하면 된다. ‘걸’이 나왔다고 가정하면 해당 상자에 ‘걸’이라는 이름의 버튼이 나오게 되고 참여자는 ‘걸’ 버튼을 누르고 ‘내보내기’ 혹은 ‘판에서 선택’ 버튼을 눌러 다음 행동을 선택하게 된다. ‘윷’, ‘모’가 나왔을 경우 참여자는 ‘윷’, ‘모’가 아닐 때까지 윷을 던지는 것이 일반적이다. 이를 위해 연속적으로 윷을 던진 경우 각각의 이동 규칙에 해당하는 버튼을 생성하게 된다.

a - 3) 게임 진행 과정



* 위의 게임은 ‘유희왕’의 게임 진행 과정이다. 각 플레이어는 턴을 기다리며 자신의 차례를 기다리게 된다. 카드를 드로우하고 자신의 카드를 이용해 게임을 전개한다. 자신의 전개가 끝나면 자신의 턴을 종료하고 상대의 턴으로 넘기게 된다.
* 위와 같은 카드 게임의 프로세스를 생각했을 때, 윷놀이도 비슷하다고 생각하였다.

텍스트, 폰트, 도표, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 윷놀이도 ‘던지기’의 과정을 통해 게임을 전개하게 되고 상대의 말 혹은 자신의 말과의 상호작용을 통해 새로운 이벤트를 전개할 수 있게 된다.
* 우선 게임의 참여자는 윷을 던질 수 있는 권리를 가진다. 윷을 던지면 그 결과에 따라 자신의 말을 새로 보내거나 아니면 게임 판 위의 말을 옮길 수 있게 된다. 던져서 나온 결과가 ‘윷’ 혹은 ‘모’인 경우 던질 수 있는 권리를 1회 추가로 받게 된다. 최종적으로 던질 수 있는 권리를 소모하게 되면 참여자는 자신의 말을 control할수 있는 권리를 가지게 된다.
* 게임의 참여자가 새로운 말을 control하게 되는 경우, 상대의 말을 잡는 경우와 그렇지 않은 경우를 생각할 수 있다. 상대의 말을 잡은 경우 다시 던질 수 있는 권리를 가지게 되므로 던지기 Phase로 회귀하게 된다. 그렇지 않은 경우 자신의 턴을 종료하고 다음 참여자의 던지기 Phase로 넘어가게 된다.
* 게임의 참여자가 게임 판 위의 기존 말을 control하게 되는 경우, 마찬가지로 상대의 말을 잡는 경우와 그렇지 않은 경우를 생각할 수 있다. 상대의 말을 잡은 경우 다시 던질 수 있는 권리를 가지게 되므로 던지기 Phase로 회귀하게 된다. 그렇지 않은 경우 자신의 턴을 종료하고 다음 참여자의 던지기 Phase로 넘어가게 된다. 그런데 여기서 하나의 경우를 더 생각해야 한다. 기존의 말을 움직이는 경우 참여자가 모든 말을 내보낼 수 있는 경우의 수가 생기게 된다. 참여자가 자신의 말을 모두 내보내고 모두 골 인 시킨 경우 게임을 종료할 수 있게 된다.

a - 4) 게임 로직 구상

스크린샷, 텍스트, 원, 디자인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 사각형 게임판을 기준으로 로직을 생각한다.
* 참여자가 2명일 때 한 명을 빨간색으로 나머지 한 명을 파란색으로 표시할 수 있다.
* Grouping의 경우 말이 몇 개가 한 칸에 있는 지를 표현하는 것이므로 숫자로 표현하면 로직을 단순화할 수 있다.

텍스트, 스크린샷, 원, 디자인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 빨간색의 말이 위의 상황에서 ‘개’가 나와 2칸을 움직이게 된다면 현재 자신의 좌표의 버튼 색을 흰색으로 바꾸고 숫자를 0으로 초기화한 후 움직일 좌표의 색을 빨간색, 숫자를 1로 바꾸어 주면 된다. 이 경우 파란색 참여자는 자신이 내보낼 수 있는 말의 개수를 2개 더해주어야 한다.
* 위의 사진에서 파란색 말의 위치에 빨간색 말이 위치한 경우, 현재 자신의 좌표의 버튼 색을 흰색으로 바꾸고 숫자를 0으로 초기화한 후 움직일 좌표의 색은 동일하게 유지하되 숫자는 [해당 위치의 숫자 + 움직일 말의 숫자]로 갱신해주면 된다.

스크린샷, 원, 패턴이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 사각형 형태의 판에서 가운데로의 진입은 각 꼭짓점에 말이 위치했을 경우만 가능하다.

텍스트, 스크린샷, 라인, 패턴이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 이러한 움직임을 표현하기 위해 겉의 경로를 이차원 배열의 [0]번째 인덱스로 매핑하면 된다. 가운데로 진입하게 되는 점의 좌표는 point[0][5], point[0][10]일 때이다. (이는 실제 좌표가 아닌 논리적 좌표이다.)

텍스트, 라인, 그래프, 도표이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 가운데로 향하는 좌표의 경우 각각 [1], [2]에 인덱싱하면 된다. 추가적으로 파란색 선의 경우 좌표가 point[1][3]일 때 point[2][3]으로 바꾸어 주는 절차를 진행해주어야 한다.
* 종점 판단은 크게 두 가지로 나눌 수 있다. 빨간색 선을 통해 나가는 경우와 초록색 선을 통해 나가는 경우. 전자의 경우 논리적 좌표가 point[0][20]을 넘길 때, 후자의 경우 논리적 좌표가 point[2][6]을 넘길 때 종점을 지났다고 판단할 수 있다.
* 오각형과 육각형도 이러한 형태의 논리적 좌표 매칭을 이용하면 된다.

a - 5) 개발 환경

* **Intelij IDE**를 이용해 **JDK 21**을 사용해 프로그래밍을 진행한다.
* 코드는 YootProject package에 작성한다.
* 이후 진행될 Junit test는 src directory가 아닌 project 하위에 test directory를 새로 만들어서 진행한다.
* Img 파일을 import할 때에는 상대적 주소를 이용해 가져온다. 절대 주소를 이용하면 다른 환경에서의 실행이 문제가 된다.

B) Use Case Model

**b - 1) Use Case Diagram**

텍스트, 도표, 스크린샷, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* Player는 모든 게임 내의 use case의 접근 권한을 가지고 있다.
* 초기 화면인 Game Setting은 그 값을 인자로 Game 화면을 구성하기에 Main Game은 Game Setting을 볼 수 있다.
* Main Game에서 승리자가 나오면 End Page로 넘어가므로 Main Game은 End Page의 내용을 알고 있어야 한다.
* End Page에선 Restart를 할지 혹은 End를 할지를 선택해야 한다.
* End Page에서 Restart를 하면 Game Setting부터 다시 진행하므로 End Page는 Game Setting의 내용을 알아야 한다.

b – 2 - 1) Use Case #1: Game Setting

텍스트, 스크린샷, 도표, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

|  |  |
| --- | --- |
| **Usecase** | **윷놀이 게임 설정** |
| **개 요** | **윷놀이를 하는 참여자 수, 말 개수, 게임 판을 설정해야 함.** |
| **Primary Actor** | **Player** |
| **Precondition** | **윷놀이 프로그램이 실행된 상태이다. (Intelij를 실행 후 run한 상태)** |
| **Scenario** | **1. 참여자 수를 정한다.**  **2. 사용할 말의 개수를 정한다.**  **3. 게임판의 모양을 정한다.** |
| **Postcondition** | **플레이어 수와 말의 개수, 그리고 게임판의 모양이 결정된다.** |
| **Extra Requirement** | * **플레이어 수는 최소 2명, 최대 4명이다.** * **사용할 말의 개수는 최소 2개, 최대 5개다.** * **판의 모양은 사각형, 오각형, 육각형으로 세 가지 종류이다.** |

b – 2 - 2) Use Case #2: Main Game

텍스트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

|  |  |
| --- | --- |
| **Usecase** | **윷놀이 게임** |
| **개 요** | **윷놀이를 게임 화면으로 참여자가 플레이하는 환경이다.** |
| **Primary Actor** | **Player(s) (2명 이상이므로 복수)** |
| **Precondition** | **사용자가 게임 환경을 설정하고 관련 정보를 토대로 게임판이 설정됨.** |
| **Scenario** | **1. 유저가 윷을 던져서 결과가 나오게 된다.**  **2-1. 유저가 던진 윷이 ‘도’, ‘개’ ‘걸’인 경우 진출한 말 혹은 새로운 말 중에서 나아갈 말을 선택할 수 있다.**  **2-2. 유저가 던진 윷이 ‘윷’, ‘모’인 경우 다시 던질 준비를 한다. ‘1.’의 과정으로 다시 돌아가고 2-1로 분기할 때까지 계속해서 반복한다. 이때 반복하는 동안의 던진 결과는 저장소에 보관되며 우측 화면의 UI로 표기가 된다.**  **3-1. 말을 움직일 때 움직일 곳에 말이 없다면 기존의 위치의 칸의 색깔을 다시 흰색으로 바꾼다. 도착할 위치의 경우 현재 플레이어의 색깔 및 말의 개수를 같이 표기한다.**  **3-2. 말을 움직일 곳에 자신의 말이 존재하는 경우 기존의 위치의 칸의 색깔을 다시 흰색으로 바꾼다. 도착할 위치의 버튼의 숫자를 기존의 위치하던 숫자 + 추가될 숫자를 적용해서 다시 화면에 표현한다.**  **3-3 말을 움직일 곳에 상대 말이 있는 경우 기존의 위치의 칸의 색깔을 다시 흰색으로 바꾼다. 도착할 위치의 버튼의 색깔을 현재의 플레이어의 색깔로 바꾸고 숫자도 마찬가지로 바꾼다. 다른 사람의 말은 다시 회수되며 우측 상단에 잡힌 말의 플레이어에 대한 정보를 다시 갱신한다. 잡은 사람은 다시 1.의 과정으로 돌아가 윷을 던지게 된다.**  **3-4. 말을 움직일 곳이 원점 혹은 그 이후를 지나는 경우 기존에 위치하던 버튼의 색깔을 다시 흰색으로 바꾸어 주고 현재 플레이어의 말의 개수를 조정해준다. 만약 조정한 말의 남은 개수가 0개라면 5.의 과정으로 넘어간다.**  **4. 위의 모든 과정을 반복하고 아무런 추가적인 이벤트가 없다면 현재 플레이어의 순서를 종료하고 다음 플레이어로 턴을 넘긴다. (다른 참여자가 1.의 과정으로 넘어가게 된다. 만약 승리자가 나오면 5.로 넘어감)**  **5. 승자가 나왔으면 엔딩 화면으로 분기한다.** |
| **Postcondition** | **한 번의 프로세스가 끝나고 다른 참여자로 턴이 넘어가거나 게임이 종료된다.** |
| **Extra Requirement** | * **윷을 던져서 나온 값이 ‘윷’, ‘모’인 경우 다시 계속해서 던질 수 있다.** * **말을 움직여서 상대방의 말을 잡으면 잡은 말의 사용자는 다시 한번 윷을 던질 수 있다.** * **말을 움직일 때 움직이는 규칙을 준수해야 한다.** * **말의 움직이는 규칙은 사각형일 때와 오, 육각형일 때의 차이가 있다.** * **말이 판의 각 꼭짓점에 위치할 때에만 가운데의 경로로 이동할 수 있다.** |

b – 2 - 3) Use Case #3: End Page

텍스트, 직사각형, 폰트, 화이트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

|  |  |
| --- | --- |
| Usecase | 윷놀이 게임 종료 화면 |
| 개 요 | 윷놀이 참여자는 게임을 재시작 혹은 종료해야 함. |
| Primary Actor | Player |
| Precondition | 참여자 한 명이 승리 조건을 만족하여 승리자가 되었음. |
| Scenario | 1 - 1. 재시작 버튼을 누른다.  1 - 2. 종료 버튼을 누른다.  2 – 1. Usecase#1로 분기한다.  2 – 2. 게임이 종료된다. |
| Postcondition | 재시작 시 게임을 처음부터 다시 세팅하게 되고 종료 시 창이 닫힌다. |
| Extra Requirement | * **유저가 응답을 할 때까지 프로그램은 대기해야 한다.** |

