**설계**

1. **MVC 설계 개요**
2. **Class Diagram, Package Diagram**
3. **Use Case별 Sequence Diagram**
4. **Model, View, Controller 내 중요 요소**
5. **OOAD**
6. **SOLID 원칙**
7. **GRASP**

**구현**

1. **View 계층 인터페이스 일관성 유지**
2. **필수 변경점: UI 프레임워크 API 대응**
3. **변경 불필요: Model / Controller**

**1. MVC 설계 개요**

**흐름:** **View → Controller → Model → Controller → View**

해당 프로젝트를 설계함에 있어 중점을 둔 부분은 다음과 같다. **Controller에게 데이터 전달자를 담당시키고 Model은 순수 게임 로직, View는 입출력 처리만을 담당**하도록 해서 각각의 역할이 명확히 분리되도록 하였다. 이를 기반으로 View 계층을 Swing에서 JavaFX로 전환하더라도 수정 범위를 View에 국한시켜서 다른 계층에서의 변경을 최소화시켰다.

**Model:** 게임의 상태와 로직을 관리하는 부분입니다. 예를 들어 GameState 클래스는 말들의 위치, 턴 정보 등 **게임 상태**를 보관하고 윷 던지기나 말 이동 등의 **게임 로직을 처리**합니다. 또한 YutGameRules와 같은 클래스는 말 이동 시 적용되는 **게임 규칙**(윷놀이 규칙: 이동 경로, 잡기 등)을 캡슐화합니다. Model은 **UI에 직접 접근하지 않고**, 대신 게임 이벤트 객체(GameEvent)를 생성하여 변경 사항이나 요청 사항을 알리는 방식으로 동작합니다.

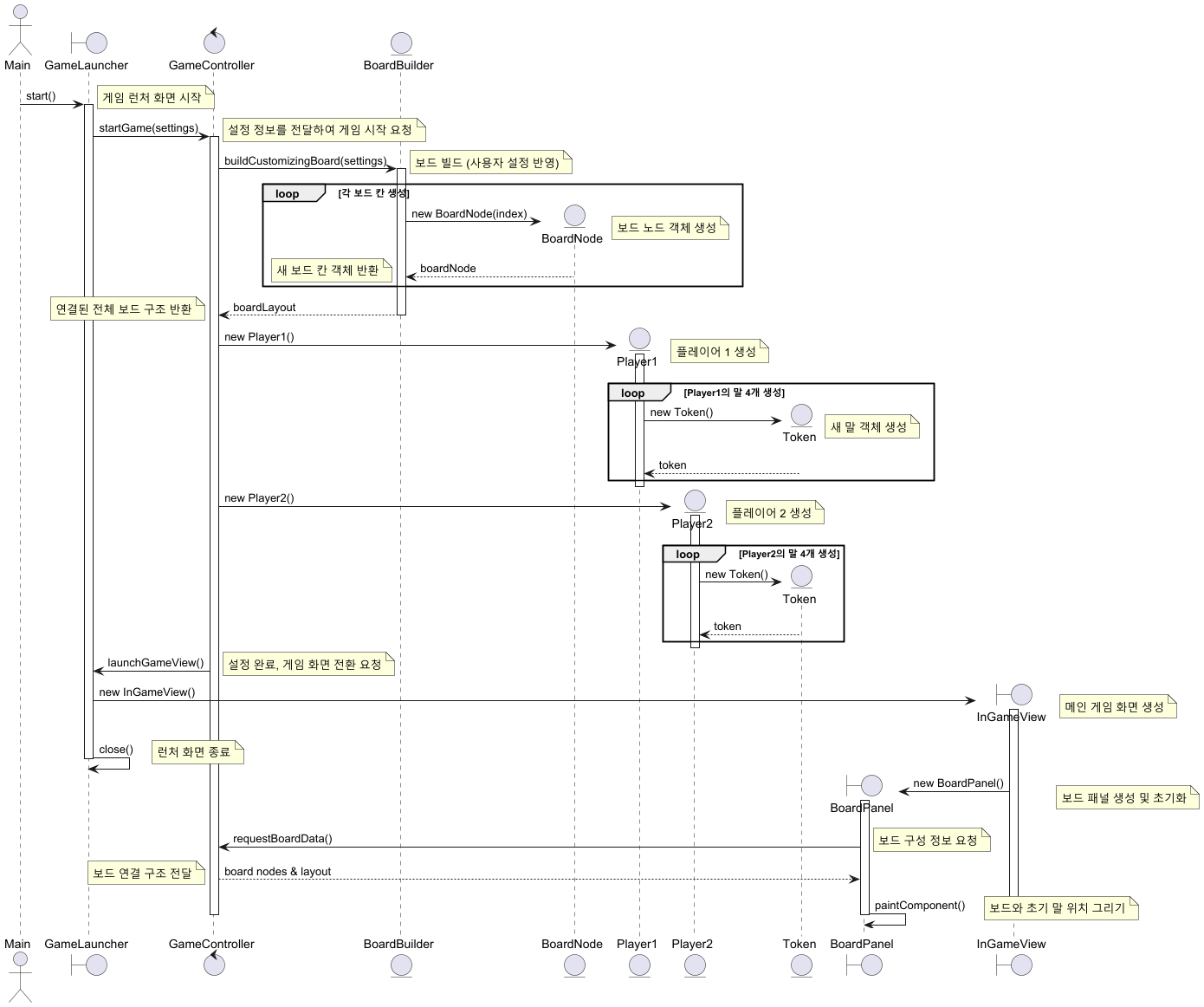
**View:** JavaFX 및 Swing으로 구현된 **사용자 인터페이스** 부분입니다. 말판, 말(token) 등의 그래픽 요소를 표시하고 사용자로부터 **입력**을 받아냅니다. View는 Cotroller의 중개를 통해 UI를 업데이트하거나 사용자에게 추가 입력을 요청합니다. View 자체는 게임 로직을 직접 수행하지 않고, 표시와 입력 역할에 집중합니다.

**Controller:** 사용자 입력을 model에 전달하고 model의 결과를 view에 반영하도록 중개하는 역할입니다. **주된 역할은 뷰로부터 전달된 사용자 액션을 받아 모델의 메서드를 호출**하는 것입니다. 또한 필요하다면 모델로부터의 이벤트를 받아 추가 로직을 수행하거나 (예: 턴 관리 등), 뷰를 업데이트하는 것을 도울 수 있습니다. 이를 통해 View와 Model이 서로 직접 통신하지 않으며, Controller가 양측을 연결하면서도 각각의 분리된 책임을 유지한다.

이러한 구조 덕분에 **모델-뷰 간 결합도**가 낮아집니다. 모델은 그저 게임 상태와 로직을 관리하고, 어떤 뷰가 그것을 받는지는 모릅니다. 뷰는 모델로부터 받은 이벤트만으로 화면을 갱신하거나 사용자 입력 흐름을 제어합니다. 한편 컨트롤러는 사용자 입력이 있을 때 모델의 메서드를 호출하여 모델을 변경하며, 이러한 변경으로 발생한 이벤트는 다시 뷰에 전달됩니다.

