SE 2025-1 Term Project report

Team 17

1. 프로젝트 개요

본 프로젝트는 2025년 봄학기 객체지향 분석 및 설계(OOAD) 수업의 텀프로젝트로, 전통 윷놀이 게임을 디지털 환경에서 구현하는 소프트웨어 개발 과제이다. 주어진 제약사항과 평가 기준을 충족하면서, 다양한 UI 툴킷을 활용하고 커스터마이징 가능한 윷놀이 시스템을 구축하는 것이 핵심 목표다.

-프로젝트 목표

1. 객체지향 분석 및 설계 기법을 적용하여 유지보수 가능하고 유연한 윷놀이 게임을 구현한다.
2. MVC 아키텍처를 기반으로 UI와 비즈니스 로직을 분리하여 코드의 재사용성과 확장성을 높인다.
3. 다양한 UI 프레임워크(Swing, JavaFX 등)를 통한 다중 UI 구현을 통해 인터페이스 독립성을 확보한다.
4. 테스트 가능한 구조를 설계하고 JUnit 기반 테스트 케이스를 통해 기능의 정확성과 안정성을 검증한다.

- 주요 요구사항

1. 게임 시작 시 참여자 수(2~4명) 및 말 수(2~5개) 지정 가능, 개인전
2. 윷 던지기 방식: 랜덤 / 지정 선택 가능
3. 사용자 선택 기반의 말 이동 처리
4. 말 업기, 상대 말 잡기 기능 포함
5. 완주 시 승리 처리 및 게임 재시작/종료 기능
6. 윷판 커스터마이징 기능: 사각형, 오각형, 육각형 지원

- 세부 규칙

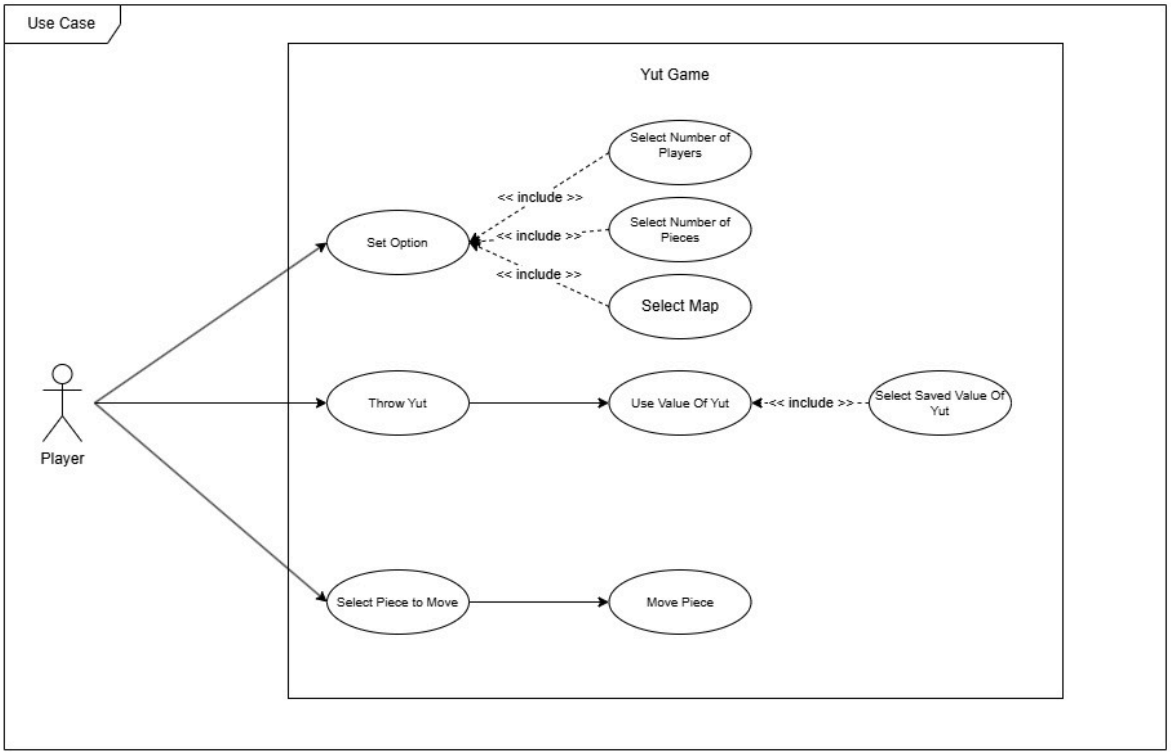
윷놀이 세부적인 규칙에 따라 진행방식이 달라지기 때문에 세부적인 규칙을 추가한다.

1. 윷이나 모로 상대 말을 잡았을 때 다시 던질 수 있는 횟수가 2회 증가한다.

2. 빽도는 판 위에 있는 말에게만 적용할 수 있다.
3. 교차점에서는 무조건 지름길만 갈 수 있다.
4. 교차점에서 빽도가 나오면 무조건 외곽으로 가고 중앙에서 빽도가 나오면 왼쪽 교차로로 간다.
5. 말이 빽도로 원점을 갔을 시 다음에 1칸 이상 가면 말이 탈출한 것으로 한다.

2. Use-case model

- Use-case Diagram



윷놀이 게임의 핵심 기능은 게임 초기 설정 (플레이어 수, 말의 개수, 윷판 형태)을 선택하고, 윷을 던진 결과에 따라 말을 이동시키는 것이다. 위에 있는 Use case diagram은 이러한 필수 기능들을 중심으로 모델링했다.

- Use-case Text

<Use Case 1>

Use Case Name	윷놀이 게임 설정
Description	윷놀이를 하는 Player수와 말의 개수, 윷판의 모양을 설정한다.
Actor	Player
Precondition	윷놀이 게임이 실행된 상태이다.