C) 기타 요구사항

c – 1) Domain Model

텍스트, 도표, 라인, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 윷놀이의 구성요소를 생각하면 위와 같을 것이다. 참여자는 2명에서 4명사이로 제한되고 각각의 참여자는 자신만의 말을 2개에서 5개까지 할당 받는다. 말은 던져진 윷의 움직임 규칙에 의해 움직이게 되고 움직인 말은 판 위에 놓이게 된다.
* Player의 경우 여러 인원이 있기 때문에 각 Player을 구분하기 위한 ID, 각 player의 남은 윷 개수, 점수 정도가 필요한 요소로 보았다.
* Piece의 경우 각각의 말이 판 위의 어느 곳에 위치할 지 각각 저장하는 것이 좋을 것으로 예상하여 Piece 내에는 Position이라는 attribute를 할당하였다.

c – 2 - 1) Activity Diagram (전체)

텍스트, 영수증이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 위의 그림은 윷놀이에서 진행되는 process에 대한 Activity Diagram이다. 각 부분을 나누어서 살펴보자.

c – 2 – 2) Activity Diagram (게임 세팅)

텍스트, 스크린샷, 폰트, 도표이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 게임을 실행하면 프로그램 user는 자신이 원하는 게임 설정을 조정하게 된다.
* 조정하게 되는 것은 총 세 종류로 참여자 수, 말 개수, 보드 형태를 정하게 된다.
* 모든 세팅을 조율한 후 게임 시작 버튼을 누르면 user가 선택한 버튼의 입력 값을 인자로 받아 게임 규칙, player 목록, 보드 및 컨트롤러를 생성하게 된다.

c – 2 – 3) Activity Diagram (윷 던지기, 말 움직임)

텍스트, 스크린샷, 폰트, 도표이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.텍스트, 도표, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 게임에 진입하게 되면 어떤 플레이어의 차례인지 먼저 인식하게 된다.
* 윷을 던지고 그 결과를 임시 저장소에 저장한다.
* 그 다음에 저장된 값이 ‘윷’, ‘모’인지를 검사하게 되고 그럴 경우 추가로 플레이어는 윷을 던지게 된다.
* 말을 움직이게 된다면 플레이어는 말을 내보내거나 판에 위치한 말을 움직이게 된다. 따라서 위와 같이 조건 분기를 넣어주었다.
* 말을 움직일 때, 움직일 곳에 상대방의 말이 있다면 말을 잡고 상대방의 대기 말을 증가시키며 다시 윷을 던질 수 있는 기회를 가지게 된다. 그렇지 않은 경우 다음 프로세스로 넘어가게 된다.
* 말을 움직일 때, 움직일 곳에 자신의 말이 있다면 말을 grouping해주고 포인트를 누적시켜 말이 업혔다는 것을 알리게 된다.

c – 2 – 4) Activity Diagram (승리 조건 및 사후 처리)

텍스트, 도표, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 게임 승리 조건을 만족하게 된다며 End Page를 실행하게 된다. End Page에는 게임을 재시작 혹은 종료할 수 있는 선택을 가지게 된다.
* 게임 승리 조건을 만족하지 않는다면 player의 turn을 상대방에게 넘기고 대기하게 된다.

c – 3) System Sequence Diagram

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* SSD는 Actor와 System 사이의 상호 작용을 표현한 Diagram으로 System의 모든 class를 세부적으로 표현하진 않았다.
* Player는 게임 종료 버튼을 누를 때까지 게임을 진행하게 된다. 게임 시작 시 Player는 게임 세팅 값을 선택하여 System에 create 명령과 함께 인자로 보내게 된다.
* 이후 승자가 나올 때까지 윷놀이 게임이 진행된다. Player의 모든 입력은 버튼을 매개로 System에 전달된다. 게임의 진행 순서는 윷 던지기 -> 말 선택 -> 움직이기 -> 턴 넘기기로 볼 수 있다.
* 승리 조건이 true가 된다면 loop을 빠져나오게 되지만 종료 버튼을 누르지 않으면 또다시 loop에 걸려 처음부터 게임을 다시 세팅하게 된다.

c – 4 - 1) Operation Contract (주요 logic method)

|  |  |
| --- | --- |
| Method 이름 | Player.createPiece() |
| Operation | 말 하나를 시작 위치에 생성한다. |
| Precondition | standbyPieces > 0 (대기 중인 말이 있어야 한다) |
| Postcondtion | - pieces.size()가 1 증가한다.  - standbyPieces가 1 감소한다.  - 새 말이 route = startRoute, position = startPosition에 생성된다. |
| Cross Reference | Use Case: "Main Game" |

|  |  |
| --- | --- |
| Method 이름 | Player.movePieceAt(int route, int pos, int moveCount) |
| Operation | 지정된 위치에 있는 말을 moveCount 만큼 이동시킨다. |
| Precondition | 해당 위치 (route, pos)에 말이 존재해야 한다. |
| Postcondtion | - 해당 말의 positionIndex가 +moveCount 만큼 변경된다.  - 경로 변경이 발생할 수 있다 (if routeChange exists).  - 백도인 경우 positionIndex가 음수가 되면 adjustForBackdo() 처리된다. |
| Cross Reference | Use Case: "Main Game" |

|  |  |
| --- | --- |
| Method 이름 | Player.checkAndMergeStack() |
| Operation | 동일 위치에 있는 자신의 말을 병합한다. |
| Precondition | 동일한 위치 (routeIndex, positionIndex)에 2개 이상의 말이 존재해야 한다. |
| Postcondtion | - 하나의 말로 병합되고 병합된 말의 point가 누적된다.  - 병합된 다른 말은 pieces 리스트에서 제거된다. |
| Cross Reference | Use Case: "Main Game" |

|  |  |
| --- | --- |
| Method 이름 | Player.captureOpponentPiece(int x, int y) |
| Operation | 지정된 위치의 상대방 말을 잡는다. |
| Precondition | (x, y) 위치에 상대방의 말이 있어야 한다. |
| Postcondtion | 해당 말은 제거되고, 그 말의 point 수만큼 상대의 standbyPieces가 증가한다 |
| Cross Reference | Use Case: "Main Game" |

|  |  |
| --- | --- |
| Method 이름 | Player.checkAndHandleArrival() |
| Operation | 도착 지점에 도달한 말을 처리한다. |
| Precondition | 적어도 하나의 말이 hasArrived() 조건을 만족해야 한다. |
| Postcondtion | 해당 말은 제거되고, 그 말의 point만큼 score가 증가한다. |
| Cross Reference | Use Case: "Main Game" |

|  |  |
| --- | --- |
| Method 이름 | PlayGame.didSomeoneWin() |
| Operation | 현재 플레이어가 승리했는지 확인한다. |
| Precondition | PlayConfig.getPieceNum()으로 설정된 말의 수가 기준이 된다. |
| Postcondtion | 현재 플레이어의 score >= pieceNum이면 true 반환, 아니면 false 반환 |
| Cross Reference | Use Case: "Main Game" |

|  |  |
| --- | --- |
| Method 이름 | Piece.move(int step) |
| Operation | 말을 step 만큼 이동시킨다. |
| Precondition | step은 정수여야 한다. (음수 가능; 백도: -1) |
| Postcondtion | - positionIndex가 +step 된다.  - getRouteChange() 결과에 따라 route가 변경될 수 있다.  - positionIndex < 0이면 adjustForBackdo()에 따라 조정된다. |
| Cross Reference | Use Case: "Main Game" |

|  |  |
| --- | --- |
| Method 이름 | PlayGame.phase1ThrowYut() |
| Operation | 현재 플레이어가 윷을 던진다. |
| Precondition | controlPhase == 1 |
| Postcondtion | - lastThrowResult에 윷 결과가 저장된다.  - 윷 또는 모인 경우 extraTurnList에 값 추가 후 다시 phase1.  - 도/개/걸/백도면 phase2로 넘어간다. |
| Cross Reference | Use Case: "Main Game" |

|  |  |
| --- | --- |
| Method 이름 | Piece.move(int step) |
| Operation | 말을 step 만큼 이동시킨다. |
| Precondition | PlayConfig.getPieceNum()으로 설정된 말의 수가 기준이 된다. |
| Postcondtion | 현재 플레이어의 score >= pieceNum이면 true 반환, 아니면 false 반환 |
| Cross Reference | Use Case: "Main Game" |

c – 4 - 2) Operation Contract (주요 UI Interaction 처리)

|  |  |
| --- | --- |
| Method 이름 | 게임 시작 버튼 클릭 (FirstPage → PlayGame) |
| Operation | 사용자 클릭 "게임 시작" 버튼 |
| Precondition | PlayConfig에 플레이어 수, 말 개수, 보드 형태가 적절히 설정되어 있어야 한다. |
| Postcondtion | - PlayGame 인스턴스가 생성된다.  - 내부적으로 Player, YootBoard, YootBoardController가 생성된다.  - 게임이 시작되고 첫 플레이어 차례로 초기화된다. |
| Cross Reference | Use Case: "Game Setting", Sequence Diagram: “click 게임 시작” |

|  |  |
| --- | --- |
| Method 이름 | "랜덤" 버튼 클릭 (YootBoard → PlayGame.actionPerformed("ROLL\_YUT")) |
| Operation | 사용자 클릭 "랜덤" 버튼 |
| Precondition | 현재 턴의 플레이어가 존재하며, controlPhase == 1 |
| Postcondtion | - Yoot.throwing()을 통해 윷 결과가 결정된다.  - 결과가 화면에 출력되고, 스택에 추가된다.  - 윷/모가 아니면 controlPhase가 2로 변경된다. |
| Cross Reference | Use Case: "Main Game", Sequence Diagram: "ROLL\_YUT" → throwing() → updateYootStack()” |

|  |  |
| --- | --- |
| Method 이름 | "내보내기" 버튼 클릭 (YootBoard → PlayGame.actionPerformed("NEW\_PIECE")) |
| Operation | 사용자 클릭 "내보내기" 버튼 |
| Precondition | - 선택된 스택 값 (selectedStackValue)이 존재해야 함  - 해당 플레이어가 대기 말(standbyPiece)을 보유하고 있어야 함 |
| Postcondtion | - 새로운 말이 생성되고 윷 결과만큼 이동  - 같은 위치에 내 말이 있으면 병합  - 상대방 말이 있으면 잡기 시도  - 스택 값 사용 후 삭제 |
| Cross Reference | Use Case: "Main Game", Sequence Diagram: "NEW\_PIECE" → createPiece() → movePieceAt() |

|  |  |
| --- | --- |
| Method 이름 | "판에서 선택" 버튼 클릭 (YootBoard → PlayGame.actionPerformed("SELECT\_ON\_BOARD")) |
| Operation | 사용자 클릭 "판에서 선택" 버튼 |
| Precondition | 스택에서 이동 값이 선택되어 있어야 함 (selectedStackValue != null) |
| Postcondtion | 다음에 사용자가 판 위의 버튼(자신의 말)을 클릭할 수 있게 상태 진입 (waitingForPieceSelection = true) |
| Cross Reference | Use Case: "Main Game", Sequence Diagram: "SELECT\_ON\_BOARD" → 상태 대기 → "MOVE\_" |

|  |  |
| --- | --- |
| Method 이름 | 판 위의 말 버튼 클릭 (MOVE\_) |
| Operation | 사용자 클릭 판 위에 있는 자신의 말 버튼 |
| Precondition | - waitingForPieceSelection == true  - 해당 말이 현재 플레이어의 소유일 것 |
| Postcondtion | - 선택된 말이 selectedStackValue만큼 이동  - 병합/잡기/도착 판정 로직 수행  - 이동 후 차례 전환 또는 추가 턴 여부 판단 |
| Cross Reference | Use Case: "Main Game" |