**2. Use Case Sequence 파악**

**2.1 Use Case 1: 윷놀이 판 디자인 및 게임 설정**



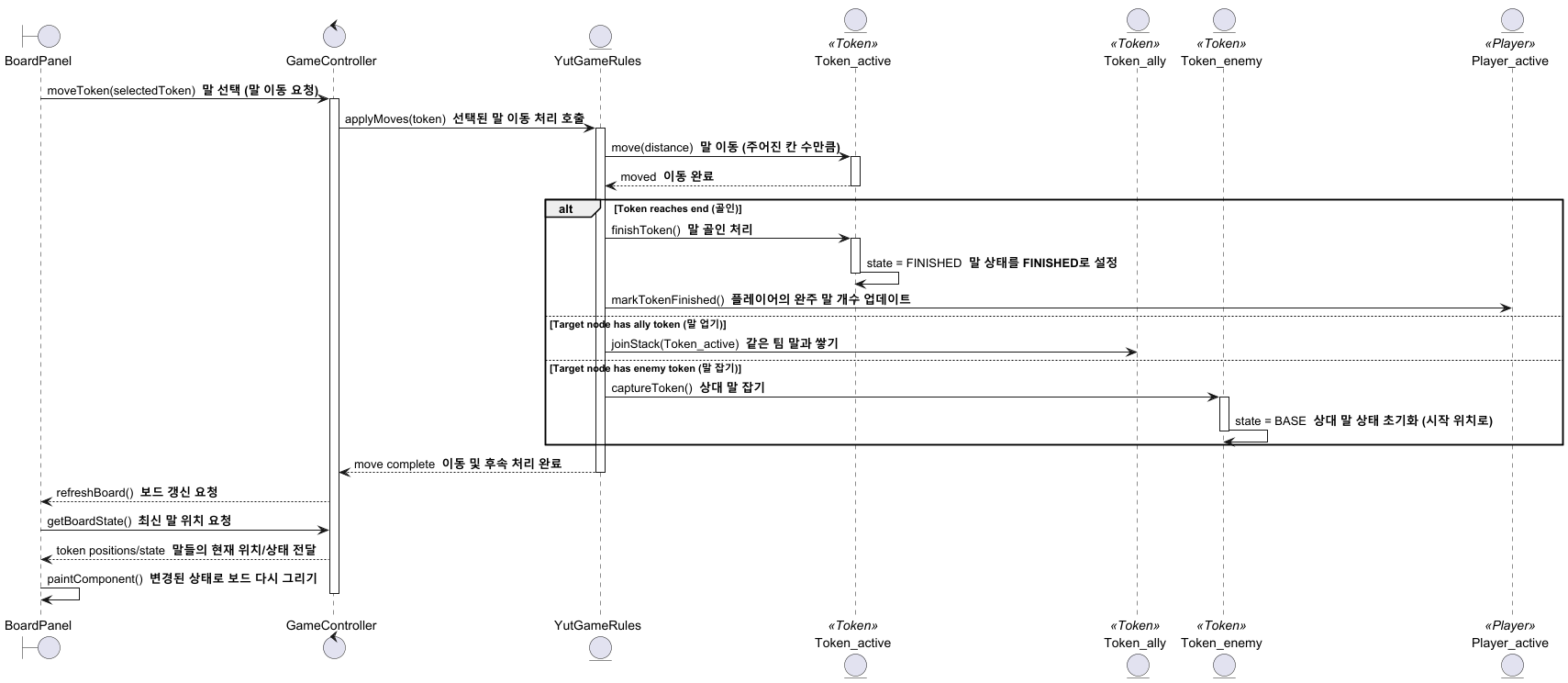
**개요:** Use Case 1은 게임을 시작하여 윷놀이 판을 커스타마이징하고 초기 설정을 수행하는 시나리오이다. 사용자가 게임 시작 옵션을 설정하면, 시스템은 윷놀이 판을 디자인하고 말들을 시작 위치에 배치하는 등 초기화 과정을 거친다. 이 과정에서 게임에 필요한 모든 Model 객체 생성 및 초기 상태 설정이 이루어지며, 최종적으로 초기 화면(UI)에 게임 판과 말들이 준비되었음을 보여준다.

1. **사용자 입력(View)** – 사용자가 게임 시작 버튼을 누르거나 설정을 완료하면, 초기 화면이 Controller에 게임 시작을 요청한다. (GameLauncher.start()).
2. **게임 시작 처리 요청(Controller)** – GameLauncher 등의 Controller 객체가 해당 요청을 받아 게임 초기화를 진행한다. 이 때, Controller는 직접 로직을 수행하지 않고, 이 요청을 Model 계층으로 전달한다.
3. **게임 보드/말 초기화 (Model)** – Controller는 Model 계층의 주요 로직 클래스에 게임 보드 구성 및 말 생성을 요청한다. Model 계층은 내부에서 보드를 생성하고 Token 객체들을 초기 위치에 생성/배치하는 로직을 실행한다. 또한 게임 규칙에 따른 팀 정보 설정, 턴 초기화 등의 작업도 Model에서 수행된다.
4. **초기화 완료 응답 (Model → Controller)** – Model 계층이 게임 설정을 완료하면 그 결과를 Controller에 반환한다.
5. **View 업데이트 지시 (Controller)** – Controller는 Model로부터 받은 초기화 결과를 바탕으로, View 계층에 게임 보드 그리기와 초기 말 표시를 지시한다 (refresh() 호출).
6. **초기 화면 표시 (View)** – View 계층은 Controller로부터 전달받은 데이터(또는 Model 객체를 조회)를 이용하여 윷놀이 판 UI를 구성한다. 말판과 말을 화면에 그리며, 사용자에게 게임 시작 상태를 보여준다.

이 흐름에서 **모든 상호작용은 View→Controller→Model→Controller→View의 순환**을 따르며, 각 단계에서 자신의 책임만 수행한다.

**2.2 Use Case 2: 말 이동 / 말 업기 / 말 잡기**

**개요:** Use Case 2는 **플레이어의 말(token)을 이동**시키는 시나리오로, 이동 과정에서 **말 업기**나 **말 잡기**가 발생하는 상황을 포함한다. 이 시나리오는 사용자가 윷 던지기 결과로 얻은 이동 거리만큼 말을 선택하여 옮기는 과정을 다루며, **게임 규칙에 따른 말의 상태 변화(업기 또는 잡기)**까지 처리한다. 크게 UI에서 말을 선택하는 입력, Model의 게임 규칙 적용, View의 보드 갱신으로 구분하였다.