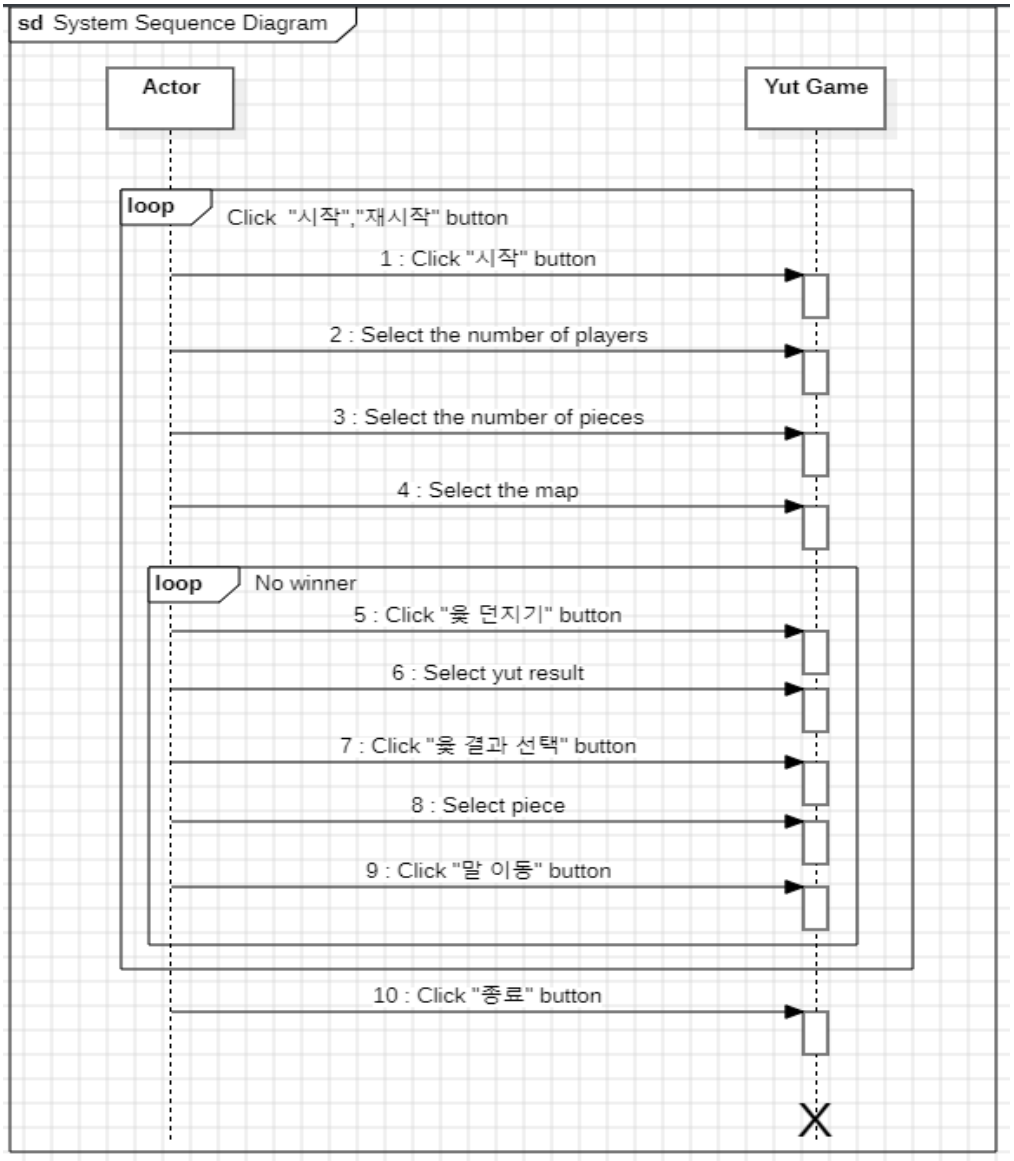
Main Flow	1. 플레이어의 수를 정한다. 2. 말의 개수를 정한다. 3. 윷판의 모양을 선택한다.
Postcondtion	선택된 설정 값이 시스템에 저장된다.
Includes	플레이어의 수는 2~4명 사이로 한다. 말의 개수는 2~5개로 한다. 윷판의 모양은 사각형, 오각형, 육각형이 있다.

<Use Case 2>

Use Case Name	윷놀이 게임 진행
Description	첫 번째 플레이어부터 승자가 나올 때까지 게임을 진행한다.
Actor	Player
Precondition	윷놀이 게임 설정이 다 완료된 후 게임이 정상 시작된 상태
Main Flow	1. 차례인 플레이어가 윷을 던진다. 2. 윷의 결과를 선택한다. 3. 말을 선택한다. 4. 나온 윷의 결과만큼 선택한 말을 옮긴다. 5. 다음 차례인 플레이어에게 턴을 넘긴다.
Alternative Flow	2-1 윷의 결과 빽도이고 판 위에 말이 없을 경우 차례를 넘긴다. 2-2 말 이동을 누를 경우 윷의 결과를 선택하지 않았다고 알려준다. 4-1 현재 플레이어의 모든 말이 결승점을 통과한 경우 현재 플레이어의 승리인 것을 알려주고 게임을 재시작/종료 할 수 있게 한다.
Postcondition	
Includes	1. 윷의 결과가 윷이나 모일 경우 한번 더 던질 수 있다. 2. 내 말이 이동한 최종 위치에 상대방 말이 있을 경우 상대방의 말을 잡아 상대방의 말을 판에서 제거하고 윷을 한번 더 던진다. 3. 내 말이 이동한 최종 위치에 나의 다른 말이 있을 경우 말을 업어서 같이 이동할 수 있다. 4. 말의 처음 위치가 갈림길 위치인 경우 무조건 갈림길로 이동한다. 5. 도에서 빽도가 나와 원점에 위치한 후 앞으로 이동하면 한 바퀴 돌지 않고 탈출한 것으로 한다. 6. 갈림길에서 빽도가 나오면 외곽으로 이동한다. 7. 중앙 위치에서 빽도가 나오면 왼쪽 대각선으로 이동한다. 8. 빽도는 판 위에 있는 말에만 적용할 수 있다.

3. SSD, Operation Contract

-System Sequence Diagram(SSD)



SSD에서는 Actor가 시스템과 상호작용할 수 있는 유일한 수단은 UI에 있는 버튼 클릭이다. Actor는 시스템의 내부 로직을 직접 제어할 수 없고, 버튼을 클릭하면 시스템은 이에 대응하는 기능을 수행한다. 이 SSD는 일어나는 모든 이벤트를 시각적으로 표현한 것이다. 사용되는 method는 Click “시작” button, Select the number of players, Select the number of pieces, Select the map, Click “윷 던지기” button, Select yut result + Click “윷 결과 선택” button, Select piece+Click “말 이동” button, Click “종료” button. 이 있다. 이 중 몇가지는 단순 UI 전환이거나 상태 변경이 없기 때문에 Operation Contract에 포함하지 않는다.

-Operation Contract

1) Select the map

Operation	selectMap(mapName: String)
Cross References	UI의 “맵 선택 화면”에서 적용된다.
Preconditions	MainApp.selectedPlayerCount, MainApp.selectedPieceCount가 유효한 값으로 설정되어있어야 한다.
Postconditions	1. 선택한 mapName에 따라 Board의 서브클래스가 생성된다. 2. 생성된 Board가 YutBoardPanel 생성자에 주입된다. 3. 플레이어 수에 따라 Player 객체 목록이 생성된다. 4. 각 Player는 선택된 말의 개수만큼 Piece를 생성한다. 5. GamePanel이 생성되어 화면에 등록된다.