D) 설계 및 구현

**D – 1) 프로젝트 설계**

**D – 1 - 1) Class Diagram**

* Class Diagram의 크기가 1 page를 모두 차지하기에 다음 page에 표현하였음을 유의할 것.
* Class Diagram를 각 파트 별로 나누어 설명 (파트를 나누는 기준은 의존성과 같은 논리적 구조로 나누어 설명하는 것이 아닌 물리적으로 나누어 설명; 시각적으로 4등분을 하여 설명)

텍스트, 도표, 평면도, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 도표, 라인, 평행이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 좌측 상단의 BoardRouteManager class는 각 보드별로 루트를 어떻게 제어할 지에 대한 내용이 담겨있다. Static class이기 때문에 어느 class에서도 접근이 가능하다.
* JFrame을 상속받는 class가 위의 사진에서 2개가 보일 것이다. 하나는 초기 화면인 FirstPage class, 나머지 하나는 게임의 재시작 혹은 종료를 선택하는 EndPage class이다.
* FirstPage에서 Player, Piece, Board에 대한 정보를 Radio Button으로 받게 되는데 이를 바로 PlayGame이라는 controller에 넘기는 것이 아닌 중간에 Adapter을 사용하여 직접적인 연결을 회피하는 Indirection Pattern을 적용하였다.
* PlayGame은 게임을 총괄하는 Controller로 윷놀이 게임을 여러 개의 phase로 나누어 관리한다. Throw, place, move, turn change와 같은 동작 각각을 phase라는 method로 나누었는데 이는 다른 사람이 phase를 추가함으로써 게임을 더 다채롭게 할 수 있게 만들어 준다. 각 phase는 서로 유기적으로 연결되어 있어 PlayGame class는 거대하지만 강력한 cohesion을 보여준다.
* 각각의 버튼은 ActionListener을 Implement함으로써 다른 기능을 수행한다.

텍스트, 스크린샷, 도표, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* PlayConfig는 Adapter을 통해 Player 수, Piece 개수, 그리고 판의 모양을 받아온다. Getter을 통해 각각의 값을 integer 형태로 받은 후 PlayGame controller가 PlayConfig의 값을 가져가서 생성할 게임의 유형을 판단하고 조정하게 된다.
* YootBoardController는 게임 화면에 있는 Board 부분에 적용되는 값을 조정하는 class이다. 모든 플레이어의 말을 관리하는 부분이며 말의 위치 및 출발점을 통과한 말의 개수 모두를 관장하는 Controller이다. 여기서 중요한 부분은 updateBoard와 updateYootStack이다. 전자는 플레이어가 여러 번 던졌을 때 각각의 결과를 Jbutton으로 추가하여 움직일 방법을 선택하게 해주며 Jbutton을 선택 후 움직이고 났을 시 Board 위에 player의 말을 표시하는 것을 도와준다. 한마디로 Board의 상태를 실시간으로 동기화하는 것을 도와준다. 후자는 전자의 Jbutton에 추가될 움직임 규칙을 저장하는 list를 관리하는 method이다. 플레이어가 원하는 이동 규칙을 적용하기 위해 UI에 logic을 적용해주는 method인 것이다.
* PlayerHandler는 Player의 값을 조정하는 Handler class이다. Player가 직접적으로 domain layer의 logic을 시행하는 것이 아닌 Handler class가 Player의 값을 getter로 받아와서 Player가 가진 정보를 이용해 연산을 진행하게 된다. 윷놀이 프로젝트를 하면서 실제 사람이 말을 움직이는 것을 보고 Player class 쪽에 Piece에 대한 간단한 연산을 추가하였었다. 이런 경우 일반적으로 cohesion이 낮을 것으로 예상되지만 Player가 굴린 윷의 결과를 move관련 method가 받아 처리하므로 각 method끼리의 협업이 잘 활성화 되어있다.
* Yoot class는 게임에서 사용되는 윷에 대한 정보를 가지고 있는 static한 attribute, behavior을 가진 class이다. [백도, 도, 개, 걸, 윷, 모]에 대한 정보를 가지고 있으며 윷의 모양이 둥근 부분과 평탄한 부분으로 나뉜다는 것을 고려하여 확률을 조정하는 부분도 있다. 지금 보면 throw에 대한 부분이 6가지 종류 밖에 없어 따로 구현한 것을 볼 수 있는데, Interface를 통해 하나의 형태로 묶어 표현하면 좋을 것 같다.

텍스트, 스크린샷, 도표, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* YootBoard는 JFrame을 extends하는 UI관련 class이다. Controller class에서의 결과를 method로 호출하여 JLabel을 통해 표시 혹은 Button을 그려 그 결과를 시각적으로 보여주는 class이다.
* Player class는 게임에 참여하는 사람들에 대한 정보를 가진 model 계층의 class이다. 이번 프로젝트를 구상할 때 사람이 직접 말을 옮기고 윷을 던지는 것에 착안하여 Player class가 그것에 대한 역할을 다하면 좋을 것이라 생각했다. SRP 입장에서 봤을 때에는 ‘말을 옮기는 책임’, ‘윷을 던지는 책임’과 같이 두 가지의 책임을 가지게 된다고 볼 수 있다. 하지만 변하지 않는 결과에 대해선 책임이 커져도 상관이 없다는 것이 중론이다. Java.util의 경우 많은 책임을 가지고 있지만 문제가 없는 것처럼 ‘윷을 던지는 책임’은 윷을 던지는 행위에 변화가 없을 것이 예상되므로 하나의 class에 책임을 부여해도 큰 문제가 발생하지 않는다고 생각한다.
* PieceHandler class는 Piece들이 가지고 있는 고유 정보를 식별 및 인식하고 정보를 수정하는 class이다. Piece의 좌표 값을 받아 좌표 값을 수정해 가운데 경로로 들어갈 수 있도록 유도하며 말이 업혀 있는지도 판단하게 된다.

텍스트, 스크린샷, 도표, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* Piece class는 Piece의 domain logic을 가지고 있는 class이다. PieceHandler의 경우 UI와 Controller 사이를 통제해주는 추상화 계층으로 볼 수 있고 Piece는 추상화 계층에 정보를 제공하는 domain layer의 class이다. Domain logic이 model 계층인 Piece에 있으므로 unit test 및 유지보수시 많은 이점을 가질 수 있게 된다. (아래는 노션에 정리한 Piece와 PieceHandler 간의 관계를 나타낸 것이다.)

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* IBoardRouteManager는 보드의 종류별 경로 처리에 대한 추상 인터페이스이다. Piece의 위치, 이동, 백도 처리, 경로 전환을 주로 담당한다. 추상 인터페이스이기 때문에 board의 모양에 따라 각각의 method가 다르게 이용된다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 위와 같이 board의 모양에 따라 경로의 개수, 특이점을 따로 관리하기 때문에 유지 보수 및 디버깅이 단순화된다.
* 특히 이는 구조확장에 좋다. SOLID에서 OCP는 확장에 대해선 열려있으며 수정에 대해선 닫혀야 한다고 말한다. 위와 같은 구조는 N각형을 추가했을 때 [N]BoardRouteManager을 따로 생성하여 사용하면 된다. Interface를 직접 수정하는 것이 아닌 새로운 class만 생성하면 된다는 점에서 이 구조는 OCP를 잘 만족한다고 볼 수 있다.

**D – 1 - 2) Sequence Diagram**

텍스트, 도표, 평행, 평면도이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* Sequence diagram이다. 하나의 actor을 배치하여 한 명의 player가 게임을 진행하는 것 같지만 여러 명이 진행한다는 것을 가정하고 설계하였다.
* 처음에 actor가 게임 시작 버튼을 누르면 PlayConfig가 FirstPage에 있는 actionperformer에 의해 인자를 가지게 된다. 받은 정보를 묶어서 FirstPage의 시작 버튼의 이벤트에 매핑하게 되고 PlayGame이 생성된다. PlayGame은 PlayConfig에서 정보를 getter로 추출하게 된다. 이를 토대로 규칙을 만들고 Player, YootBoard, Controller을 생성한다.
* 윷놀이 판이 생성되면 본격적으로 게임이 진행된다. 승자가 나올 때까지 게임은 반복된다. 판에서 Roll 버튼을 누르면 PlayGame이 이를 인식하여 throwing method를 호출해 Yoot한테 결과를 요청한다. 결과를 받고 게임 창에 결과를 보여주고 List에 임시로 결과를 추가한다. (List에 결과를 추가하는 이유는 윷, 모와 같이 다시 던지는 처리가 들어왔을 경우 나중에 플레이어가 움직이는 순서를 manual하게 선택할 수 있게 만들기 위함이다.) 윷이나 모가 나오면 이전 과정을 한번 거치고 List에 값을 추가하게 된다. 플레이어는 나온 결과를 선택하고 판에 있는 자신의 말을 움직이거나 내보내는 버튼을 선택하게 된다. 내보내는 결과를 선택하면 화면에 보여줄 말을 생성하고 움직이게 되고 그렇지 않은 경우 판위에서 선택한 말을 토대로 좌표 연산을 시작한다. 움직였을 때 자신의 말이 있으면 merge를 통해 grouping을 시행한다. 그렇지 않고 상대방의 말이 있다면 다른 플레이어한테 잡혔다는 신호를 전달하여 플레이어가 자신의 상태를 update할 수 있게 유도한다. 움직임이 끝난 후 게임이 끝날 수 있는지 알아본다. (Player가 가지고 있는 말의 개수를 토대로 게임의 승패 여부를 결정하게 된다.) 승패가 결단이 나지 않은 경우 다시 처음부터 이러한 과정을 시행한다.
* 승자가 나오면 EndPage로 화면이 바뀐다. 재시작을 누른 경우 초기 게임 세팅부터 시작하게 되므로 ref box로 이러한 선택을 그렸다. 재시작을 누르지 않고 종료를 누른 경우 exit()을 이용하기에 system을 통해 main memory를 해제하게 된다.