1. **말 선택 입력 (View)** – 플레이어가 이동시키려는 말을 UI에서 선택한다. 보드 패널이 사용자 입력을 감지하여 선택된 말과 이동할 거리 정보를 Controller에 전달한다.
2. **이동 요청 처리 (Controller)** – GameController는 View로부터 말 이동 요청을 받으면, **해당 요청을 Model 계층의 로직으로 위임**한다. Controller는 Model 내 객체의 메서드를 호출하여 실제 게임 규칙에 따른 이동 처리를 수행하도록 한다.
3. **말 이동 로직 실행 (Model)** – Model 계층에서는 moveToken, acuumulateYut 등 로직을 수행하면서 **게임 규칙을 적용**한다.
   1. 우선 해당 말의 현재 위치와 이동할 거리를 기반으로 **새 위치를 계산**한다. 이때 윷놀이 말판의 경로에 따라 진행하며, 분기점이 없는 일반 이동이면 바로 새 칸을 결정한다.
   2. 이동 도중 **분기점**이 나타나 방향 선택이 필요하면, Model은 그 상태를 Controller에 알려주어(이 부분은 Use Case 3에서 처리) 방향 결정을 기다린다. 방향 결정 후에는 남은 이동을 완료하고 최종 위치를 확정한다.
   3. 도착한 **최종 위치에 다른 말이 있는지 검사**한다. 아군 말이 있으면 **말 업기** 규칙을 적용하여 말들을 같은 칸에서 함께 움직일 수 있도록 Model 상태를 표시한다.
   4. 상대 팀 말이 존재하면 **말 잡기**를 실행한다. 잡힌 말은 Model에서 해당 말의 상태를 베이스로 되돌리고, 잡은 측에게 보너스 (추가 윷던지기)를 부여하는 처리를 한다. 이러한 결과(잡기 여부, 보너스 여부 등)를 Model에서 계산한다.
   5. Model은 이동 완료 후 **최신 말 위치와 상태 정보**(예: 말이 이동된 새로운 좌표, 업기 여부 등…)를 준비한다.
4. **이동 결과 반환 (Model → Controller)** – Model 계층이 말 이동 및 관련 로직을 끝마치면, 그 **결과 데이터**를 Controller에 전달한다. // 예시를 들까…
5. **UI 갱신 지시 (Controller)** – Controller는 Model로부터 받은 결과를 바탕으로 View에 **보드 업데이트를 지시**한다.
   1. 이동된 말의 새로운 위치를 View에 알려 해당 말 아이콘을 새 위치로 옮기도록 한다.
   2. 만약 **말 업기**가 발생했다면, View에서는 한 칸에 여러 말이 있음을 표현해야 하므로 Controller는 업기 상태를 View에 전달하여 적절히 표시하도록 한다.
   3. **말 잡기**가 일어난 경우 Controller는 잡힌 말이 보드에서 제거되었음을 View에 알려 해당 말 아이콘을 제거하거나 베이스로 이동시키도록 하고, 잡았다는 피드백(..를 잡아 추가턴을 얻었습니다!)을 표시하도록 지시할 수 있다.
   4. **결과 반영 (View)** – View 계층은 Controller의 지시에 따라 **보드 UI를 갱신**한다. 선택된 말 아이콘을 새로운 위치로 이동시키고, 업기 상태라면 말 아이콘을 겹쳐서 표현한다. 잡힌 말이 있었다면 해당 말을 보드에서 제거하고 텍스트 메시지를 띄우는 등의 화면 변화를 보여준다. 모든 변경 사항은 **Model의 최종 상태를 반영**하도록 일관되게 수행된다.
6. **후속 처리** – 화면 갱신 후, Controller는 Model에 다음 턴 준비나 게임 종료 여부 검사(Use Case 5 참고)를 수행할 수 있다.

**2.3 Use Case 3: 분기점 방향 선택**

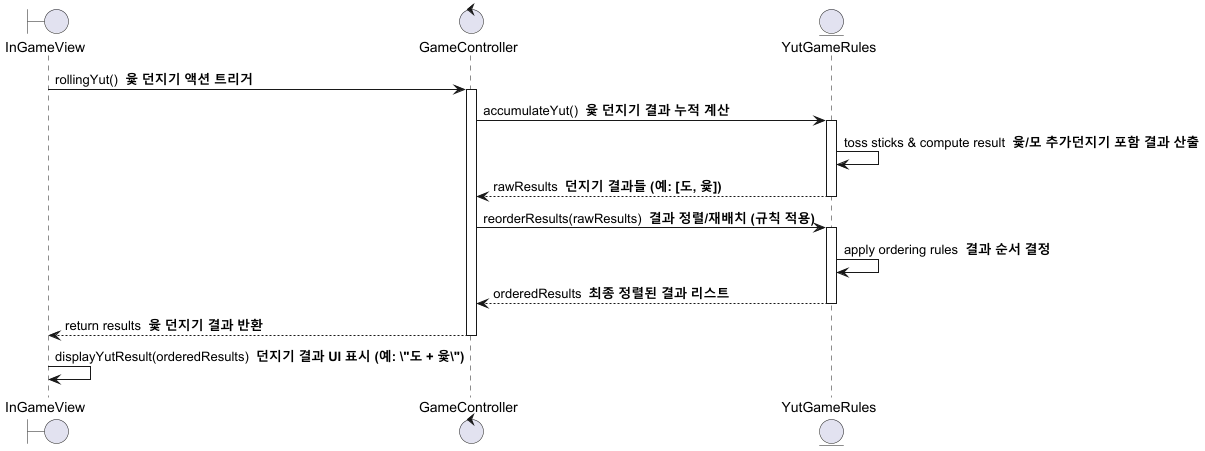
개요: Use Case 3은 **윷놀이 말이 보드의 분기점에 도달했을 때 플레이어가 진행 방향을 선택**하는 상황을 다룬다. 갈래가 나뉘어지는 지점에서 플레이어는 말이 앞으로 **어느 방향으로 진행할 지 결정**해야 한다. 해당 시나리오에서는 **사용자와 시스템 간의 상호작용이 한 번 더 필요하므로 추가적인 사용자 입력이 필요하다.** 텍스트, 스크린샷, 라인, 도표이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

1. **분기점 도달 상태 감지 (Model)** – 말이 이동 도중 분기점 칸에 정확히 멈췄다면, Model 계층은 추가 입력 필요 상태임을 인지한다.
2. **분기점 알림 (Controller)** – Controller는 Model로부터 이동 결과를 받을 때 분기점에 멈췄다는 정보를 함께 전달받는다. 이를 확인한 GameController는 사용자 결정을 받아야 함을 인식한다.
3. **사용자 입력 요청 (View)** – Controller는 분기점에서의 방향 선택을 사용자에게 요구하기 위해 View 계층에 지시를 내린다. 구체적으로, View 쪽에서 선택 창을 한다. 이 단계에서 사용자가 방향을 선택할 때까지 기다린다.
4. **플레이어 방향 선택 (View)** – 플레이어는 팝업된 UI에서 원하는 방향 버튼을 누른다. View 계층은 그 선택 결과를 즉시 Controller에 전달한다.
5. **선택 결과 전달 (Controller → Model)** – Controller는 받은 사용자의 방향 선택 정보를 Model에 다시 전달하여, 남은 이동을 이어서 처리하도록 한다. Model 내 메서드를 호출하여 분기점 이후의 경로를 결정한다.
6. **남은 이동 처리 및 완료 (Model)** – Model 계층은 입력 받은 방향에 따라 해당 경로로 말 이동을 이어서 수행한다. 분기점에서 새로 선택된 길을 따라 말의 최종 위치를 결정하고, 이동을 완료한다. 이동 완료 시 혹시 다른 말과 충돌(업기/잡기) 발생하면 (Use Case 2의 로직처럼) 추가 처리를 하고, 최종적으로 말의 위치와 상태를 갱신한다.
7. **완료 결과 반환 (Model → Controller)** – Model은 분기점 처리까지 포함한 최종 이동 결과(예: 최종 위치, 잡기 여부 등)를 Controller에 반환한다.
8. **UI 최종 갱신 (Controller → View)** – Controller는 Model의 최종 결과를 받아 View에게 보드 상태 갱신을 지시한다. 분기점을 지난 후의 말 위치로 아이콘을 옮기고, 변화된 사항을 화면에 반영하도록 View를 업데이트한다.
9. **화면 표시 (View)** – View 계층은 전달받은 정보대로 말의 최종 위치를 그린다.