2) Click “윷 던지기” button

Operation	throwYut()
Cross References	YutGamePanel에서 호출된다.
Preconditions	현재 턴의 Rule.getRemainigRolls() > 0
Postconditons	1. Yut.throwYutRand()를 통해 무작위 결과를 하나 생성한다. 2. Yut.setSelectedResult()로 해당 결과를 저장한다. 3. 결과가 윷/모인 경우 Rule.addRollChance()가 호출된다. 4. Rule.useRollChance()로 남은 던지기 횟수를 1 감소시킨다. 5. 결과가 UI에 표시된다.

3) Click “윷 결과 선택” button

Operation	useYutResult(result: String)
Cross References	YutGamePanel의 “선택한 결과 사용” 버튼
Preconditons	Yut.getResultList()에 해당 결과가 포함되어 있어야 한다. 해당 결과를 클릭했어야 한다.
Postconditons	1. 선택한 결과는 Yut.useResult()를 통해 목록에서 제거된다. 2. Rule.setDistance()에 해당 인덱스를 설정하여 이동 거리를 저장한다. 3. Yut.isAllUsed()가 true일 경우 Rule.markYutUsedUp() 호출한다. 버튼은 disable된다.

4) Click “말 이동” button

Operation	movePiece(selectedPiece: Piece)
Cross References	GameController 내부에서 호출된다.
Preconditions	Rule.getDistance()가 설정되어 있어야 한다. 선택된 Piece는 아직 종료되지 않았고, 이동 가능한 상태여야 한다.
Postcondtions	1. 이동 가능한 다음 노드를 계산하여 BoardNode를 변경한다. 2. 만약 적 말이 있다면 해당 말을 MoveResult.addCaptured()에 등록한다. 3. 도착 노드에 이동한 같은 팀 말들이 있을 경우 말을 합체한다. 4. 말이 탈출 조건을 만족하면 MoveResult.addEscaped()에 등록한다. 5. Piece.setHasMoved(true) 를 호출한다. 6. 이동 완료 후 Rule.markPieceMoved(true) 7. Rule.changePlayerIfTurnDone()에 의해 턴 교체 여부가 결정된다.

- 단순 **UI** 전환이거나 상태 변경이 없는 **method**

1) Click “시작” button: PlayerSelectPanel로 전환된다.
2) Select the number of players: 선택된 플레이어 수를 저장하고 PieceSelectPanel로 전환한다.
3) Select the number of pieces : 선택된 말 개수를 저장하고 MapSelectPanel로 전환한다.
4) Click “종료” button: 윷놀이 게임이 종료된다.

4. Design and Implementation Report

-Software Architecture

이 윷놀이 게임은 Model-View-Controller(MVC) 아키텍처를 기반으로 구성되어 있다. 사용자 인터페이스와 게임 로직을 분리해서 유지보수성과 재사용성을 높이고자 했고 UI를 교체해도 코드 변경을 최소화 하는 것을 목표로 설계되었다.

1) Model (게임 데이터/ 로직): Player, Piece, Board, BoardNode, Yut, Rule, MoveResult
2) View(UI 화면): GamePanel, YutGamePanel, MapSelectPanel, PlayerSelectPanel, PieceSelectPanel
3) Controller(게임 로직 조정자): GameController

Model

Model은 게임의 핵심 데이터를 표현하고, 게임 규칙과 상태를 처리하는 클래스들로 구성되어 있다.

1) Board
- 보드의 기본 구조와 경로 설정 로직을 담당하는 상위 클래스.

- 서브클래스로는 SquareBoard, PentagoneBoard, HexagonBoard가 있고, 사각형, 오각형, 육각형의 윷놀이 판을 구현한다.

-내부적으로 Map<String, BoardNode> 형태로 노드를 관리하고 노드 간의 연결 관계와 경로 설정을 수행한다.

2) BoardNode
-보드의 각 지점을 나타내는 클래스이다.

- 고유 식별자(id)와 연결된 다음 노드들을 포함한다.

3) Piece
-각 플레이어가 가진 말 객체를 의미한다.

-이동, 잡기, 업기같은 행동이 이곳에서 수행된다.

-현재 위치 (currentNode), 그룹핑된 말의 정보들을 관리한다.

4) Player
-게임에 참여하는 사람을 나타낸다. 이름, 색상, 보유 말 목록등을 저장한다.

-남은 말이 있는지, 없는지로 승리를 판단하는 함수를 포함한다.

5) Yut
-윷 던지기 결과를 생성하는 클래스이다. getRandomResult는 게임에서 윷의 결과를 랜덤으로 반환하는 역할을 한다.

6) Rule
-윷놀이의 다양한 규칙을 캡슐화한 클래스이다.

-말의 이동 가능 여부, 업기 가능 여부, 잡기 가능 여부등을 판단하는 함수를 포함한다.

7) MoveResult
-윷 던지기 결과에 따른 이동 정보를 저장하고 해석하는 클래스로 이동 거리, 추가 턴 여부, 멈춰야 하는 위치 등의 정보를 포함한다.

View

View는 유저와 상호작용을 UI를 담당하는 클래스들로 구성되어 있다. 사용자 입력을 받고, 게임 상태를 눈에 보이게 표현하는 역할을 담당한다.

1)GamePanel

-게임이 실행되는 메인 패널이다. 전체 게임 흐름을 포함한 화면 레이아웃을 구성한다. 내부에 윷놀이 판, 말 선택 창, 윷던지기 버튼등이 있고, 사용자와 상호작용하는 중심역할이다.

2)YutGamePanel

-실제 윷놀이 판을 시각적으로 보여주는 클래스이다. 보드의 노드 및 말의 위치, 이동 경로 등을 시각적으로 표현하고 Piece의 선택을 처리한다.

3) MapSelectPanel

- 게임 시작 전 윷판의 형태(사각형, 오각형, 육각형)을 선택할 수 있는 UI 화면이다.

4)PlayerSelectPanel

-게임 시작 시 참여자 수(2~4명)를 선택하는 UI 화면이다. 선택된 수 만큼 플레이어 객체들이 생성되고 각각 고유 색상과 이름을 할당받는다.

5) PieceSelctPanel

-각 플레이어가 사용할 말의 수(2~5)개를 선택하는 UI화면이다. 선택한 말의 개수만큼 각 플레이서의 말이 생성된다.

Controller

1)GameController

- GameController는 MVC 구조의 중심에서 모델과 View를 연결하는 핵심 클래스이다. 게임 진행을 관리하고 사용자의 입력에 따라 게임 상태를 변경하고 그 결과를 화면에 반영하는 역할을 한다.