**D – 1 - 3) MVC Architecture Diagram**

도표, 평면도, 기술 도면, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 프로젝트의 class를 위와 같이 Model, Controller, View로 분리할 수 있다. View와 Controller가 UI Layer에 속하게 되고 Model은 Non UI Layer에 속하게 된다. Model 계층은 낮은 계층이기 때문에 Controller와 View를 참조할 수 없다.
* View 계층에는 화면을 나타내는 FirstPage, YootBoard, EndPage 계층이 있다. 각각 Controller와 View 계층 내에서 상호작용을 하게 된다.
* FirstPage의 값은 Indirect하게 Model에 전달되는데 여기서 중간자 역할을 하는 것이 추상화 계층인 Controller 계층이다. Adapter, Controller와 같은 이름을 이용한다. Model에서의 Handler은 Model을 사용하기 편리하게 일부 기능을 분리한 것이지만 Adapter와 Controller는 UI에서 들어온 event를 domain 계층에서 인식할 수 있게 값을 가공 혹은 인터페이스를 제공해준다.
* Model 계층에 Interface가 있기에 이상하다고 생각할 수 있다. Linskov 치환 원리에 의해 어떠한 board가 주어져도 규칙을 적용하여 게임을 진행할 수 있어야 한다. Fragile하지 않은 구조를 위해 각 Board manager을 따로 만들어서 사용할 수도 있다. 하지만 이는 조건 분기의 취약점을 이용하게 된다. 교재에서 조건 분기를 이용한 특정 method 호출은 새로운 구조의 확장을 억제하고 특정 method에 의해 fragile할 수 있게 된다고 나와있다. 이러한 문제점을 인식하였기에 Interface를 이용해 board마다의 이동 규칙을 따로 적용하였다. 다른 domain 부분에 대해서 interface를 적용하지 않은 이유는 어떠한 type의 board일지라도 적용되는 것이 동일하기 때문에 추상화 계층을 통해 나눌 의무가 없었다.

D – 1 - 4) 의존성 그래프

텍스트, 도표, 라인, 평면도이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 의존성 그래프에서의 화살표는 visibility를 의미한다고 생각하면 안 된다. 여기서의 화살표는 의존성 방향을 나타낸다. FirstPage의 경우 입력 값이 Adapter, Interface, PlayGame에 영향을 미치기에 화살표 방향이 FirstPage에서 상대방 class 방향으로 가리키고 있다.
* Interface를 이용하는 ‘xxxRouteManger’ class의 경우 interface를 이용하지만 implement라는 특수 관계가 얽혀 있기에 implement 기호를 이용하여 의존성을 표시하였다.

**D – 2) 프로젝트 구현**

D – 2 - 1) Code

D – 2 – 1 - 1) FirstPage

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* FirstPage의 코드이다. 처음에 선언한 static int의 형태의 상수는 실제 플레이어, 말의 개수가 아닌 선택할 수 있는 종류의 개수를 말한다. 플레이어의 명수는 [2, 3, 4]이기 때문에 MAX\_PLAYER = 3이 된다. 이는 Radio Button의 개수를 위해 생성한 것으로 생각하면 된다.
* 각각의 adapter class는 playconfig의 값을 받을 준비를 한다. 아래의 코드는 FirstPage의 생성자 내부의 EventListner이다. 여기서 버튼의 입력값을 config가 가지게 되고 이를 adapter에 분배하는 형식을 볼 수 있다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

D – 2 – 1 - 2) PlayConfig

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* PlayConfig 내부의 method를 보면 default로 게임 인원 수 = 2, 말의 개수 = 2, 판의 모양 = 사각형으로 지정하여 사용자가 입력을 까먹었을 시 이를 이용할 수 있게 설계하였다.

D – 2 – 1 - 3) PlayerAdapter

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* Adapter class의 내부를 살펴보면 PlayConfig를 인자로 받아 내부에 자신의 역할에 해당하는 값을 받아온다. 또한 FirstPage의 Event button을 이용해 값을 직접 받아 그 값을 string -> int로 변환한다. Int로 변환한 값을 PlayConfig의 setter로 넘겨 config 값을 설정해준다. 생성자의 내부에 있는 playConfig는 이처럼 FirstPage에서의 Event Listener의 값을 받고 값을 할당 받은 뒤에 형성된 것임을 알 수 있다. (다른 Adapter도 위와 동일한 구조이므로 설명을 생략한다.)

D – 2 – 1 - 4) YootBoard

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

YootBoard class 내에 특별한 logic은 존재하지 않는다. 이는 YootBoard class가 순수 View를 담당하기 때문이다. (아래에는 주요 method에 대한 작동에 대한 notion)

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

D – 2 – 1 - 5) PlayGame

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 위의 코드는 PlayGame class 내의 phase 중 던지기 phase에 대한 코드이다. Yoot class에서 값을 가져와 보여주고 이를 list에 추가하는 것을 볼 수 있다. 던진 결과가 윷, 모인 경우 던진 기록을 초기화하고 다시 phase 1으로 돌아가게 하는 것을 볼 수 있다.
* 기능을 자세히 설명하자면 다음과 같다. 1) 현재 phase가 1인지 확인 후 실행 -> 2) Yoot.throwing()을 통해 윷 결과(도, 개, 걸 등)를 구함 -> 3) extraTurnList에 결과값 저장 (스택처럼 여러번 이동 가능하도록) -> 4) YootBoard에 결과 표시 (showThrowResult) -> 5) 결과가 **윷 or 모**일 경우 한 번 더 던지도록 phase를 유지 -> 6) 이외의 경우 controlPhase를 2로 설정하여 다음 단계로 이동

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다. 텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 위의 코드는 phase 2: 새로운 말 선택이다. 특이사항은 만약 백도가 나왔다면 생성 없이 바로 턴을 넘긴다는 것이다. 이 method의 기능은 1) Player.createPiece()를 통해 대기 중인 말 생성 -> 2) 시작 위치(route=0, pos=0)에서 해당 moveValue만큼 이동 -> 3) 도착 체크 및 점수 반영 (checkAndHandleArrival) -> 4) 내 말과 병합 여부 확인 (checkAndMergeStack) -> 5) 상대 말을 잡았는지 확인 → 잡았을 경우 턴 유지 및 윷 스택 유지 -> 6) 게임 종료 체크 (checkWinner) -> 7) 스택 값 소모 처리 (consumeSelectedStackValueOnce) ->8) 더 이동 가능한 스택 값이 없다면 다음 턴으로 전환으로 설명할 수 있다.

텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 이전 페이지에 있는 코드는 이미 판 위에 있는 말을 움직이는 Phase 3이다. Method의 작동 방식은 다음과 같다. 1) Player.movePieceAt(...) 호출로 해당 좌표의 말을 이동 -> 2) 도착 체크 및 점수 반영 (checkAndHandleArrival) -> 3) 내 말과 병합 여부 확인 (checkAndMergeStack) -> 4) 상대 말이 잡혔는지 확인 → 잡았을 경우 턴 유지 -> 5) 게임 종료 체크 (checkWinner) -> 6) 스택 소모 처리 및 다음 턴으로 전환 또는 추가 턴 유지

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 위의 method는 다음 턴으로 넘기는 nextTurn method이다. 현재 플레이어가 승리자가 아닌 경우 아래와 같이 모듈러스 연산을 통해 플레이어를 인식할 수 있도록 연산한 것을 볼 수 있다. Player과 turn의 값이 1차이가 발생하는 이유는 player 자체의 값은 index이 될 때 0부터 시작했기 때문에 P1은 0번째 인덱스에 있게 된다. 이를 P1으로 표기하기 위해선 +1을 해야한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 위의 method는 각 phase에서 승리조건을 판별할 때 사용하는 checkWinner method이다. Score의 값이 플레이어가 초기에 할당 받은 값과 동일하다면 승리로 판단한다.

D – 2 – 1 - 6) Yoot

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 위의 method는 Yoot class에서 윷을 던졌을 때의 결과를 출력해주는 method이다. 윷의 앞면과 뒷면의 평탄도가 다르기에 PROB\_FLAT = 0.6f로 잡았다. 확률에 따라 random variable이 조건을 만족하는 지 안 하는 지를 판단하게 된다. 이를 통해 임의의 sum 값을 가지게 되고 나온 값을 토대로 각각 도, 개, 걸, 윷, 모를 나타낸다. 백도의 경우 윷 가락 하나의 결과를 토대로 백도인지 도인지를 구분하게 된다.

D – 2 – 1 - 6) YootBoardController

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 위의 코드는 YootBoardController의 코드 중 주요 method이다.
* updateYootStack(List<Integer>, ActionListener) 는 윷 결과 스택(extraTurnList)을 UI 패널에 버튼으로 표시하고 클릭 이벤트를 연결한다. updateBoard(List<Player>, ActionListener)는 플레이어 말들의 현재 위치를 보드 위에 그래픽 버튼으로 렌더링 및 상태 레이블 업데이트한다. updateTurn(int)은 턴을 UI 상단 레이블에 표시 (현재 차례: P1 등)한다. getpieceBtn()은 선택된 말 버튼에 대한 참조 반환한다.
* 이 클래스가 주로 하는 것은 UI 상태 동기화, 버튼 구성 및 이벤트의 연결 (중간자 역할), 그리고 겹침 처리 및 시각적 우선순위 조정이 있다.
* YootBoardController의 핵심 가치는 View layer인 YootBoard와 Model layer인 Player과 Piece 사이를 연결하는 것에 있다. 이 클래스는 두 계층 사이를 연결하며 보드 업데이트, 결과 출력, 턴 전환과 같은 화면 연출을 수행한다.

D – 2 – 1 - 7) Player

* Player class는 게임 내의 참여자에 대한 클래스로 중요한 위치에 있다. Getter와 같은 정보를 제공하는 것이 대부분이지만 그 외의 주요한 역할을 제공하고 있다. Getter의 경우 해당 정보만 return하는 것이므로 getter 외의 주요 method에 대한 코드 설명을 할 것이다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 자신이 보유하고 있는 말 중 하나를 내보내는 method이다. 대기 중인 말(standbyPieces)이 있다면 보드 위에 새 말(Piece)을 생성하여 pieces 리스트에 추가한다. 내보낸 말은 시작 위치로 초기화되고 플레이어의 대기 말 개수가 감소한다. 이는 UI에서 ‘내보내기’ 버튼 클릭 시 활성화된다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 특정 좌표에 말의 개수를 확인하는 method이다. 특정 좌표(route, position)에 있는 내 말이 있다면 해당 말 위에 업혀 있는 말 개수(point)를 반환한다. 없다면 0을 반환한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 말을 이동하는 method이다. 특정 위치에 있는 내 말을 찾은 후 move(step) 호출을 한다. 논리적 좌표 기반으로 이동을 수행하게 되고 말을 선택한 후 스택 값으로 이동시 호출되는 method이다.

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 말이 판에서 나왔는지 확인 및 점수에 대한 처리를 하는 method이다. 말 리스트 중 hasArrived()를 만족하는 말이 있다면: 1) 그 말을 제거 -> 2) 해당 말의 포인트만큼 점수(score) 누적을 진행한다. 도착한 말이 있다면 true를 반환한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 해당 좌표에 특정 플레이어의 말이 있는지를 확인하는 method이다. 해당 좌표에 위치한 말의 인덱스를 반환하지만 없다면 -1을 반환한다. 말을 선택 시 자신의 말인지 아닌지를 판별하기 위해 사용된다.