**2.4 Use Case 4: 윷 던지기**

**개요:** Use Case 4는 플레이어가 **윷을 던지고 결과를 처리하는 과정**을 다룬다. 윷 던지기 결과로 이동할 수 있는 칸 수가 결정되며 (예: 도, 개, 걸, 윷, 모, 빽도 등..), 경우에 따라 추가 턴이 주어지기도 한다. 이 시나리오에서는 **사용자가 윷 던지기 UI를 조작하는 입력**부터 **Model의 랜덤 로직을 통한 결과 산출**, **View에 결과 표시**까지의 흐름을 포함한다.



1. **윷 던지기 입력 (View)** – 사용자가 자신의 턴에 UI 상의 "윷 던지기" 버튼을 클릭한다. Swing 기반 게임 화면(View 계층)이 이 이벤트를 감지하여 Controller에 윷 던지기 동작을 요청한다.
2. **던지기 액션 전달 (Controller)** – GameController는 윷 던지기 요청을 받으면, **실제 윷 던지기 로직 실행을 Model에 위임**한다. Controller는 Model 계층의 윷 처리 모듈을 호출하여 **결과 계산**을 수행하도록 한다.
3. **윷 결과 계산 (Model)** – Model 계층에서는 윷 던지기 로직이 수행된다. 이 로직은 **게임 규칙에 따른 윷 결과 결정**을 포함하며, 도/개/걸/윷/모 중 하나를 결정한다.
   1. 결정된 결과에 따라 **이동할 칸 수 및 추가 턴 여부** 등의 부가 정보도 Model에서 판단한다. (예: '윷'이나 '모'가 나오면 한 번 더 던질 권한 부여)
   2. Model은 그 결과를 **게임 상태에 반영**한다.
4. **결과 반환 (Model → Controller)** – Model이 윷 던지기 결과 계산을 완료하면, **결과 데이터**를 Controller에 전달한다. 여기에는 예를 들어 "결과=도(1칸)", "추가 턴 없음" 등의 정보가 포함된다. Model은 단순 값 반환 또는 Game 상태 객체 갱신 형태로 결과를 Controller와 공유할 수 있다.
5. **결과 해석 및 View 지시 (Controller)** – Controller는 Model로부터 받은 결과를 확인하고, 이를 사용자에게 보여주기 위해 View를 업데이트한다.
   1. 윷 던지기 결과를 View에게 전달하여 화면에 표시하도록 한다.
   2. 추가 턴이 있는 경우 그 상황을 알리는 UI 업데이트 (예: "추가 턴을 얻었습니다" 메세지)도 지시할 수 있다.
6. **결과 표시 (View)** – View 계층은 Controller로부터 전달받은 윷 결과 정보를 **화면에 표현**한다. 또한 추가 턴 안내나 말 이동 가능 표시 등 부가적인 UI 변화도 반영한다. 플레이어는 이 결과를 보고 이동시킬 말을 선택하게 되며, 이는 Use Case 2로 이어진다.
7. **다음 단계 진행** – 윷 던지기 결과 표시 후, Controller는 게임 진행을 이어가기 위해 대기한다. 플레이어가 말을 선택하면 (Use Case 2로 연결), 혹은 추가 던지기 권한이 있으면 다시 윷 던지기 버튼을 누르는 등, 다음 액션으로 넘어간다. Use Case 4 자체는 한 번의 윷 던지기와 그 결과 처리를 끝으로 완료된다.

**2.4 Use Case 5: 한 팀이 모든 말을 내보냈을 때 게임 종료**

**개요:** Use Case 5는 게임 종료 조건을 만족했을 때 승자를 결정하고 게임을 종료하는 시나리오이다. 윷놀이 게임에서는 한 팀의 모든 말이 결승 지점(골인 지점)을 통과하여 판을 떠나면 그 팀이 승리하며 게임이 끝난다. 본 시나리오는 특정 팀의 말 완주 확인부터 게임 종료 처리 및 결과 표시까지의 흐름을 포함한다. 일반적으로 말 이동(Use Case 2) 후에 종료 조건을 체크하며, 조건 만족 시 더 이상의 턴을 중지하고 최종 결과를 보여준다.

텍스트, 도표, 평행, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

1. **게임 종료 조건 체크 요청 (Controller)** – 매 턴의 말 이동이 끝날 때마다, Controller(GameController)는 Model에 게임 종료 조건을 확인하도록 요청한다. (예: hasFinished() 호출) 이는 말 이동 직후 이루어지며, 특정 팀의 말이 모두 나갔는지 검사하는 로직을 실행한다.
2. **승리 조건 판정 (Model)** – Model 계층에서는 현재 게임 상태를 조회하여 승리 조건을 판정한다. 예를 들어 각 팀별로 완료된 말의 개수를 추적하고 있다가, 어떤 팀의 완료 말 수 == 전체 말 수인지를 검사한다.
   1. 만약 조건이 충족되지 않으면 False를 반환하고 흐름을 종료한다(게임 계속 진행).
   2. 어느 한 팀이 승리 조건을 만족하면 Model은 게임 종료 신호를 Controller에 전달한다. 이때 우승 팀 정보("Player 승리!")를 같이 제공하거나, Model 내부에 상태를 업데이트해 둘 수 있다.
3. **종료 신호 처리 (Controller)** – Controller는 Model로부터 게임 종료 여부와 우승팀 정보를 전달받으면, 게임을 종료하는 절차를 진행한다. 여기에는 더 이상의 입력/턴 진행을 막는 제어와 결과를 UI에 표시하는 작업이 포함된다. Controller는 즉시 다음과 같은 동작을 수행한다:
   1. **게임 루프 종료:** 내부적으로 게임 진행을 멈추고, 남은 턴이나 타이머가 있다면 취소한다.
   2. **View에 결과 표시 지시:** Controller는 View 계층에 우승 팀과 게임 종료를 알리는 화면을 보여주도록 명령한다.
4. **게임 종료 화면 표시 (View)** – View 계층은 Controller의 지시에 따라 게임 종료 메시지와 우승 정보를 사용자에게 보여준다. View는 전달받은 승리 팀 정보 등을 시각적으로 표현하는 일만 담당한다.
5. **게임 종료 후 처리** – 사용자가 종료 메시지를 확인하고 게임을 닫거나, 새로운 게임을 시작할 수 있다.