주요 기능을 하나 씩 설명하면

a) 우선 턴관리가 있다. 각 플레이어의 차례를 관리하고 자동으로 다음 플레이어로 전환한다.

b) 윷 던지기 처리를 한다: 사용자가 버튼을 누르면 Yut 클래스를 통해 랜덤 결과 또는 지정 결과를 생성한다.

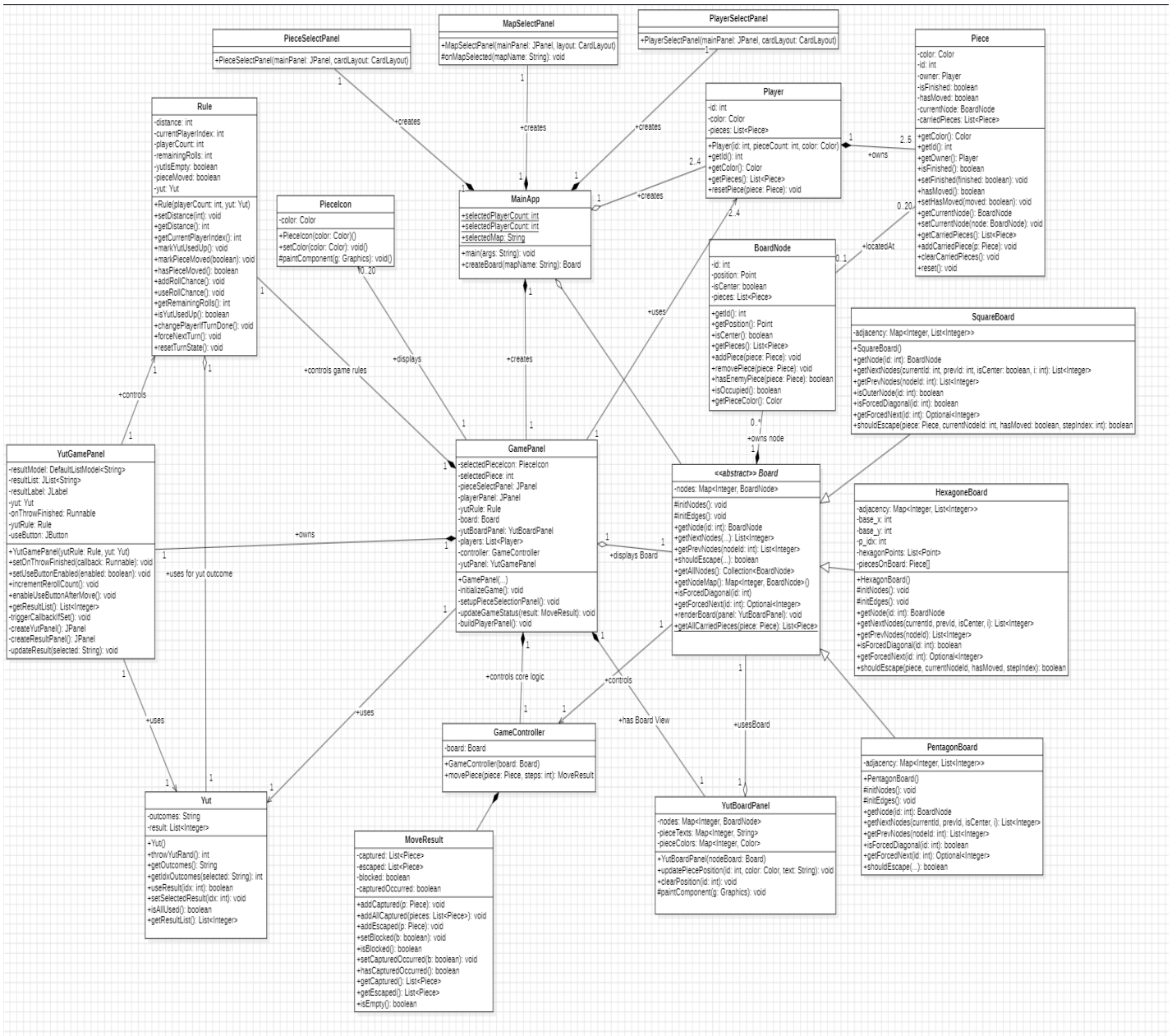
c) 말 선택 ,이동: MoveResult를 기반으로 사용자가 선택한 Piece의 다음 위치를 계산하고, Rule을 통해 업기 및 잡기 여부를 판단한다.

d) 게임 종료 판단: Player 클래스의 상태를 기반으로 모든 말이 완주한 플레이어가 있으면 승리 처리한다.

e) UI, 모델 연결: 예로는 MapSelectPanel에서 선택한 보드 타입에 따라 적절한 인스턴스를 생성하고 YutGamePanel에 적용한다.

이러한 책임 분리를 통해, 모델 로직은 UI에 의존하지 않도록 유지되며, 다른 UI 툴로 최소한 코드 변경으로 전환 가능해진다. 그리고 이것은 코드이 재사용성과 유지 보수성을 향상시킨다.

-Class Diagram

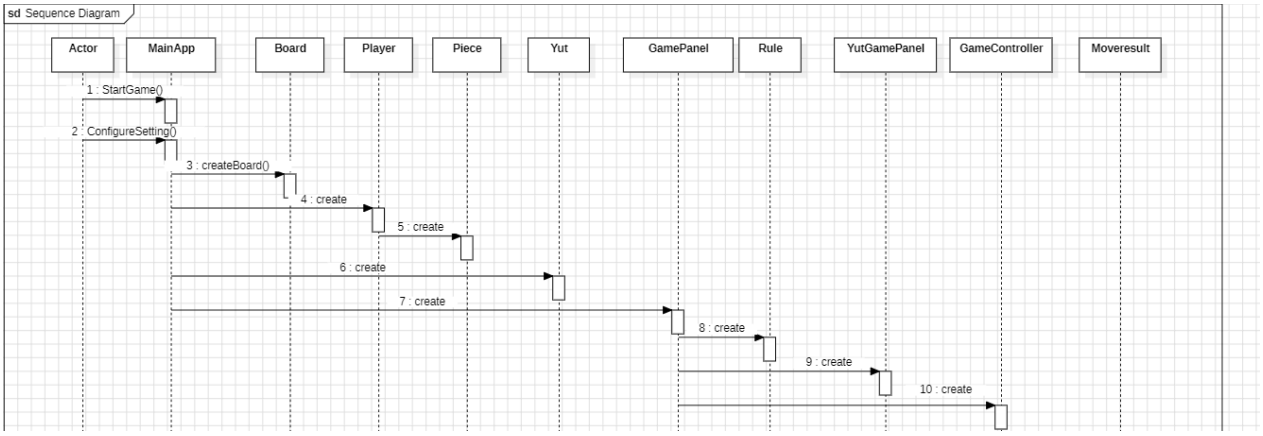


위에 클래스 다이어그램은 윷놀이 시스템의 구조를 보여준다. MainApp 전체 프로그램의 진입점으로, 게임 패널 및 설정 패널을 초기화하고 화면 전환한다. GamePanel은 UI의 중심 역할을 하며, 보드, 플레이어, 윷 관련 패널과 컨트롤러를 조립하여 관리한다. Player는 2~5개의 Piece를 보유하고, Piece는 현재 위치를 BoardNode를 통해 나타낸다. Board는