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 말을 업는 method이다. 동일한 위치에 존재하는 말들을 찾아 병합하며 선착한 말(a)에 다른 말(b)의 포인트를 더하고 b는 제거한다. 성공적으로 병합되면 true를 반환한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 상대 방 말을 잡는 method이다. 특정 좌표에 있는 말을 찾아 제거하고 해당 말의 point만큼 standbyPieces 증가 (말을 다시 대기 상태로 되돌림)한다. 잡은 경우 true를 반환한다.

D – 2 – 1 - 7) PlayerHandler

* PlayerHandler 클래스는 Player 객체에 대한 얕은 래퍼(Wrapper) 역할을 하며, UI 또는 컨트롤러가 Player 객체를 직접 접근하지 않고 조작할 수 있도록 위임 메서드(delegate methods)를 제공한다. 즉, 게임 로직과 UI 계층 간의 중간 다리 역할을 한다.
* 그렇지만 PlayerHandler 클래스를 controller라고 보진 않는다. 사용자의 입력을 직접 처리하는 것이 아닌 PlayGame이라는 controller를 보조하는 도구로써 사용되기 때문이다. 그래서 이 클래스는 Model과 Controller 사이의 Adapter 역할을 하는 class로 취급한다.

텍스트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 위의 코드를 보면 대부분 Player에 있는 class를 위임받은 것임을 알 수 있다. 이러한 구조는 Player 객체가 바뀌더라도 Handler를 통해 인터페이스만 유지하면 다른 계층의 변경은 불필요하다는 장점을 가지게 된다.
* 이 클래스의 역할은 Player 클래스의 메서드를 호출하여 게임 상태를 변경하거나 조회하는 것, 외부에서 Player 내부 구조를 숨기고 필요한 인터페이스만 노출하게 한다.
* Player에는 없고 PlayerHandler에만 있는 method 중에서 주요 method는 hasPieceAt(int x, int y)이다. 내부적으로 player.findPieceIndexAt(x, y) >= 0을 이용하여 지정 위치에 플레이어의 말이 존재하는지 확인한다. 이는 Player에서는 직접 제공되지 않는 추상화된 편의 method이다.

D – 2 – 1 - 8) Piece

텍스트, 폰트, 스크린샷이(가) 표시된 사진

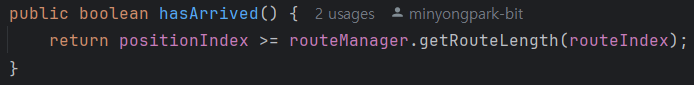
AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 위의 클래스는 움직임을 처리하는 method이다. handlePathChage를 통해 어디로 갈 지 조정하게 된다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 위치 이동 후, 경로 전환 지점인지 확인하고 경로 변경 처리를 하는 method이다.
* DI 방식으로 field에 interface가 있는데 이를 토대로 특정 판 모양에서 적용할 움직임 규칙을 적용하여 움직인다. (private final IBoardRouteManager routeManager;)



* 현재 위치가 도착 지점을 넘었는지 여부 확인하는 method이다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 위치를 비교하는 method로 움직이는 말과 멈춰 있는 말과의 좌표 비교를 할 때 사용한다.

D – 2 – 1 - 9) PieceHandler

* PlayerHandler와 같은 역할을 수행하며 Piece가 UI와 Controller끼리 상호작용할 수 있도록 도와준다. Piece의 내부 로직을 몰라도 PieceHandler의 로직을 통해 Piece의 내부기능을 모두 사용할 수 있다.
* 이러한 Handler class의 사용은 Piece의 내부 로직을 변경하지 않고도, PieceHandler를 통해 기능을 확장하거나 조정 가능하게 해주며 이는 OCP를 잘 따른다고 볼 수 있다. 추상화에 대한 많은 경험이 없어 DIP를 명확히 적용했다 볼 수 없지만 만약 향후 IPiece 같은 추상화가 도입된다면 자연스럽게 DIP 구조로 확장 가능하다. PieceHandler는 외부에서 접근 가능한 인터페이스만 제공하기에 SRP를 만족한다고 할 수 있다. 이처럼 Handler classes (Player, Piece)는 SOLID에서 SRP와 OCP를 잘 만족한다고 볼 수 있다.
* GRASP의 관점으로 볼 때 Handler classes들은 다음의 원칙을 만족한다고 볼 수 있다. PieceHandler는 사용자 입력에 따른 도메인 객체의 조작을 담당 그리고 말 이동, 병합, 도착 처리 등 도메인 객체에 메시지를 전달하고 그 결과를 반환하므로 Controller의 역할을 한다고 볼 수도 있다. (MVC에서의 controller와는 다름) PieceHandler는 Piece 객체를 알고 있으므로, “말이 어디에 있냐”, “병합 가능한가”, “도착했는가” 등의 판단을 가장 잘 할 수 있는 위치에 있다. 이는 Information Expert 규칙이 잘 적용되었음을 보여준다. 마지막으로 UI, PlayGame 등은 Piece에 직접 접근하지 않고 PieceHandler를 통해 우회적으로 처리하게 되므로 low coupling을 통한 change propagation을 방지할 수 있게 된다.

텍스트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

D – 2 – 1 - 10) BoardRouteManager

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 위의 코드는 BoardRouteManager 클래스이다. 직접적으로 logic을 처리하는 class는 아니지만 매우 중요한 역할을 가진다. 현재 위치(route, position)에서 이동한 결과, 새로운 경로와 위치로 전환해야 할 때 주로 사용한다. (예: 사각형 판에서 외곽을 돌다가 중앙으로 진입하는 경우, RouteChange 객체를 반환하여 경로 전환을 알림.)
* 이러한 형태의 클래스는 여러 보드에서 동일한 구조를 공유하도록 한다. (재사용성 증가) 또한 유저의 말이 특정 지점에 도달했을 때 보드 구조에 따라 다른 경로로 이동 가능하도록 지원한다.
* 이는 GRASP의 Information Expert, Low Coupling, High Cohesion 원칙을 모두 따른 설계로 볼 수 있다. RouteChange는 경로 변경에 필요한 정보를 직접 가지고 있고, 그 자체가 경로 전환 정보의 전문가로 설계되었다고 볼 수 있다. BoardRouteManager.RouteChange는 단순 데이터 구조이며, 어디에나 전달 가능하지만 별도 의존성을 갖지 않기 때문에 결합도가 낮다. SquareBoardRouteManager, PentBoardRouteManager, HexBoardRouteManager는 필요할 때만 이 구조를 반환하고, 서로 독립적으로 유지된다. RouteChange 클래스는 오직 경로 전환을 표현하기 위한 목적으로만 설계됨. 불필요한 기능이 없고 단일 책임에 충실하므로 높은 응집성을 가지며 SRP를 만족한다고도 볼 수 있다.

D – 2 – 1 - 11) IBoardRouteManager

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* getRouteChange(int routeIndex, int positionIndex)는 해당 위치에 도달했을 때 다른 경로로 분기할 조건이 있는지 확인하여, 있다면 RouteChange 객체로 반환한다. getRouteLength(int routeIndex)는 특정 경로(route)의 총 길이(칸 수)를 반환한다. getStartRoute()는 게임 시작 시 말이 출발하는 초기 경로 번호를 반환 (보통 외곽 경로가 0번)한다. getStartPosition()는 게임 시작 시 말이 위치하는 초기 경로 내 인덱스를 반환 (보통 0)한다. adjustForBackdo(int routeIndex, int positionIndex)는 백도(BACKDO)처럼 음수 위치에 도달했을 때의 보정 처리 로직이다. (예: -1 → route 0의 마지막 위치 등으로 변환)
* 해당 인터페이스의 설계 목적 및 특징은 다음과 같다. 1) **보드 형태 전략 추상화:** 게임에는 다양한 형태의 보드 (4각형, 5각형, 6각형 등)별로 서로 다른 경로 규칙이 있다. IBoardRouteManager는 이를 인터페이스로 추상화하여 각 구현체(SquareBoardRouteManager, PentBoardRouteManager, HexBoardRouteManager)에 위임한다. **2. SRP (단일 책임 원칙):** 이 인터페이스는 경로 계산 및 분기 처리에만 책임을 갖는다. 경로 외 다른 기능은 전혀 포함하지 않는다. **3. OCP (개방-폐쇄 원칙):** 보드 종류가 늘어나더라도 IBoardRouteManager를 구현한 새로운 클래스만 추가하면 된다. 기존 코드 변경 없이 보드 기능 확장이 가능하다. **4. Strategy Pattern:** 보드의 이동 전략을 런타임에 다형적으로 적용할 수 있다. (예: if (boardShape == 5) routeManager = new PentBoardRouteManager();) 그리고 GRASP의 관점에서 볼 때 이러한 인터페이스는 Information Expert, Low Coupling, High Cohesion을 만족한다고 볼 수 있다.

D – 2 – 2) JavaFX로 구현할 시 변경 점

* **환경설정**
* **JavaFX를 사용할 때엔 우선 외부 라이브러리를 설치해야 한다. JDK 11버전 이전의 경우 JavaFX가 내장되었지만 이후부턴 Swing만 내장되어 있다.**

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* JavaFX를 Gluon에서 다운받고 난 후 압축을 해제하고 나면 다음과 같은 4개의 파일이 보일 것이다.
* 보이는 directory 중 bin을 Intelij에 경로를 추가해주어야 한다.

스크린샷, 텍스트, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* Intelij의 프로젝트 구조에 들어가서 module을 들어가면 다음과 같은 화면이 뜬다. 여기서 ‘+’ 버튼을 눌러 이전 사진에서의 lib 경로를 추가해줘야 한다.
* 이후 Configuration도 수정해주어야 한다. (자세한 설명은 다음 페이지에 이어서)

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 위와 같은 형식으로 Configuration을 추가해주어야 한다. Build and run에서 JDK를 21버전으로 만들어 준 후 -cp를 project folder로 설정한다. 이후 바로 아래의 VM 추가를 보면 ‘--module-path "C:\javafx\javafx-sdk-21.0.7\lib" --add-modules javafx.controls,javafx.fxml’ 과 같이 설정해주어야 한다. "C:\javafx\javafx-sdk-21.0.7\lib" 이 부분은 위의 lib 주소이므로 사용자마다 다를 것이므로 주의해야 한다. (프로젝트 하위에 javafx-sdk-21.07/lib가 존재하므로 개인적으로 세팅할 경우 이 경로를 활용하면 간단하게 세팅할 수 있다.)
* 그 아래를 보면 실행할 File을 설정하는 곳이 보이는데 이 부분에서 FirstPagefx를 선택하면 세팅이 끝난다.
* **UI의 특징 (JavaFX)**
* 기존의 Swing UI는 픽셀 단위로 위젯을 직접 배치하는 형식을 이용하였다. 반면 JavaFX의 경우 Swing의 불편한 View 배치를 해소하기 위해 XML 방식의 자료 배치를 이용한다. HTML, CSS의 방식과 동일하게 FXML의 형식의 배치 방식을 이용하는데 이는 웹 컴포넌트의 배치 방식과 유사하다. 즉 웹 개발 경험이 있는 경우 Swing보다 더 정교하게 UI를 구성할 수 있다.
* Swing과 다르게 외부 media를 잘 이용할 수 있다는 점이 있어 영상, 실시간 차트와 같은 내용을 넣기도 좋다.
* **코드의 주요 변경점**

1. **fXML 확장자를 이용한 view 구현**

* JavaFX의 경우 HTML, CSS와 같은 Markup 문법의 언어를 이용한 UI 배치를 지원하고 SceneBuilder라는 미리 보기 tool이 fxml 코드를 이용하기 때문에 기존의 Page를 그리던 코드를 fxml의 확장자에 맞게 바꾸었다.

텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* FirstPagefx.java의 view loader 함수이다. Loader 변수의 경우 fxml 파일을 가져와서 Parent 추상 클래스의 root instance에 이를 넘겨준다. Parent는 모든 레이아웃 컨테이너의 최상위 타입으로 root를 통해 모든 레이아웃을 그릴 수 있게 해준다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 이전 페이지의 마지막 사진은 FirstPagefx.fxml의 일부 코드이다. CSS와 비슷하게 bracket을 통해 컴포넌트의 성질을 나타내고 각 컴포넌트의 constraints 및 성질을 <> 사이에 배치하여 화면을 구성한다.

1. **동적인 버튼 배치를 YootBoardController로 기능 이전**

* JavaFX에서는 @FXML annotation 통해 FXML 파일에 정의된 UI 요소들을 컨트롤러에서 조작할 수 있다. Java Swing은 JPanel, JButton 등을 코드 상에서 자유롭게 생성 및 배치했지만, JavaFX는 FXML을 통해 레이아웃을 정의하고, 그 정의된 요소들은 오직 해당 FXML에 연결된 컨트롤러에서만 접근이 가능했다. 따라서 YootBoard.fxml의 동적 버튼 초기화 코드를 FXML과 연결된 YootBoardController 내 initBoard() 등에서 수행하도록 리팩토링하였다. (아래의 사진은 @FXML을 이용한 UI 요소의 예시이다.)
* 텍스트, 폰트, 스크린샷, 그래픽이(가) 표시된 사진

  AI가 생성한 콘텐츠는 부정확할 수 있습니다..

1. **UI 컴포넌트 배열 방식 변경**

* FXML의 fx:id는 단일 UI 요소에만 이름을 붙일 수 있음. 배열이나 리스트를 직접 FXML에서 만들 수 없다. 이를 해결하기 위해 FXML에서 각각의 버튼에 fx:id="btn00", btn01 등 개별 식별자를 지정하고, 컨트롤러에서 @FXML Button btn00, btn01, ..., btn44; 처럼 멤버 변수로 선언한 뒤, initialize() 메서드에서 배열이나 2차원 배열로 수작업으로 구성하였다.

1. **버튼 초기화 방식 변경**

* Swing의 경우 new JButton(new ImageIcon("image.png")) 처럼 생성자에서 바로 이미지 삽입이 가능했다. 위젯을 코드로 직접 관리하였기 때문이다. 반면 JavaFX는 Button 생성 후 setGraphic(new ImageView(new Image("image.png")))와 같이 별도로 이미지 설정 필요. JavaFX는 Graphic 요소(예: 이미지, 아이콘 등)를 명시적으로 설정해야 하기 때문이다. 이는 웹 프로그램의 랜더링과 유사하다.

1. **뷰 컴포넌트 접근 방식 변경**

* JavaFX에선 다른 클래스에서 직접 @FXML 노드(버튼, 레이블 등)를 참조할 수 없다. 따라서 FXMLLoader로 FXML을 로딩한 클래스에서 ‘.getController()’를 호출하여 컨트롤러 참조를 얻었다. 필요한 정보를 컨트롤러의 메서드를 통해 간접적으로 전달하거나 요청함. 이를 통해 JavaFX의 indirect 성질을 볼 수 있다.

1. **Action Listener를 Event Handler로 변경**

* Java Swing은 Action Listener를 통해 사용자와 UI 간의 interface를 제공했지만 JavaFX는 button.setOnAction(e -> doSomething())과 같이 javafx.event.EventHandler 기반 구조를 사용하며, addActionListener는 존재하지 않는다.

1. **이벤트 식별자 방식 변경**

* Swing의 방식은 다음과 같이 button.setActionCommand("ROLL\_YUT") → 리스너에서 getActionCommand()로 분기 처리하였다. JavaFX의 경우 button.setId("ROLL\_YUT")를 사용해 ID를 저장하였고 추가 데이터가 필요할 경우 button.setUserData(...) 활용하였다.

1. **Dispose() -> Stage.close()**

* Swing의 경우 Dispose() 함수를 통해 창을 닫았지만 JavaFX에선 이를 지원하지 않는다. 대신 Stage.close()라는 함수를 제공한다. 결국 stage에 대한 정보가 필요한데 컨트롤러 또는 외부에서 Stage 객체 참조 유지하며 Controller 클래스나 FXMLLoader를 통해 Stage를 전달하거나 저장해두고 stage.close() 호출하는 방식을 통해 창을 닫을 수 있다.
* **정리 (JavaFX 코딩 후기)**
* MVC 형식의 프로그램을 작성하였지만 Java Swing의 경우 Legacy 형태의 동적 widget 방식의 UI를 사용하고 JavaFX의 경우 web과 유사한 markup 언어를 사용한다는 점에서 생각보다 코드를 많이 바꾸게 되었다.
* 화면을 생성할 때 Null에 대한 예외 처리가 항상 동반되었으며 pixel을 통한 직접적인 UI 배치가 아닌 CSS 스타일의 화면 배치 방식은 WEB 개발 경험이 전무한 사람들이 코딩하기에 학습의 cost 및 trial이 컸다.
* Swing의 경우 배열과 반복문을 이용한 UI 배치를 통해 간편하게 UI를 배치할 수 있었지만 JavaFX의 경우 버튼 하나하나의 id를 따로 지정하는 방식으로 인해 이러한 방법을 사용하지 못했다.
* Swing과 달리 JavaFX는 JDK 11버전부터 외부 module로 분리되었기 때문에 사전에 환경 세팅이 복잡하다는 단점이 있었다. JDK별 다른 JavaFX 모듈이 존재했으며 버전간 호환성 문제가 있기에 협업에서 버전 합의는 중요한 이슈로 작용했었다.
* Swing과 달리 프로그램의 UI가 웹페이지 느낌이 강하게 난다.

텍스트, 스크린샷, 소프트웨어, 컴퓨터 아이콘이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.텍스트, 스크린샷, 도표, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* Front-end 웹 개발 경험이 적은 경우 이러한 프로그램을 만들 때 UI 구성에 많은 어려움을 가질 것 같다는 생각이 많이 들었다.

**E) Junit Test**

**E - 1) Test 환경 구축**

* 1) 테스트 폴더를 project 폴더 하위에 새로 추가해준다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 2) 생성된 폴더에서 모듈 세팅을 ‘Tests’로 지정하면 위와 같이 초록색 폴더로 바뀐다.
* 3) 테스트할 class 파일을 생성한다. 생성 시 Junit library의 dependency가 설정이 되어 있지 않으면 Fix 버튼을 눌러 종속성을 만족시켜준다.

텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* (위의 그림은 프로젝트와 무관하지만 동일하게 진행하면 됨)
* 4) 생성된 test class 파일은 테스트를 진행할 package와 동일한 명칭의 package 하위 폴더에 위치해야 한다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 5) Test 코드를 작성 후 우클릭을 하면 아래의 사진과 같은 옵션이 뜨는데 Run ‘Test’를 누르면 Junit test가 실행된다.

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 6) Test가 진행되면 각 테스트가 성공 혹은 실패했는지를 색으로 표현해준다. 초록색은 성공을 빨간색은 실패를 나타낸다. Test class 자체가 초록색으로 빛나는 경우 모든 test case를 통과했다는 것을 알려준다. 테스트 진행 결과 및 src의 package 이름 그리고 테스트 진행 시간도 알려준다.

 (모든 테스트가 통과한 모습)



* 테스트 진행 후 테스트 코드 파일에 들어가면 좌측 중단점을 찍는 부분에 초록색으로 체크 표시가 보일 것이다. 빨간색 ‘X’ 표시는 assert의 결과가 False임을, 초록색 ‘V’ 표시는 True가 return 되었음을 알려준다. Test class 하위의 모든 method의 선언 부분의 좌측에는 위와 같이 성공 여부가 모두 표시되어 있다.
* 모든 Junit test를 진행하기 위해선 test method 위에 ‘@Test’를 표시해주어야 한다. Machine이 notation을 보고 해당 method가 unit test인지 아닌지를 판단하게 된다.

E - 2) 테스트 코드

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 위의 코드는 각 테스트 method가 실행되기 전에 초기화를 진행해주는 setup 함수이다. Junit 4 version까지는 @before decorator을 통해 setup 함수임을 알려주었지만 Junit 5 version부턴 @BeforeEach라는 표현을 통해 각 method가 call되기 전에 setup이 진행된다는 것을 더 명확하게 알 수 있게 해준다. Setup method에선 각 판 별로 적용되는 규칙을 불러오기 위해 Interface를 선언해준다.
* getStartRoute의 경우 시작 시 밟고 있는 곳이 외곽 route인데 프로젝트 내에선 외곽 루트를 route 0로 표현한다.
* getStartPostion의 경우 시작 시 밟고 있는 위치의 point가 0이기 때문에 위와 같이 초기화해준다. 배열로 따지면 윷놀이 판의 좌표는 point[route][position]과 같은 형태로 표현할 수 있다. Row에 해당하는 것은 외곽 루트인지 아니면 중앙 루트인지를 나타낸다.
* getRouteLength의 경우 외곽 route를 기준으로 도착 지점까지의 거리이다. Junit test에선 이를 간단하게 하기 위해 길이를 10으로 설정했다. Piece가 움직인 거리가 10보다 커지거나 같으면 도착으로 취급한다.
* 경로 변경에 대한 함수 부분에서 Junit test 초기값으로 length를 10으로 주었기에 route 분기 없음을 나타내었다.
* 백도가 좌표 값이 0인 곳에서 나타났을 경우 다시 0으로 바꾸어 주기 위해 adjustForBackdo 함수도 선언해주었다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 첫번째 test method는 게임 setting 화면에서 인자를 받을 때 Player 인자가 Interface와 잘 연결되고 ID 및 Player가 소지하고 있는 piece의 개수가 잘 배정되는지 알아보는 method이다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 두번째 test method는 piece가 잘 생성되는 지를 알아보는 method이다. 게임 setting 화면에서 handler가 create 함수를 call하는 데 성공했는지를 알려주고 생성한 piece의 개수가 조건과 맞는 지 비교를 통해 piece 생성의 에러 여부를 찾는 method이다. (사진 크기 이슈로 다음 페이지로 넘어가서 다음 method를 설명한다.)

텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 3번째 test method는 움직임에 대한 test이다. 가장 중요한 test로 말이 잘 움직이는 지 판단하는 code이다.
* 우선 Player을 1명, piece를 1개를 생성한다. 판의 모양은 움직임 자체에 영향을 주지 않기에 default로 선택된 사각형 모양의 판이 형성될 것이다.
* 초기 route와 post는 모두 0이다. movePieceAt의 경우 route, pos를 move 값에 따라 조정하는 함수이다.
* ‘걸’이 나오면 +4 칸을 움직이게 되고 초기 조건에 따라 route = 0, pos = 4가 될 것이라는 것을 예상할 수 있다. 이후 백도가 나오게 된다면 route = 0, pos = 3이 될 것이라는 것을 예상할 수 있다. 만약 movePieceAt method가 작동이 잘 안된다면 위의 예상 값과 실제 값이 다르게 나와 False를 return할 것이다.

텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 윷놀이에선 자신의 말 위에 다른 자신의 말을 겹칠 수 있다. 이를 grouping이라고 한다. Project에서의 grouping 동작은 덧셈 연산을 통해 이루어진다.
* getPieces.size()는 나가 있는 piece 차지하고 있는 크기이다. 만약 board 위에 piece위 좌표가 다르다면 이 method의 return 값은 나가 있는 piece의 개수가 된다.
* Grouping이 된 경우 나가 있는 piece의 좌표가 하나로 특정되므로 return 값이 1이 된다. Piece는 각각의 고유 piece\_ID를 배정받는다. 위의 시나리오에선 0번 piece위에 1번 piece가 도착하는 경우이다. 이 경우 piece\_ID = 0인 piece의 point는 자신의 point + 다른 말의 point (grouping도 고려)가 된다. 자신의 point는 default로 1이 배정되어 있고 다른 piece도 마찬가지로 1이 배정되어 있다. 따라서 예상 return 값은 2가 된다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 이 test method는 상대 방의 piece를 잘 잡는지 test하는 method이다. 우선 player을 2명을 생성한다. 각각의 player은 2개의 piece를 가지고 있다.
* 자신의 piece가 상대의 piece를 capture한다는 것은 attack하는 piece가 defend하는 piece의 좌표 위로 가는 것과 동일하다. 이 경우 움직임을 따로 구현하는 것보단 attack 입장에서 defend 입장의 좌표로 setting하는 것이 test 구조가 간단하게 된다.
* 이 test에서 주목해야 하는 것은 잡힌 입장에서의 piece의 size 그리고 대기하고 있는 piece의 개수의 갱신이다. Defend의 piece 하나가 board위에 있었고 결국 잡혔기에 예상되는 defend 측의 piece size는 0이 된다. 초기 값으로 piece를 2개 배정받았기에 다시 대기하는 piece의 개수도 2개 될 것으로 예상할 수 있다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 이 test method는 승리 조건이 잘 작동하는 지 알아보는 test method이다. Player을 1명 생성하고 piece는 1개 주어진다. 게임은 5각형 board 위에서 진행한다고 가정한다.
* Test의 시나리오는 도착 지점에서 한 칸을 움직였을 때 바로 승리 조건을 만족하는 지 판단하는 것이다.
* Piece가 도착 지점에서 한 칸을 이동한 후 player의 score가 1이 증가되었는지 확인하고 didSomeoneWin method를 call하여 승리자를 판별하게 된다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 마지막 test는 각 모양별 board가 잘 생성되는 지이다. Board를 생성할 때 parameter을 3개를 넘긴다. Parameter가 잘 넘어갔으면 생성이 되었을 것이고 그렇지 않은 경우 할당을 실패하여 Null이 될 것이다. 이러한 시나리오를 고려하여 assserNotNull(board) method를 이용하였다.

E - 3) 테스트 진행과정

* 모든 test method가 실행되기 전엔 @BeforeEach decorator로 지정된 setup method를 거치게 된다.
* 다음 페이지에 Junit test에 대한 Sequence diagram이 있다.

텍스트, 영수증, 번호, 문서이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

E - 4) 테스트 결과

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* Test 결과 모든 Junit test에서 예외 없이 성공했음을 볼 수 있다.

Test 시간은 443ms이고 test 중에서 가장 오래 걸린 것은 동적 할당이 필요한 board creation이다. 여기서 의문인 것이 왜 사각형 board가 가장 오래 걸리는 것일 것이다. 그 이유는 사각형 원판의 경우 Design (코드 디자인이 아닌 외형적 디자인)에 가장 많이 신경을 썼고 img 파일을 많이 불러오기 때문에 가장 오래 걸린 것이다.

E - 5) Junit version 5에 대하여

* Junit test를 진행할 때 test package의 version을 꼭 알아야 한다.
* 진행한 Junit test의 경우 수업시간에 나온 decorator인 @Before이 아닌 Junit test 5 version의 표현 방식인 @BeforeEach를 사용한다.
* 이전 버전과는 다르게 Junit 5는 Jupiter 노트북의 package를 import해야한다.
* 기존의 Junit 4 버전과 5 버전의 코드를 혼용하여 작성하고 싶은 경우 junit-vintage-engine 의존성을 코드 파일에 추가적으로 migration 해야 한다.
* 다음 페이지는 Notion에 정리한 Junit 4과 Junit 5 버전의 큰 차이점에 대해 정리한 내용이다.

텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 위와 같이 Annotation의 경우 더 자세하게 바뀌었다.

텍스트, 폰트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* Junit 5의 구성 요소는 위와 같이 크게 세 가지로 나뉘게 된다.
* 또한 Junit 5의 경우 Back-End 개발자의 편의성을 고려해 Gradle, Maven builder을 통해 다양한 assertion, Tag filtering, Parameterized test를 지원한다.
* Junit 5를 사용할 때 builder가 Spring boot 기반이 아니더라도 일반적인 pure Java에서도 돌아간다.

F) Github Report

**F - 1) GitHub 및 Git 사용 방안**

* 프로젝트 주소: https://github.com/BigBeautifulM/SETermProject18
* GitHub 및 Git 사용 경험이 적은 팀원을 위한 사용 방안 및 문제 해결 방안을 설명

**F– 1 - 1) GitHub & Git 사용 방법: Local에 Workspace 생성**

* 생성된 GitHub 저장소를 Local에 clone하여 사용해야 한다.
* 우리의 프로젝트를 예를 들어 설명한다. local에서 작업할 폴더를 생성 후 terminal을 켜준다.
* ‘git clone “<https://github.com/BigBeautifulM/SETermProject18>"’을 입력
* 로컬에 Workspace가 생성된다.

**F – 1 - 2) GitHub & Git 사용 방법: Pull request 관리 및 에러 핸들링**

* 로컬에서 master branch가 아닌 자기가 만든 branch인 경우 push 이후 master branch와의 merge가 필요하다.



* GitHub의 Repository를 들어가면 상단에 ‘Pull requests’라는 옵션이 보일 것이다.



* 위와 같이 New pull request를 실행하여 merge를 진행하면 된다.
* 보통의 경우 local에서 upstream push를 하게 될 경우 repository를 접속하자 마자 pull request 제안이 보이게 된다.
* Merge를 할 때 Conflict 메시지가 뜨면서 병합이 안되는 경우가 있다.
* 이는 Master branch에서의 변경 사항이 병합하려는 branch에 적용되지 않았기 때문이다.
* 현재 GitHub는 온라인 상에서 Conflict 코드 부분을 알려주고 GitHub 내에서 코드를 수정함으로써 병합을 할 수 있게 도와준다.

**F – 1 - 3) GitHub & Git 사용 방법: Conflict의 사전 예방 방지**

* Master branch가 갱신된다면 항상 ‘git pull origin master’을 통해 최신화를 진행해야 한다.
* Master branch를 pull하고 현재 작업중인 branch와의 병합도 필요하다.
* Git checkout [브랜치 이름] 혹은 git switch [브랜치 이름]를 이용해 작업 중인 branch로 분기한다. 이후 git merge master을 통해 master에 적용된 최신 사항을 작업중인 branch로 병합해야 한다.
* 만약 다른 사람이 작성한 remote branch를 다른 팀원이 수정하고자 한다면 git pull -b [브랜치 이름] origin/[브랜치 이름]을 통해 해당 branch만 local로 pull하여 작업을 진행해야한다.

**F – 2) Intelij dev 환경에서의 효율적인 Git 사용 방안**

텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* Intelij 하단 좌측의 Terminal icon을 누르면 위와 같이 Intelij 내에서 terminal을 조작할 수 있다.
* 실시간으로 업데이트 사항을 조원과 소통하며 branch 관리 및 병합 작업을 수행하면 능률을 높일 수 있다.
* Windows의 경우 기본적으로 PowerShell로 연결되며 CMD, Linux Kernel도 연결하여 프로그래밍을 진행할 수 있다.
* 그 밖에도 Setting-Plug in에서 Git, GitHub 관련 기본 setting을 조절하여 프로젝트를 진행하는 데에 도움을 받을 수 있다.

텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 일반적인 Terminal과 다르게 Intelij는 상단의 directory 선택창의 우측을 보면 git 및 github관련 명령어 집합이 버튼으로 mapping되어 있다.
* Commit 로그를 바로 작성할 수 있으며 Branch 생성, Push, Branch 제거도 가능하다.
* Intelij의 프로젝트의 경우 가끔 git pull을 통해 최신화를 했더라도 갱신 사항이 바로 적용되지가 않는 경우가 있다. 이때 위의 Update Project를 통해 해결할 수 있다.

F - 3) GitHub

F – 3 - 1) Directory 구조

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 프로젝트의 directory 구조이다. Doc은 프로젝터에서 사용했던 문서가 들어있다. Project directory는 java Swing으로 구성한 프로젝트 코드가 있고 projectjavafx directory는 javaFX로 구성한 프로젝트 코드가 있다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* Doc 폴더의 하위 폴더 구조이다. Diagram, 테스트 시나리오, 개요, 회의 내용 및 발표문 그리고 2차 과제 제출 때 사용할 문서를 나누어 저장한다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* Project 폴더의 하위 폴더 구조이다. Src directory에는 실행하는 코드가 들어있고 test directory에는 Junit test에 사용한 test suite code가 들어있다. (javaFX도 동일)

F – 3 - 2) Merge 및 commit 관리

* 팀원들의 GitHub 사용 경험을 고려하여 GitHub 사용 규칙 및 Commit 규칙을 정하였다.
* Merge를 할 때에는 ‘김찬중’ 학생을 참고인으로 Pull Request를 생성한다. 생성 시 Discord와 같은 연락 도구를 사용하여 Pull Request 요청 사실을 모든 조원에게 전송한다.
* 디버깅이 완료되고 문제가 없을 시 master branch에 병합을 진행한다.
* Commit의 경우 message 작성에 대해 규칙을 정하였다.
* Commit log는 다른 작성자가 알아볼 수 있도록 유의미한 문장으로 작성되어야 한다. 예를 들어 프로젝트 내의 육각형 모양의 판의 UI를 수정한 경우 ‘육각형 판 UI 수정’과 같이 다른 팀원들이 한 눈에 알아볼 수 있도록 수정사항을 메시지에 드러내야 한다.
* Commit은 사소한 부분이 생기더라도 message로 남겨야 한다. Commit history를 통해 에러가 발생한 부분을 예측할 수 있고 복구를 진행할 수 있기 때문이다.

스크린샷, 도표, 텍스트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 이번 프로젝트의 Code Frequency이다. Commit을 통해 프로젝트의 변경 사항의 종류를 나타낸 것으로 Addition이 대부분을 차지하고 있다. Delete의 경우 Debug를 위해 작성한 console print용 코드 및 refactoring이 대부분을 차지한다.