**3. Model, View, Controller 내 중요 요소**

MVC 아키텍처에 따라 View, Model, Controller 각 계층별로 핵심 클래스를 정리하고, 주요 메서드와 해당 계층의 역할 및 다른 계층과의 상호작용 방식을 설명한다. 이 구조는 앞서 언급했듯이 각 계층이 자신의 책임만을 수행하도록 설계하였다. 이는 분리를 바탕으로 UI를 Swing에서 JavaFX로 교체할 시에 View계층에서의 코드 수정을 최소화하기 위함이다.

**View 계층**

**<InGameView>**

InGameView 클래스는 게임 화면을 구성하고, 사용자 행동(윈도우 버튼 클릭, 토큰 선택, 분기 선택 등)에 대한 입력을 받아 컨트롤러에게 전달하며, 토큰 위치·상태 변경 결과를 시각적으로 보여 주는 역할을 담당합니다.

**역할**

1. **보드(Board) 및 토큰(Token) 렌더링**
   * 내부에 BoardView나 캔버스(canvas) 컴포넌트를 포함하여, 현재 GameState가 알려주는 토큰 위치 정보를 그립니다.
   * 토큰 이미지나 말판 이미지 위에 토큰의 좌표를 매핑하여 표시하고, 업혀 있는 토큰들은 계층 구조에 맞게 겹쳐서 렌더링합니다.
2. **사용자 입력 처리 (버튼, 분기 선택 등)**
   * 윷 던지기 버튼 클릭: 사용자가 “윷 던지기” 버튼을 누르면 onRollYut 콜백을 트리거하여 실제 던지기 로직을 실행하도록 합니다.
   * 토큰 클릭/선택: 사용자가 본인의 말들 중 선택한다면 GameController에게 해당 토큰을 이동할 예정이라고 알립니다.
   * 분기점 선택 다이얼로그: 만약 현재 이동해야 할 토큰이 분기점(갈림길)에 위치해 있다면, 미리 정의된 콜백(onBranchSelected)을 통해 선택 가능한 경로 목록을 보여주고, 사용자가 하나를 선택하면 그 결과를 컨트롤러에 전달합니다.
3. **상태 패널**
   * 화면 우측에 플레이어 및 말 등의 게임 정보를 보여 줍니다.
4. **게임 초기화 및 종료 뷰 관리**
   * initialize() 메서드를 통해, 새 게임이 시작될 때 보드판과 토큰 초기 위치를 설정합니다.
   * 게임이 끝나면 clearBoard() 등의 메서드를 호출하여, 화면을 초기화하고 “재시작/종료” 선택 창을 띄워 줍니다.

**Model 계층**

**<YutGameRules>**

**YutGameRules** 클래스는 윷놀이에서 윷 던지기 결과 산출부터 토큰 이동 및 그에 따른 잡기·업기·완주 처리까지, 실제 게임 규칙 전반을 구현하는 모듈입니다.

**역할**

1. **윷 던지기 결과 계산**
   * 실제 난수를 기반으로 “도·개·걸·윷·모·빽도” 결과를 산출하고,
   * “윷이나 모가 나오면 한 번 더 던진다”는 룰에 따라 반복 던지기를 처리하여 최종 결과 목록과 메시지를 반환합니다.
2. **사용자 입력 순서 재배열 검증 (validateReorderInput)**
   * 플레이어가 직접 던진 윷 결과들에 대한 원하는 배열을 제출하면 해당 목록을 실제 던진 결과와 비교해 동일한 멀티셋인지, 범위가 올바른지 확인하고, 이상이 없으면 재배열된 리스트를, 문제 있으면 에러 메시지를 담아 반환합니다.
3. **토큰(Token) 이동 로직 (moveToken, moveTokenBackward)**
   * moveToken은 파라미터로 받은 토큰과 이동할 칸 수(–1~5), 분기 경로 결정자(branchSelector)를 이용해, 현재 위치에서부터 한 칸씩 전진하거나 후진하며 최종 도착 지점을 계산합니다. 먼저 token.getTopMostToken()으로 실제 움직일 대표 토큰을 정하고, 상태가 ACTIVE인지 확인합니다. 그 뒤 calculateTargetNode를 호출해, 각 칸마다 current.getNextNodes()를 조회하고 분기점에서는 branchSelector로 경로를 선택하여 목표 노드를 결정합니다. 만약 분기점이 아닌 단일 경로라면 자동으로 한 칸씩 다음 노드를 따라 이동하고, 보드 끝을 만나면 targetNode는 null이 되어 곧바로 완주 처리(finishToken)로 넘어갑니다.
   * 목표 노드(targetNode)가 null이 아닐 때는, 현재 위치(currentNode)에서 leave(actualToken)로 토큰을 제거한 뒤, enter(actualToken)로 목표 노드에 올리고 tokenManager.updateTokenPosition으로 위치 정보를 갱신합니다. 이때 업힌 토큰(stackedTokens)이 있다면, 대표 토큰과 같은 노드로만 위치 정보만 변경해 줍니다. 이후 handleCaptureAndStacking을 호출해 해당 노드 위에 존재하는 다른 플레이어 소유 토큰은 resetToken으로 잡아 출발점으로 돌려보내고, 같은 소유 토큰은 addStackedToken으로 대표 토큰 아래로 올려 업음으로써 잡기·업기 기능을 모두 수행합니다. 마지막으로 MoveResult 객체를 반환해 “이동 성공 여부(success), 잡기 여부(catched), 완주 여부(finished), 메시지”를 컨트롤러에 전달합니다.
   * moveTokenBackward는 빽도(–1칸) 결과가 나왔을 때 호출하는 메서드로, 동작 원리는 moveToken과 유사하나 calculateTargetNode 대신 current.getPreviousNodes()를 따라 한 칸씩 뒤로 이동합니다. 이후에도 목표 노드가 없다면(출발점보다 뒤로 더 갈 수 없으면 이동 실패), 아니면 앞서 설명한 대표 토큰 제거→목표 노드 입장→위치 갱신→업힌 토큰 동기화→잡기·업기 처리 순으로 진행하여 MoveResult를 리턴합니다.
   * 이렇게 moveToken과 moveTokenBackward 두 메서드를 통해 “한 칸씩 이동하면서 분기점에서는 branchSelector를 호출해 다음 노드를 선택하고, 보드 끝이나 출발점까지 가면 완주·실패 처리를 한다”는 핵심 로직이 완결됩니다.
4. **잡기 및 업기 처리 (handleCaptureAndStacking)**
   * 이동을 완료한 토큰이 도착한 노드에 올라와 있는 다른 토큰들을 검사하여
     1. **다른 플레이어 소유**면 resetToken을 호출해 해당 토큰을 출발점으로 되돌리고(잡기),
     2. **같은 플레이어 소유**면 해당 토큰을 현재 대표 토큰 아래로 쌓아 업는 처리를 수행합니다.
   * 잡기 여부를 리턴해 상위 로직에서 한 번 더 던지기 등 추가 행동을 결정할 수 있도록 돕습니다.
5. **결과 객체 반환**
   * YutThrowResult: 던진 윷 결과 목록과 메시지 리스트를 담아 반환.
   * MoveResult: 이동 성공 여부(success), 잡기 발생 여부(catched), 완주 여부(finished), 그리고 출력할 메시지를 담아 반환.
   * 이를 통해 다른 모듈은 복잡한 내부 로직 없이도 무슨 일이 일어났는지만 확인해서 다음 흐름을 이어갈 수 있습니다.