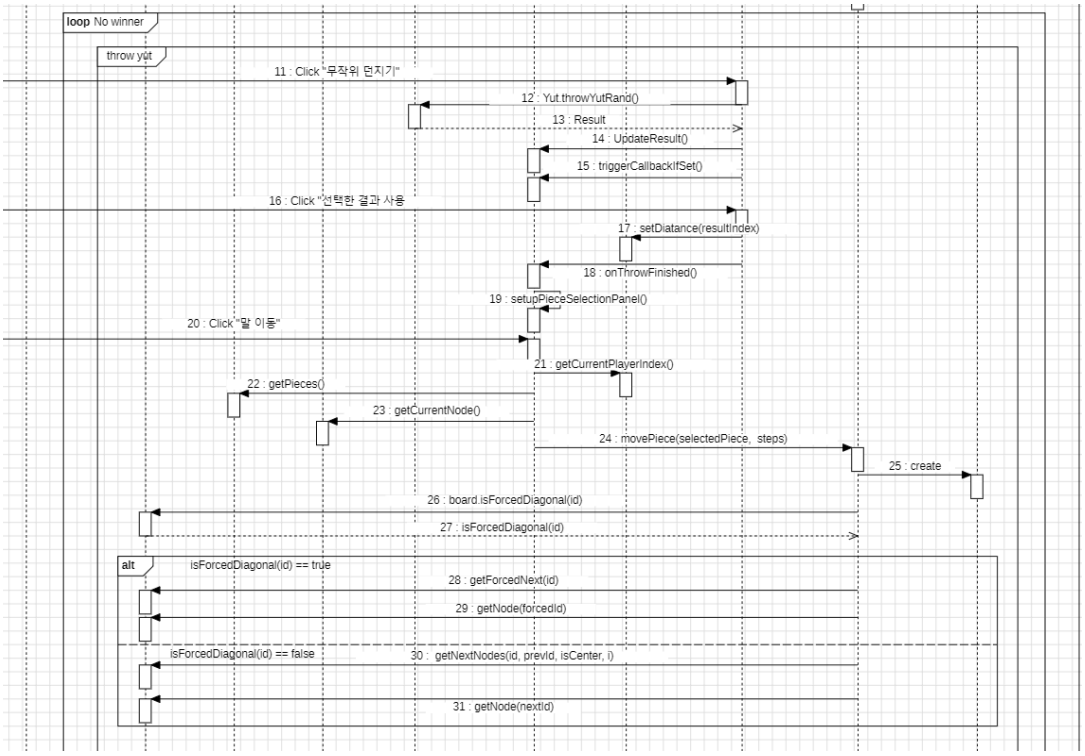
사각형, 오각형, 육각형으로 구현되고 BoardNode를 포함한다. YutGamePanel과 Yut은 윷 던지기 로직, 결과 처리를 담당하고, Rule은 턴, 이동 거리, 추가 기회 등 게임 상태를 관리한다. 말 이동결과는 GameController를 통해 처리되고, 이동 결과는 MoveResult 객체로 캡슐화 된다.

Class 간의 관계는 주로 Assocation, Aggregation, Composition으로 구성되어 있고, UI와 로직은 분리되어 있다.