F - 4) Git

F – 4 - 1) Git Branch 전략 및 기여

* 우리의 project의 역할 분담은 UI, Logic, 기획 & Document로 크게 세 부분으로 나눌 수 있다. 모든 프로젝트의 진행사항을 master branch 상에서만 진행한다면 자신의 진행한 부분을 다른 사람이 잘못 수정하거나, 자신의 프로젝트 수행 정도를 잘 인식하지 못하는 경우가 있다.
* 따라서 각 인원별로 branch를 생성하여 프로젝트를 진행한다.
* UI의 경우 자신이 맡은 UI 부분에 대한 이름을 이용하여 branch를 생성한다.



* Branch의 이름을 결정할 때에는 [수행 파트][개발 or 디버그]와 같이 자신이 맡은 역할 중 어느 부분을 진행한지의 여부를 파악할 수 있게 해야 한다.
* Logic 부분의 경우 branch 이름을 결정할 때 [logic 이름][개발 or 디버그]와 같은 형태로 최대한 branch 생성을 유도하였다.

텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 지난 1달 동안의 기여 정도를 나타내는 graph이다. (pull request 빈도는 태아카, 김찬중, 이정우, 엄태형, 박민용 순) 총 28개의 pull request가 있었으며 약 86번의 commit이 있었다. 제일 많았던 pull request의 경우 로직에서 논리적 좌표를 실제 좌표와 대조하는 부분의 에러가 많아 Debug 횟수가 많았다. 이와 같이 Debug를 여러 번 진행할 때 하나의 branch에서 모든 것을 진행하는 것보단 각 Debug 부분을 소분할 하여 Merge Request를 진행함으로써 Conflict 가능성을 최대한 낮추었다.

텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 태아카 학생의 기여 정보이다. 주로 Logic에 대한 코드를 작성하였으며 오각형, 화면 분기 처리를 담당하였다. Junit test의 시나리오를 제공받아 코드로 표현하기도 하였다. 디버그 요청을 팀원한테 받으면 태아카 학생이 주로 이를 맡았으며 commit의 대부분이 오류 수정이었다.

텍스트, 스크린샷, 멀티미디어 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 김찬중 학생의 기여 정보이다. 프로젝트 managing (기획, 코드 흐름), 문서 총괄 (파트별 보고서, UML, 발표자료), Github pull request 관리, 디버깅을 담당했다. Junit 시나리오, 발표 시나리오와 같이 문서에 대한 총괄 및 코드 품질 개선을 위한 역할을 담당했다 5월 초순의 경우 많은 pull request 및 코드 에러가 있었기에 다른 학생들의 코드를 점검 및 디버깅을 했기에 commit 공백이 있다.

텍스트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 이정우 학생의 기여 정보이다. 프로젝트 초창기의 Swing UI의 대부분을 맡았다. 윷놀이 화면에 대한 배치, 버튼 생성에 대한 작업을 주로 맡았다.

텍스트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 박민용 학생의 기여 정보이다. 태아카 학생과 같이 게임의 logic을 구현하였다. 사각형, 육각형, 말의 분기에 관한 것을 처리했으며 게임 phase 쪽 코드를 담당하였다.

텍스트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 마지막으로 엄태형 학생의 기여 정보이다. Swing UI의 setting 화면, end page의 UI를 담당했으며 프로젝트 2차 제출에 사용할 JavaFX 관련 코딩을 전담하였다.

스크린샷, 멀티미디어 소프트웨어, 그래픽 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 상단의 사진은 실제로 진행한 Branch 합병 및 생성에 대한 work flow 일부이다.
* Conflict를 최소화하기 위해선 master branch에서 분기하여 자신의 branch를 생성하는 것이다.
* 위의 경우 한 명의 팀원이 하나의 task를 수행하면 merge를 통해 master에 병합을 진행했다. 이후 다른 branch에서 작업하던 팀원들은 pull request 소식을 통해 master branch를 갱신하였고 local branch에 merge함으로써 Conflict를 회피할 수 있었다.
* 코드 외의 documentation의 경우 doc branch를 통해 관리 감독하였다. 원격 Repository (즉 GitHub)에선 doc을 merge한 후 삭제를 항상 진행하였다. (다음 페이지 참고)

텍스트, 스크린샷, 폰트, 도표이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* ‘push -u’는 upstream push로 remote branch에 없는 local branch를 push하기 위한 명령이다. 위의 경우 GitHub 내의 branch에 doc branch가 없다는 것이 전제가 되어 진행되는 process인 것을 알 수 있다.

F - 4 – 2) Backup 전략

* 프로젝트를 진행하다 보면 불의의 사고로 인해 데이터가 이상하게 수정되거나 삭제되는 경우가 있다. 이러한 불상사를 미연에 방지하기 위한 규칙이 있다.

텍스트, 스크린샷, 폰트, 도표이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

* 위의 사진은 프로젝트에서 backup을 어떻게 진행하였는지를 도식화한 사진이다.
* Local에서 update가 발생하여 remote repository의 변화를 발생시켰다고 가정하자. (이때의 가정은 update가 local 환경에서 완벽하게 돌아가는 경우를 상정한다.) 팀원 중 한 명(김찬중; 기획, GitHub, 문서 담당)이 local에 backup branch를 생성한다. Master branch를 local에도 update를 적용한 후 이를 backup branch에 merge 시킨다. 이후 다음 update까지는 backup branch를 격리시킨다.
* 위와 같은 과정은 다른 팀원이 실수로 master branch에서 작업을 하여 모든 branch와의 conflict가 발생했을 때 hard하게 복구할 수 있게 해준다.
* Git의 사용을 능수능란하게 할 수 있다면 stash나 commit log를 이용한 history backup을 할 수 있지만 팀원의 Git 사용 여부를 고려한 결과 위와 같은 방법이 안전할 것 같다고 생각했다.

3. Conclusion

G - 1) 조원 정보 (역할 배분)

|  |  |
| --- | --- |
| **이름** | **역할** |
| **김찬중 (20213780)** | **Project Manager:**  **GitHub 관리, 프로젝트 기획 및 디버깅, 문서화 및 발표** |
| **박민용 (20213113)** | **Logic 구현 (말의 움직임 및 좌표 매핑)** |
| **태아카 (20234483)** | **Logic 구현 (그 외의 logic) 및 Junit test** |
| **엄태형 (20202029)** | **UI 구현 (메인 화면(Java Swing), JavaFX 담당)** |
| **이정우 (20202021)** | **UI 구현 (게임 판, Java Swing 담당)** |

G - 2) 프로젝트에서 사용했던 Tool 및 Site

* Github: <https://github.com/BigBeautifulM/SETermProject18>
* Git: <https://git-scm.com/>
* Gluon (JavaFX JDK archive): <https://gluonhq.com/products/javafx/>
* JavaFX 공식 문서: <https://openjfx.io/>
* UML tool #1 (draw.io)
* UML tool #2 (Plant UML) 공식문서: <https://plantuml.com/ko/>
* Intelij: <https://www.jetbrains.com/idea/>

G - 3) 프로젝트 회고 (PM 관점, 김찬중)

* 팀원들의 Github, Git, Java 사용 경험이많지 않았기에 두려움이 많았었지만 결국 프로젝트를 완수할 수 있었다는 점에서 다행이라 생각한다.
* 팀원들이 1달 동안 여러 사람끼리 협업한 경험이 없었지만 협업하는 방법을 빠르게 숙지하여 critical한 conflict 문제가 발생하지 않은 것이 신기하고도 다행인 것 같다.
* Unit test를 설계하면서 프로젝트 코드의 이해도가 높아진 것 같고 Debug 시간의 단축을 몸소 느낄 수 있었다. 다른 프로젝트를 이용할 때 TDD와 같이 test를 이용해 debug 난이도를 낮춰보면서 프로젝트를 진행해보고 싶다.
* Java Swing을 코딩할 때 워낙 legacy한 느낌을 받아 어려울 것 같았지만 생각보다 쉽게 코딩할 수 있었고 JavaFX의 경우 CSS나 HTML을 이용한 프로젝트를 진행해본 경험이 없어 컴포넌트 배치와 같은 UI관련 issue가 많았던 것 같다.
* Model의 코드를 수정하지 않더라도 UI-layer의 코드가 생각보다 많이 바뀌는 것을 보고 model의 독립성이 얼마나 중요한지 깨달을 수 있었다. (UI마다 다른 규약이 있으며 규약에 의해 controller의 코드가 상당부분 바뀌는 것을 보고 model의 독립성이 깨지면 프로젝트 코드를 0부터 다 바꾸어야 할 것 같다는 생각을 몸소 느끼게 되었다.)
* 프로젝트의 IDE 구성을 할 때 많은 시행착오가 있었지만 공식문서를 발췌하면서 프로젝트 관리 능력이 상당히 올라간 것을 느꼈다.

**G - 4) 프로젝트 회고 (팀원 관점)**

* **박민용**: 이번 팀프로젝트를 하면서 MVC 아키텍처 패턴에 대해 더 자세하게 알 수 있게 되었습니다. 또한 프로그램을 만들 때 어떤 식으로 설계를 해야 하는지, 테스트는 어떤 식으로 해야 하는지, Github를 어떤 식으로 같이 사용하는지 등을 배울 수 있었습니다.
* **태아카**: 이번 팀 프로젝트를 통해, 개발보다 초기 문서화 및 계획 수립의 중요성을 깊이 실감할 수 있었습니다. 기능별로 작업을 분담하는 방식이었기 때문에, 각자 맡은 부분을 어떻게 구현할지 명확히 documentation 해두는 것이 필수적이었습니다. 이를 통해 각 모듈을 통합할 때 발생할 수 있는 디버깅 작업을 최소화할 수 있었습니다. 또한, 팀원 들과의 의사소통과 회의도 프로젝트의 중요한 부분이라는 것을 다시 한번 느꼈습니다. 개발만이 전부가 아니라, 기획, 문서화, 협업 등 전반적인 소통 과정이 프로젝트의 완성도를 높이는 핵심이라는 점을 배울 수 있었습니다.
* **엄태형**: 새로운 Ui를 이식하는 작업을 전적으로 맡은 첫 프로젝트였습니다. 이식 과정에서 정돈된 소프트웨어 구조와 기능 분할이 업무 부담을 크게 줄여주었고, 의존성과 관련된 문제를 크게 줄여주어 디버깅 과정의 효율을 크게 늘려주었습니다. 또한 적절한 소프트웨어 기능 분할이 체계적인 분업에 크게 도움을 주기도 했습니다. 빠르고 효율적인 작업을 가능케 해주었습니다. 체계적인 소프트웨어 구조의 중요성을 실감할 수 있었던 프로젝트였습니다.
* **이정우**: 여러 명이서 하는 프로젝트 경험이 적어서 이번 프로젝트는 저에게 더 의미가 있었던 것 같습니다. 특히, 이번 프로젝트를 진행하면서 협업의 어려움과, 지속적이고 정확한 피드백의 중요성을 깨닫았습니다. 앞으로는 역할 분담 이후에도 서로의 진행 상황을 자주 공유하며, 전체 흐름을 놓치지 않도록 노력할 생각입니다.