**<GameState>**

**GameState** 클래스는 게임 전체의 진행 흐름과 상태를 한 번에 관리하는 핵심 책임을 가집니다. 즉, 개별 말 이동이나 룰 적용은 YutGameRules에 위임하되, 누구의 턴인지, 남은 이동 횟수가 얼마인지, 현재 어떤 단계에 있는지, 그리고 최종 승리자가 누구인지 등을 추적·판단하는 기능을 담당합니다.

**역할**

1. **플레이어 목록과 순서 관리**
   * 게임에 참여한 Player 객체들을 List<Player> players로 보관하고, 매 턴마다 currentPlayer를 갱신하여 누구 차례인지를 관리합니다.
2. **보드(Board) 및 말 위치(TokenPositionManager) 관리**
   * 실제 윷놀이판을 표현하는 Board 객체와, 각 토큰들이 보드 위 어느 칸에 있는지를 기록·조회하는 TokenPositionManager를 한 곳에서 참조하여, 말이 이동할 때마다 위치를 업데이트하거나 잡기/업기 상황을 판단할 수 있도록 돕습니다.
3. **게임 단계 관리**
   * GamePhase enum을 통해 아직 시작 전인지, 진행 중인지, 혹은 이미 종료되었는지를 상태 변수 GamePhase로 표시합니다. 예를 들어 FINISHED 단계일 때는 더 이상 윷 던지기나 이동 명령을 받지 않도록 막아 줍니다.
4. **남은 이동 횟수(Remaining Moves) 추적**
   * 한 턴 동안 윷을 던져서 나온 여러 회차(윷·모가 나와서 한 번 더 던짐)를 List<Integer> remainingMoves에 보관해 두고, 플레이어가 움직여야 할 연속된 칸 수를 차례대로 꺼내 사용하도록 관리합니다. 이 데이터가 바닥나면 “턴 종료 → 다음 플레이어로 넘어감”을 자동으로 판단하게 됩니다.
5. **승리자 판단(Win Condition)**
   * 이동 도중 혹은 턴 종료 시점마다 어떤 플레이어가 모든 말을 완주했는지를 감시하여, 최종적으로 Player winner 필드에 이긴 플레이어 객체를 저장합니다. 이후 게임 단계(phase)를 FINISHED로 바꾸어, 더 이상 턴을 진행할 수 없음을 Controller 에 알려 줍니다.

**<Token>**

Token 클래스는 각 플레이어가 사용하는 말 객체 하나를 나타내며, 게임 진행 중 말이 가질 수 있는 모든 상태(활성/비활성/완주), 업기 여부, 분기 선택 정보 등을 전부 관리합니다. 이 클래스를 통해 컨트롤러는 어떤 토큰이 어디에 있고, 누구 소유이며, 업혀 있는 말은 어떤 것인지를 쉽게 파악하고 조작할 수 있습니다.

**역할**

1. **말 식별 정보 관리**
   * Name 필드를 통해 말 하나를 고유하게 구분합니다.
   * owner (Player 객체)를 통해 이 말이 어느 플레이어에게 속해 있는지 저장하여, 잡기·업기 처리 시 소유자를 구별할 수 있습니다.
2. **업기 정보 관리**
   * stackedTokens: List<Token> 리스트에, 이 토큰 위에 업혀 있는 모든 하위 토큰을 저장합니다.
   * addStackedToken(Token t) 메서드를 통해 다른 토큰을 현재 대표 토큰(“top-most”) 아래로 쌓을 수 있으며, getStackedTokens()로 업힌 말 목록을 조회합니다.
   * 업힌 말이 대표 토큰과 함께 이동하거나 완주 처리될 때, 재귀적으로 함께 움직이도록 설계되어 있습니다.
3. **대표 토큰 결정**
   * getTopMostToken() 메서드는 현재 이 토큰이 만약 다른 토큰 밑에 업혀 있다면, 최종적으로 보드 위에서 실제로 움직일 대표 토큰이 무엇인지를 찾아 반환합니다. 소유자(owner)가 가진 모든 토큰을 순회하며, 자신이 stackedTokens 목록에 포함된 토큰이라면 그 위의 대표 토큰을 재귀적으로 찾아 올라갑니다.
   * 이를 통해 토큰 이동 명령이 들어왔을 때 실제 보드 위에서 움직여야 할 토큰을 손쉽게 식별할 수 있습니다.
4. **분기 선택 및 이전 노드 정보 저장**
   * nextBranchChoice: BoardNode 필드는, 분기점에 진입했을 때 다음 이동 구간으로 어느 노드를 선택했는지를 기억합니다.
   * previousNode: BoardNode 필드는, 직진할 때 이전에 머물렀던 노드를 저장합니다.

**Getter/Setter 및 편의 메서드 제공**

* + getName(), getOwner(), getState(), getStackedTokens(), getNextBranchChoice(), getPreviousNode() 등 다양한 접근자(Accessor)로 내부 데이터를 안전하게 읽습니다.
  + setState(TokenState newState), setNextBranchChoice(BoardNode node), setPreviousNode(BoardNode node) 등의 설정자(Mutator)로 상태나 분기 정보를 업데이트합니다.
  + clearStackedTokens() 메서드는 완주 시점이나 리셋이 필요할 때 업힌 토큰 목록을 비워줍니다.

**Controller 계층**

**<GameController>**

게임 진행 제어의 중심 클래스로, 한 판의 윷놀이 게임에 필요한 상태 관리와 흐름 제어를 담당합니다. 그러나 게임 규칙의 실제 판단이나 데이터 변경 로직은 모두 Model 클래스들에 위임하며, GameController 자체는 그 호출 순서와 흐름을 관리합니다.