Sequence Diagram

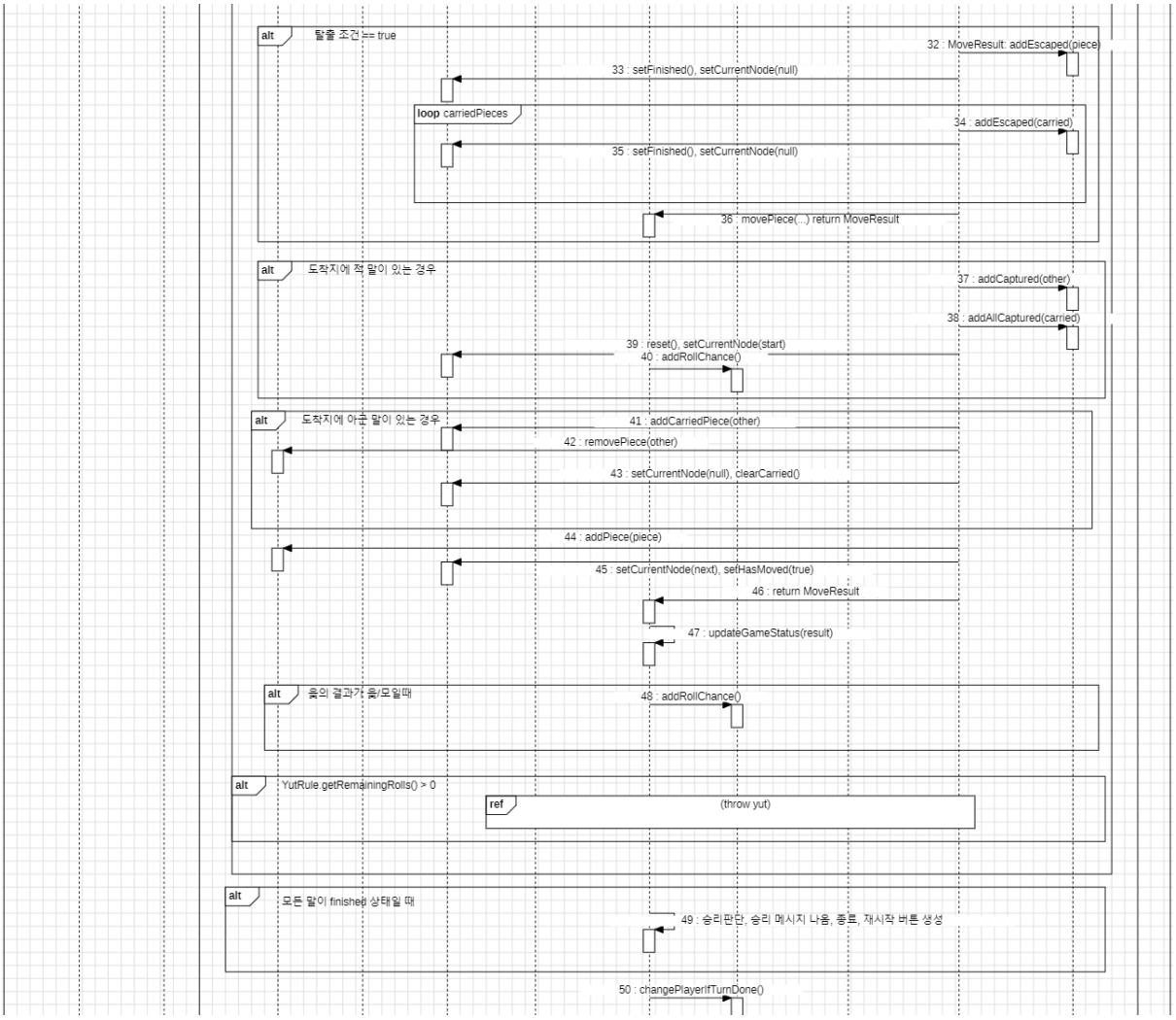


(1~10): startGame 후 configureSetting(플레이어 수, 말의 개수, 윷판 선택을 의미한다.) 완료하면 MainApp이 Board, Player, Yut, GamePanel을 만들고 Player는 Piece를 생성한다. 그 후에 GamePanel은 Rule, YutGamePanel, GameController를 생성하고 여기 까지 완료 되면 윷놀이 게임을 시작할 준비가 된 것이다.



(11~15): 게임이 시작하고 나서의 sequence다. 무작위 던지기를 누르면 YutGamePanel에서 Yut에 있는 throwYutRand()함수를 사용해 result를 가져오고 GamePanel에 윷 결과를 업데이트한다. 그 후에 선택한 결과를 사용하면 Rule이 setDistance()로 이동 거리 정보를 저장하고 triggerCallbackIfSet()을 통해 UI 업데이트를 트리거 한다.

(16~25): Actor가 말 이동 버튼을 누르면 현재 말에 대한 정보를 받아와서 GameController.movePiece()가 실행되고 Moveresult가 생성된다.
(26~27): Board에서 지름길 판단하는 isForcedDiagonal을 받아와서 true이면 지름길로 가고 false이면 지름길이 아닌 곳으로 가게 된다.



(32~36): 말이 탈출하는 조건 shouldEscape()를 가져와서 탈출 조건이 맞으면 탈출 시키고 안에 loop문은 업고 있는 말까지 탈출하는 구조이다.

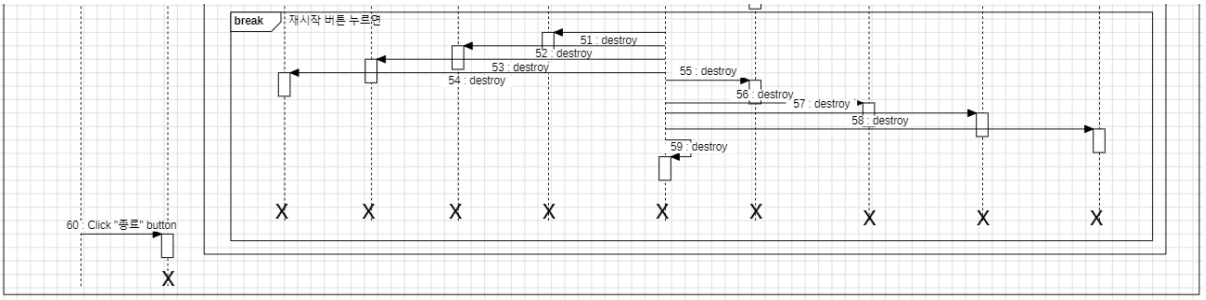
(37~40): GameController.movePiece(Piece piece, int steps) 이곳에서 최종 위치에 상대팀 말이 있다면 잡고 잡힌 말은 reset되고 윷 굴리는 기회를 한번 더 주게 된다.

(41~44): GameController.movePiece(Piece piece, int steps) 여기서 또한 최종 위치에 우리팀 말이 있으면 그 말은 업고 다닐 수 있게 된다.

(44~47): GamePanel.updateGameStatus()로 말의 위치를 업데이트하고 결과를 리턴한다.

48: 윷의 결과가 모/윷이면 윷 굴리는 기회를 한번 더 주는 것이다.

(11 ~ 48 밑에 alt문 까지)가 throw yut이고 굴릴 수 있는 기회가 남아있으면 throw yut을 반복하는 것이다.
49는 승리조건 판별이고 50은 상대에게 턴을 넘기게 되는 것이다.



loop문을 나갈 수 있는 조건은 게임이 끝나 재시작 버튼을 누르거나 종료 버튼을 누르면 생성됐던 객체들이 destroy 되면서 재시작인 경우에는 다시 mainApp이 실행되고 종료 시에는 꺼지게 된다.

Key Feature Implementations

1) 말 이동 처리

GameController.movePiece(Piece piece, int steps)에서 발생한다.
BoardNode current = piece.getCurrentNode(); 말의 현재 위치를 확인 한다. 그 후에 보드 객체를 통해 다음 노드들을 계산한다.

```
if (direction == 1 && board.shouldEscape(piece, id, piece.hasMoved(), count)) {
    System.out.printf("▩ 말%d 탈출 조건 충족: 위치 %d → 탈출 처리됨\n", piece.getId(), id);
    result.addEscaped(piece);
    for (Piece carried : getAllCarriedPieces(piece)) {
        result.addEscaped(carried);
        carried.setFinished(true);
        carried.setCurrentNode(null);
        System.out.printf("▩ 업혀있던 말%d도 함께 탈출 처리됨\n", carried.getId());
    }
    piece.setFinished(true);
    piece.setCurrentNode(null);
    return result;
}
```

이동 중, 윷놀이 규칙에 따라 탈출 조건 shouldEscape()를 만족하면, 해당 말과 업혀 있는 말들을 판에서 제거하고 MoveResult에 반영한다.

그 후에는 지름길 진입 여부를 판단하고, 해당하면 강제로 지름길 노드로 진입된다.

```
if (direction == 1) {
    if (i == 0 && board.isForcedDiagonal(id)) {
        Optional<Integer> forced = board.getForcedNext(id);
        if (forced.isPresent()) {
            int forcedId = forced.get();
            System.out.printf("▩ 강제 지름길 진입: %d → %d\n", id, forcedId);
            next = board.getNode(forcedId);
            prevId = id;
            continue;
        }
    }

    List<Integer> nextList = board.getNextNodes(id, prevId, board.getNode(id).isCenter(), i);

    if (nextList.isEmpty()) {
        System.out.println("▩ 다음 노드 없음: 이동 중단");
        return result;
    }

    next = board.getNode(nextList.get(0));
    prevId = id;
} else {
    List<Integer> prevList = board.getPrevNodes(id);
    if (prevList.isEmpty()) return result;
    next = board.getNode(prevList.get(0));
}
```

각 이동 단계마다 보드의 노드 간 연결 정보를 기반으로 다음 위치를 탐색하며 경로를 정한다

지름길 조건은 i==0& board.isForcedDiagonal 이 true여야 한다. 이 뜻은 시작하는 노드가 갈림길 노드인가를 의미한다. 만족하면 getForcedNext(id) 해당 노드에서 진입 가능한 지름길의 다음 노드 ID 반환하는 것이다.