**역할**

1. **윷 던지기 제어 및 결과 수집**
   * rollingYut() 메서드를 통해 사용자가 “윷 던지기” 버튼을 눌렀을 때 호출됩니다.
   * 테스트 모드 혹은 일반 모드 여부에 따라 YutGameRules.throwSingleYut()을 반복 호출하여 “도·개·걸·윷·모·빽도” 결과를 수집하고, 결과마다 뷰에 메시지를 띄워 줍니다.
2. **획득한 이동 결과 목록(steps 리스트) 전개 및 토큰 이동 루프**
   * 던진 윷의 결과가 리스트 형태로 누적되면, 그 순서대로 하나씩 꺼내어 토큰을 움직일 차례를 실행합니다.
   * 각 결과마다 handleMoveExecution() 메서드를 호출하여 “현재 플레이어가 이동 가능한 토큰을 선택 → 분기점이면 경로 선택 → 실제 이동(모델 호출) → 뷰 업데이트”를 반복합니다.
   * 토큰 이동 시 발생하는 잡기(isCatched) 여부나 완주(isFinished) 여부를 확인합니다. 잡기가 발생하면 추가 턴에 관한 메시지를, 완주가 발생하면 곧바로 handleGameEnd()를 호출하여 승리자 판별과 재시작/종료 선택 창을 뜨게 합니다.
3. **토큰 선택 및 분기 선택 중재**
   * 이동 차례가 돌아온 플레이어는 view.selectToken(...)을 통해 자신이 움직일 토큰을 직접 골라야 합니다.
   * 만약 현재 토큰의 위치가 분기점(BoardNode.getNextNodes().size() > 1)이라면, 컨트롤러는 view.selectPath(options)를 호출해 사용자가 어느 방향으로 갈지 선택하도록 한 뒤,
   * 선택된 경로를 토큰의 setNextBranchChoice(...)에 저장하여, 모델이 해당 분기 경로를 따르도록 합니다.
4. **모델(GameState/YutGameRules) 호출 및 상태 동기화**
   * 실제 “토큰을 한 칸씩 움직여야 할지, 잡기·업기·완주 여부는 무엇인지” 같은 구체적인 룰 처리는 모두 YutGameRules.moveToken(...)에 위임합니다.
   * 컨트롤러는 GameState를 통해 현재 각 토큰의 위치(TokenPositionManager)를 조회하거나 업데이트하고,
   * YutGameRules로부터 반환된 MoveResult(성공 여부, 잡기 여부, 완주 여부, 메시지 등)를 받아서 다음 화면 갱신 또는 턴 전환, 게임 종료 로직을 결정합니다.
5. **뷰 업데이트 및 사용자 피드백**
   * 토큰 이동이 정상적으로 처리되면(moveResult.isSuccess()), 컨트롤러는 view.updateBoard(...)와 view.updateTokens()를 호출하여 실제 보드 UI와 토큰 위치를 화면에 다시 그립니다.
   * 또한 view.showMessage(...)와 view.showError(...)를 통해 “잡혔습니다!”, “완주했습니다!” 등 각 단계별 메시지를 사용자에게 보여 줍니다.
6. **턴 교체 및 승리 판단**
   * 이동 결과 리스트가 모두 소진되면(remainingMoves가 빈 목록이 되면) 컨트롤러는 내부적으로 gameState.nextPlayer()를 호출하여 “다음 플레이어로 턴을 넘기거나”,
   * MoveResult.finished == true인 경우에는 handleGameEnd()를 통해 해당 플레이어를 승리자로 확정하고, 재시작 혹은 종료 여부를 묻는 창(GameEndChoice)을 띄워 줍니다.
   * 재시작을 선택하면 onGameRestart.run()을, 종료를 선택하면 onGameExit.run()을 실행해 애플리케이션 차원의 후속 처리(화면 전환, 리소스 정리 등)를 트리거합니다.

**4. OOAD**

**캡슐화**

캡슐화는 객체 내부의 데이터를 private 등으로 은닉하고, 외부에는 필요한 기능만 public 메서드 형태로 제공하여 상호작용하는 것입니다. YutGameRules는 내부 필드를 전부 private으로 숨겨 두었고 외부에서는 오직 public 메서드를 통해서만 동작을 실행하거나 결과를 조회할 수 있게 하였습니다. 그리고 결과를 담는 객체들 또한 내부 상태를 직접 변경할 수 없도록 private final 필드 + getter만 허용하는 형태로 설계하였습니다.

**5. SOLID 원칙**

**단일 책임 원칙 (SRP)**

단일 책임 원칙은 한 클래스가 한 가지 책임만을 가져야 하고 변경의 이유도 하나뿐이어야 한다는 원칙이다. 설계 과정에서 SRP를 적용하고자 하였고 잘 따르고 있다. Token 클래스는 게임 말 한 개의 데이터와 관련 동작만 관장하며, InGameView 클래스는 UI 표시 및 입력 처리만 담당하는 등 각 클래스가 명확한 역할을 갖는다. 예를 들어 Token 클래스의 변경은 말의 상태/동작과 관련된 요구사항 변화에 한정되며, UI 변경이나 게임 흐름 제어와는 무관하다. 이러한 SRP 준수로 인해 시스템은 구조가 단순해지고 클래스별로 변경에 강건하며, 개별 구성요소를 이해하거나 수정하기가 수월해졌다.

**개방-폐쇄 원칙 (OCP)**

개방-폐쇄 원칙은 소프트웨어 구성 요소가 확장에는 열려 있고 수정에는 닫혀 있어야 한다는 원칙이다. 윷놀이 게임 구현에서 OCP의 적용은 주로 UI 교체 가능 구조와 관련하여 드러난다. 예를 들어, 현재 Swing으로 구현된 InGameView를 JavaFX 기반의 새로운 뷰로 교체하거나 추가하더라도, GameController나 Model 로직을 수정하지 않고도 새로운 View 계층을 추가하여 확장할 수 있다. 이는 GameController가 View의 구체적 클래스에 의존하지 않고 독립적으로 상호작용하도록 설계되었기 때문에 가능하다. 그 결과 시스템은 기존 코드를 변경하지 않고도 새로운 UI 구현을 확장 형태로 받아들일 수 있어 OCP를 만족한다. 또한 게임 규칙의 변경이나 추가 기능 구현 시에도, YutGameRules를 확장하거나 새로운 규칙 클래스(예: 변형된 규칙 집합)를 추가하는 방식으로 구현을 확장할 수 있으며 기존 클래스들의 수정을 최소화한다. 이러한 구조는 변경에 유연하며 요구사항이 추가될 때 안정성을 높인다.

**6. GRASP**

**정보 전문가 (Information Expert)**

정보 전문가 원칙은 특정 기능을 수행하는 데 필요한 정보를 가장 많이 알고 있는 클래스에게 그 책임을 할당하는 원칙이다. 본 윷놀이 게임 설계에서는 말(Token)의 위치와 이동 같은 핵심 정보는 **Token 클래스**가 가장 잘 알고 있으므로, 해당 클래스가 자신의 움직임 처리나 상태 변경 책임을 맡고 있다. 예를 들어, Token 클래스는 자신의 현재 위치, 이동 가능한 경로 및 도착 여부 등을 판단하는 기능의 정보 전문가로서 동작한다. 또한 **YutGameRules 클래스**는 윷놀이의 규칙(윷 던지기 결과 해석, 이동 거리 결정, 추가 턴 여부 등)에 대한 정보를 집중적으로 보유하고 있어 게임 규칙 적용 책임을 맡는 정보 전문가 역할을 한다. 이처럼 필요한 정보를 많이 가진 클래스에 책임을 할당함으로써 설계가 **정보 전문가 원칙**을 따르고 있으며, 이는 각 기능이 가장 관련 깊은 클래스에 구현되어 효율적이고 일관된 동작을 가능하게 한다.

**창조자 (Creator)**

창조자 원칙은 어떤 객체가 다른 객체를 생성할 때, 생성되는 객체에 필요한 데이터를 많이 가지고 있거나 구성 요소로 포함하는 객체가 그 **생성 책임**을 가져야 한다는 설계 원칙이다. 윷놀이 게임 구현에서 **GameController 클래스**는 게임 진행에 필요한 여러 객체를 초기화하고 생성하는 책임을 맡는다. 예를 들어, GameController는 게임 시작 시 말(Token) 객체들을 생성하거나, 게임 규칙을 관리하는 YutGameRules 객체를 생성 및 보유한다. GameController가 이러한 객체들을 포함하거나 이들과 밀접하게 연관되어 있으므로, **창조자 원칙**에 따라 해당 객체들의 생성을 담당하도록 설계되었다. 이로써 객체 생성 로직이 관련된 한 곳에 모여 유지보수가 쉬워지고, 생성과 사용의 응집도가 높아졌다.

**컨트롤러 (Controller)**

컨트롤러 원칙은 시스템 외부로부터 들어오는 입력 이벤트를 처리하고 해당 책임을 적절한 객체에 위임하는 **조정자 역할**의 객체를 두는 설계 개념이다. 본 시스템에서 **GameController 클래스**가 바로 이러한 컨트롤러 역할을 수행한다. GameController는 사용자 입력(예를 들어, 윷 던지기 버튼 클릭이나 말 이동 선택 등)을 **View**로부터 받아서, 그에 따른 게임 로직을 호출하고 업데이트를 처리한다. 구체적으로 GameController는 입력을 받으면 **Model** 영역의 클래스들(예: Token, YutGameRules 등)을 사용하여 게임 상태를 변경하고, 결과를 **View**에 전달하여 화면을 갱신한다. 이처럼 GameController가 중심이 되어 UI와 게임 로직 사이를 중재함으로써, 시스템 이벤트 처리가 한 곳에서 이루어지는 **컨트롤러 원칙**이 적용되었다.

==========================================================

**구현**

Swing에서 JavaFX로 UI 프레임워크가 전환되었지만, View 계층의 클래스 구조와 책임 분담은 거의 그대로 유지되었습니다. 각 파일의 핵심 로직(보드 그리기, 입력 대화, 상태 표시, 컨트롤러 연계)은 동일한 메서드와 흐름으로 구현되었고, 차이점은 Swing 컴포넌트를 JavaFX 컴포넌트로 대체한 문법적 수정과 약간의 구조 재배치뿐입니다. 예를 들어, 메서드 이름과 호출 방식(예: refresh(), showMessage(), selectToken() 등)이 그대로 유지되어 컨트롤러나 모델과의 연계 코드에 변경이 필요 없었고, 내부 구현만 JOptionPane → Alert 같은 대응 API로 교체되었습니다. BoardPanel/BoardView처럼 역할이 동일한 클래스를 사용하고, GameEndChoice 같은 설계 요소도 재사용함으로써 UI 교체로 인한 영향 범위를 최소화했습니다. 결국 MVC 구조의 View 부분이 프레임워크 변경에도 불구하고 동일한 책임을 수행하고 있으며, 코드 변경은 UI 라이브러리 문법상의 차이 외에는 동일한 수준입니다.

**1. View 계층 인터페이스의 일관성 유지**

**공통 메서드**

* refresh(), showMessage(String, String), selectToken(), selectPath() 등
* Controller는 View의 내부 구현(Swing/JFX)과 무관하게 항상 동일한 메서드 호출
* 이처럼 메서드 이름과 파라미터, 반환값을 전혀 변경하지 않고 Swing → JavaFX 간에도 그대로 쓰도록 함으로써, 뷰 교체 후에도 Controller 코드를 한 줄도 수정하지 않도록 할 수 있었다.

**이벤트 처리 흐름의 동일성**

* “랜덤 윷 던지기”와 “지정 윷 던지기” 버튼 클릭 시 Controller 호출 로직(rollingYut())
* Swing: rollButton.addActionListener(e -> controller.rollingYut());
* JavaFX: rollButton.setOnAction(e -> controller.rollingYut());
* 내부 API(addActionListener vs setOnAction)만 달라졌을 뿐, 버튼 클릭 시 컨트롤러 호출이라는 흐름 자체는 완전히 동일하다.
* 따라서 View 교체 시에도 Controller 쪽 코드는 전혀 손댈 필요 없이 View 레이어의 이벤트 연결부만 Swing→JavaFX 대응 메서드로 바꾸면 된다.

**2. 필수 변경점: UI 프레임워크 API 대응**

Swing과 JavaFX는 컴포넌트, 그래픽 처리, 다이얼로그, 레이아웃 처리 등 모든 면에서 서로 호환되지 않으므로, View 교체 과정에서는 아래 항목들을 반드시 변경해야 했다. 이 변경은 모두 프레임워크가 다르기 때문에 할 수밖에 없는 필수 작업이다.

|  |  |  |
| --- | --- | --- |
|  | **Swing** | **JavaFX** |
| **UI 컴포넌트 클래스** | javax.swing.JFrame  javax.swing.JPanel  javax.swing.JButton  javax.swing.JOptionPane | javafx.stage.Stage  javafx.scene.layout.Pane/BorderPane  javafx.scene.control.Button  javafx.scene.control.Alert  /javafx.scene.control.TextInputDialog |
| **그리기 방식** | protected void paintComponent(Graphics g)  g.drawLine(...), g.fillOval(...), g.drawOval(...)  repaint() 호출로 화면 갱신 | Canvas canvas = new Canvas(width, height);  GraphicsContext gc = canvas.getGraphicsContext2D();  gc.strokeLine(...), gc.fillOval(...), gc.strokeOval(...)  canvas 위에 그리는 후, clearRect() + 다시 그리기 방식으로 화면 갱신 |
| **다이얼로그 API** | JOptionPane.showMessageDialog(...)  JOptionPane.showInputDialog(...)  JOptionPane.showOptionDialog(...) | new Alert(AlertType.INFORMATION) / showAndWait()  new TextInputDialog() / showAndWait()  new ChoiceDialog<>(options) / showAndWait() |
| **이벤트 등록 방식** | button.addActionListener(e -> handler()); | button.setOnAction(e -> handler()); |
| **UI 스레드**  **관리 교체** | SwingUtilities.invokeLater(() -> | Platform.runLater(() -> |
| **레이아웃**  **매니저** | SwingUtilities.invokeLater(() -> | Platform.runLater(() -> |

* 위의 모든 변경점은 Swing 코드를 JavaFX로 그대로 옮길 수 없고, 반드시 대응 API를 호출해야만 동작하기 때문에 View 계층에서 필수적으로 수정된 부분이다

**3 변경 불필요 영역: Model / Controller**

1. **Model 클래스**

* 게임 규칙과 말 이동·잡기·업기·추가 턴 부여, 커스터마이징 로직을 모두 포함
* View 교체와 전혀 무관하게 순수한 비즈니스 로직만 수행
* 결과적으로 Swing → JavaFX 전환 시 단 한 줄도 수정할 필요가 없었다.

2. **Controller 클래스**

* View에게서 받은 이벤트(rollingYut(), selectToken() 호출 등)에 따라 Model을 호출
* 이동 후 view.refresh(), view.showMessage(), view.selectPath() 등 View 인터페이스 메서드를 호출
* View가 Swing이든 JavaFX든 상관없이 메서드 시그니처만 동일하면 그대로 동작
* 따라서 Controller 코드 또한 전혀 변경하지 않고 재사용할 수 있었다.

3. **결과: 전체 변경 범위 제한**

* Swing → JavaFX 전환 시, View 계층의 파일만 한정적으로 수정
* Model과 Controller 계층은 패키지 import를 제외하고는 **한 줄도 수정하지 않고** 그대로 빌드·실행 가능
* 이로써 UI만 교체해도 동작하는 구조를 달성했다.