아닐 경우는 getNextNodes 를 통해 일반 경로를 계산해서 가게 된다..

이동 중 최종 위치에 상대방의 말이나 같은 팀의 말이 있을 수 있다. 이런 경우일 때 잡기와 업기가 필요하다. 이것 또한 movePiece()에서 판단한다.

2) 말 잡기/ 업기

```
for (Piece other : new ArrayList<>(next.getPieces())) {
    if (!other.getOwner().equals(piece.getOwner()) && other.hasMoved() && !other.isFinished()) {
        System.out.printf("🎯 말%d이 적 말%d을 잡았습니다.\n", piece.getId(), other.getId());
        result.addCaptured(other);
        result.addAllCaptured(getAllCarriedPieces(other));
    }
}
```

말 잡는 조건은 다른 말이 존재 해야 한다. next.getPieces()는 도착할 위치에 이미 존재하는 말 리스트이다. 조건 !other.getOwner().equals(piece.getOwner() 로 말이 상대방이 말인 것이고 hasMoved() 이미 이동했던 말이어야 한다. !ifFinished() 탈출한 말이 아니어야 한다는 것이 조건이다. 이 조건에 다 만족하면 result.addCaptured()로 잡은 말을 기록하고 getAllCarriedPieces()로 업혀있던 말들 까지 전부 잡히게 된다. 이후 위치를 초기위치로 보내고 말의 상태를.reset()로 다 초기화 시킨다.

```
for (Piece captured : result.getCaptured()) {
    BoardNode node = captured.getCurrentNode();
    if (node != null) node.removePiece(captured);
    captured.reset();
    BoardNode startNode = board.getNode(0);
    startNode.addPiece(captured);
    captured.setCurrentNode(startNode);
}
```

말을 업게 되는 조건은 getOwner()가 같아야하고 이미 이동한 말에 탈출하지 않은 말이여야 한다. 업는 방식은 piece.addCarriedPiece(other)로 직접 업고 업고 있던 말도 함께 포함된다. 업힌 말은 위치가 null이 되고 carriedPieces list도 비워진다.

```
for (Piece other : new ArrayList<>(next.getPieces())) {
    if (other.getOwner().equals(piece.getOwner()) && other.hasMoved() && !other.isFinished()) {
        piece.addCarriedPiece(other);
        piece.getCarriedPieces().addAll(getAllCarriedPieces(other));
        next.removePiece(other);
        other.setCurrentNode(null);
        other.getCarriedPieces().clear();
        System.out.printf("🤝 말%d이 말%d을 업었습니다.\n", piece.getId(), other.getId());
    }
}

next.addPiece(piece);
piece.setCurrentNode(next);
piece.setHasMoved(true);
```

이러한 로직을 통해 윷놀이의 가장 중요한 기능인 업기, 잡기가 구현된다.

부가적인 구현 사항

1) 빽도 처리

```
if (!canMoveExists && isBackDoOnly && yutRule.getRemainingRolls() == 0) {
    JOptionPane.showMessageDialog(this, "적용할 수 있는 말이 없어 턴을 넘깁니다.");
    yutRule.forceNextTurn();
    yutPanel.setUseButtonEnabled(true);
    buildPlayerPanel();
    return;
}
```

빽도는 현재 판 위에 있는 말에만 적용할 수 있다는 규칙을 만들었기 때문에 만약 빽도만 있고 판 위에 움직일 수 있는 판이 없고 윷을 굴릴 수 있는 기회가 없다면 “적용할 수 있는 말이 없어 턴을 넘깁니다.”라는 메시지를 출력하고 강제로 다음사람 턴으로 넘기게 하였다.

UI toolkit 변경 시 변경 되는 부분

본 프로젝트는 JavaSwing을 기반으로 초기 UI를 구성했고, 이후 JavaFX로 UI Toolkit을 변경했다. 이 과정에서 View에 해당하는 Class GamePanel, MainApp, MapSelectPanel, PieceIcon, PieceSelectPanel, PlayerSelectPanel, YutBoardPanel, YutGamePanel 8개의 Class는 JavaFX와 Swing이 UI컴포넌트 구조 및 이벤트 처리 방식, 색상 설정 방식, 레이아웃 구성 등에서 차이가 있어서 JavaFX에 맞게 코드를 수정했다.

또한 'Point' 객체를 기반으로 좌표를 관리하던 기존 구조는 JavaFX에서 지원하지 않기 때문에 BoardNode, Board 클래스에서 좌표를 다루는 부분은 JavaFX에서 사용 가능하게 수정했다. 이로 인해 Board를 사용하는 일부 클래스들에도 간접적인 수정이 발생했다.

그러나 게임 로직이 포함된 부분과 UI, Board와 관련이 없는 부분은 UI와 무관하므로 수정되지 않았다.

5. Junit Test

Junit Test 가능한 구조로 설계되었으며, Junit5를 활용하여 총 6개의 핵심 기능에 대해 테스트를 수행했다.

```java
package YutYutGame;

import org.junit.jupiter.api.BeforeEach;


public class YutYutGameJunitTest {
    private GameController controller;
    private Player player1, player2;
    private Piece piece1, piece2;
    private Board board;

    @BeforeEach
    void setup() {
        player1 = new Player(0, 4, Color.RED);
        player2 = new Player(1, 4, Color.BLUE);

        board = new SquareBoard();
        controller = new GameController(Arrays.asList(player1, player2), board);


        piece1 = player1.getPieces().get(0);
        piece2 = player2.getPieces().get(0);
        board.getStartNode().addPiece(piece1);
        board.getStartNode().addPiece(piece2);
    }
```

Player 2명과 각각 말은 4개씩 보유하게 했고 판은 사각형으로 하였다. 두 말을 시작점에 놓고 6개의 테스트를 진행했다.

1) ThrowYutResultValid()

```java
    @Test
    void testThrowYutResultValid() {
        Yut yut = new Yut();
        for (int i = 0; i < 100; i++) {
            int result = yut.throwYutRand();
            assertTrue(result >= 0 && result <= 5, "윷 결과가 유효한 범위를 벗어났습니다: " + result);
        }
    }
```

0~5가 빽도~모이다. 그것 외에 다른 것이 나오는지 테스트 해보았다.

윷놀이에서 정해진 값이 나오는 것은 굉장히 중요하다고 생각해서 테스트 하였다.

이상한 값이 나오는 경우 “윷 결과가 유효한 범위를 벗어났습니다”라는 문구를 뜨게 했다.

2) testPieceMove

```
@Test
void testPieceMove() {
    BoardNode start = board.getStartNode();
    start.addPiece(piece1);
    piece1.setCurrentNode(start);

    controller.movePiece(piece1, 3);

    assertEquals(3, piece1.getDistanceMoved(), "말이 3칸 이동하지 않았습니다");
}
```

윷놀이에서 가장 중요한 것은 말이 윷의 결과만큼 움직이는 것이다. 그래서 말이 잘 움직이는 지 테스트 해보았다. piece1을 3만큼 움직여보았고 혹시 실패할시 “말이 3칸 이동하지 않았습니다.”라고 뜨게 하였다.

3) testCatchPiece()

```
@Test
void testCatchPiece() {
    BoardNode start = board.getStartNode();
    start.addPiece(piece1);
    start.addPiece(piece2);
    piece1.setCurrentNode(start);
    piece2.setCurrentNode(start);


    controller.movePiece(piece2, 2); // piece2 먼저 이동
    controller.movePiece(piece1, 2); // 같은 위치로 piece1 이동 → piece2 잡힘

    assertEquals(0, piece2.getCurrentNode().getId(), "잡힌 말이 시작점(ID 0)으로 돌아가지 않았습니다");
}
```

이것은 상대방의 말을 잘 잡을 수 있는지 체크하는 것이다. piece2를 2에 놓고 piece1을 2만큼 움직였을때 piece2가 잡히면서 시작점으로 돌아가는 지 테스트 해보았다. 실패할 경우 “잡힌 말이 시작점으로 돌아가지 않았습니다.” 라고 뜨게 하였다. 잡기는 윷놀이에서 변수를 많이 만들 수 있는 규칙이기 때문에 정확하게 구현되는 것이 중요하다고 생각했다.

4) testGroupPieces()

잡기와 유사한 업기도 테스트해보았다. piece1이 1에 있고 같은 팀의 다른 말을 1로 보냈을 때 업고 잘 가는지 테스트 해보았다. 실패할 경우 “말 업기가 되지

```
@Test
void testGroupPieces() {
    Piece secondPiece = player1.getPieces().get(1);

    BoardNode start = board.getStartNode();
    start.addPiece(piece1);
    start.addPiece(secondPiece);

    piece1.setCurrentNode(start);
    secondPiece.setCurrentNode(start);

    // 먼저 piece1 이동 (업히는 대상)
    controller.movePiece(piece1, 1);

    // 그 다음 secondPiece가 같은 칸으로 이동해서 업기 시도
    controller.movePiece(secondPiece, 1);

    assertTrue(secondPiece.getCarriedPieces().contains(piece1), "말 업기가 되지 않았습니다");
}
```

않았습니다.” 라는 문구를 뜨게 했다. 업기 또한 잡기와 유사하게 큰 변수를 많이 만들고 꼭 잘 기능해야하는 규칙이라고 생각했다.

5) testNextTurnChangesPlayer()

```
@Test
void testNextTurnChangesPlayer() {
    int initialIndex = controller.getCurrentPlayerIndex();
    controller.forceNextTurn();
    int newIndex = controller.getCurrentPlayerIndex();
    assertNotEquals(initialIndex, newIndex, "턴이 변경되지 않았습니다");
}
```

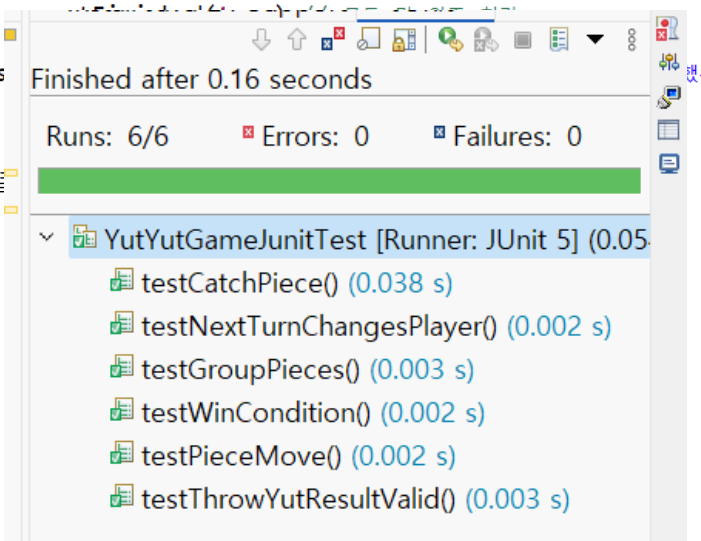
윷놀이를 할 때 턴이 넘어가지 않고 한명만 계속 윷을 던지게 되면 이것은 게임이라고 할 수 없을 만큼 심각한 상황이 된다. 그렇기 때문에 턴이 잘 넘어가는지 테스트 해보았다.
forceNextTurn(); 을 하고 나서 턴이 바뀌지 않으면 “턴이 변경되지 않았습니다.”라는 문구가 뜨게 하였다.

6) testWinConditon()

```
@Test
void testWinCondition() {
    for (Piece p : player1.getPieces()) {
        p.setFinished(true);// 모든 말 완주 처리
    }
    assertTrue(player1.allPiecesFinished(), "모든 말이 완주했는데 승리로 처리되지 않았습니다");
}
```

게임에서 승패가 결정나지 않고 움직일 말도 없는데 끝나지 않는다면 치명적인 결함이 있는 것이다. 그렇기 때문에 게임이 끝날 조건이 되면 잘 끝나는 지 확인하기 위해 테스트를 진행했다.
player1의 모든 말을 완주 처리했을 때 끝나지 않는 다면 “모든 말이 완주했는데 승리로 처리되지 않았습니다.” 라는 문구를 뜨게 하였다.

```
@Test
void testWinCondition() {
    for (Piece p : player1.getPieces()) {
    }
    ass
}
```



Junit Test 결

6 Github project report

- Github 주소: https://github.com/YutYutgame

김재원 : 윷놀이 로직 코딩, JavaFX UI 전환 보조 , Junit test, Sequence Diagram

이병현 : 윷놀이 로직 코딩, JavaFX UI 전환 담당 Usecase Diagram

정재훈 : 시작 UI, 보드판 UI 설계, 윷놀이 로직 코딩 보조, 영상 촬영, Class Diagram

활동 내용 사진