**윷놀이 게임 설계 및 구현 리포트**

**1.시스템 개요**

**1.1 시스템 구조**

본 윷놀이 게임은 계층형 아키텍처를 기반으로 설계되어 UI 프레임워 크 변경에 대한 유연성을 극대화했습니다.

[Presentation Layer - UI]

├── YutGameUIInterface (인터페이스)

│ ├── YutGameUI (Swing 구현체)

│ └── YutGameJavaFXUI (JavaFX 구현체)

├── BoardPanelInterface (인터페이스)

│ ├── AbstractBoardPanel (추상 클래스)

│ │ ├── SwingBoardPanel (Swing 구현체)

│ │ └── JavaFXBoardPanel (JavaFX 구현체)

└── MainLauncher (UI 실행 관리)

[Business Logic Layer - 게임 로직]

├── GameController (컨트롤러)

├── Game (게임 상태 관리)

├── YutThrower (윷 던지기 로직)

└── YutThrowResult (윷 결과)

[Domain Layer - 도메인 모델]

├── Board (게임판)

├── BoardShape (게임판 형태)

├── Player (플레이어)

├── Piece (말)

└── Position (위치)

[Infrastructure Layer - 기반 시설]

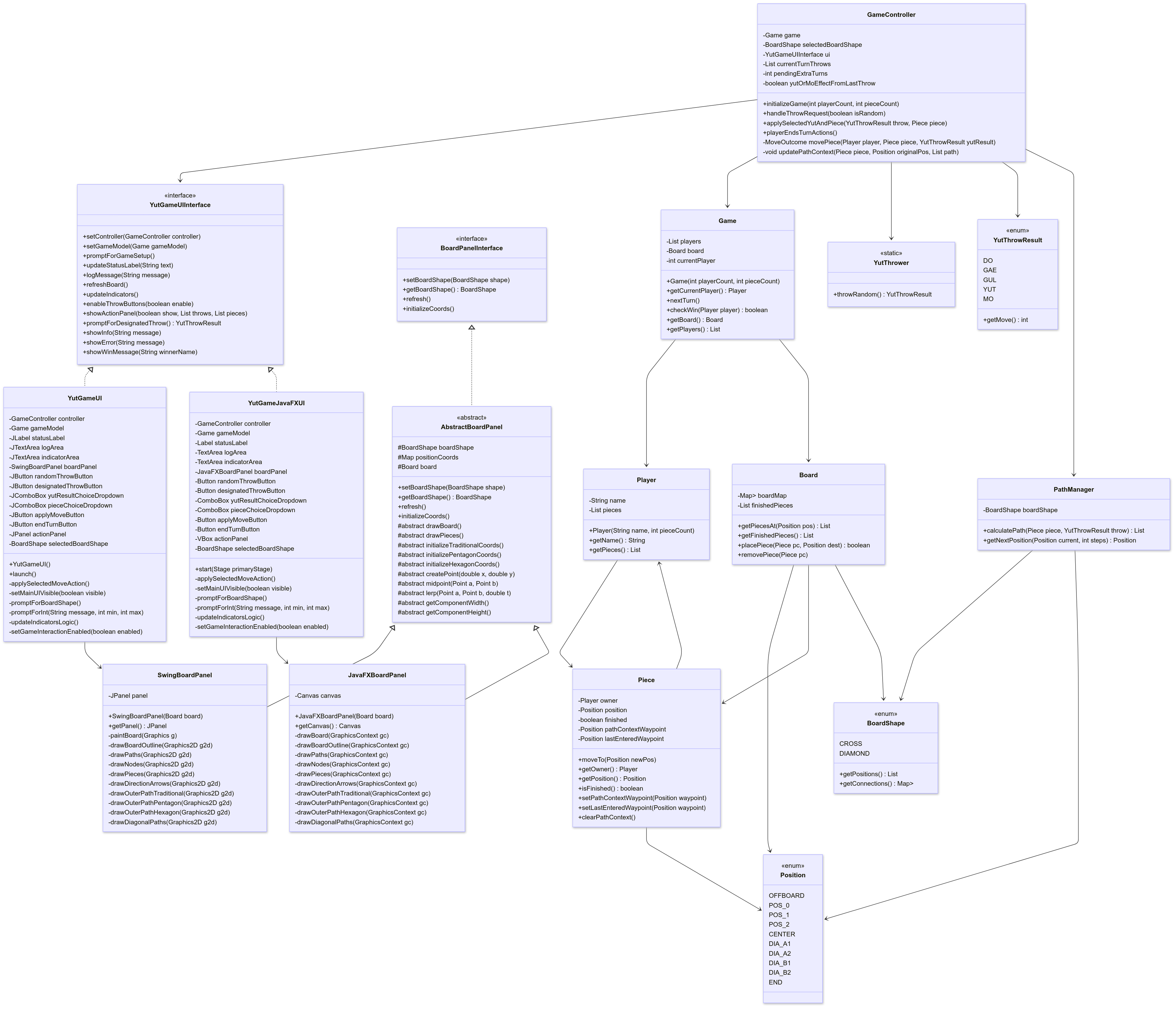
└── PathManager (경로 관리)

**1.2 핵심 설계 원칙**

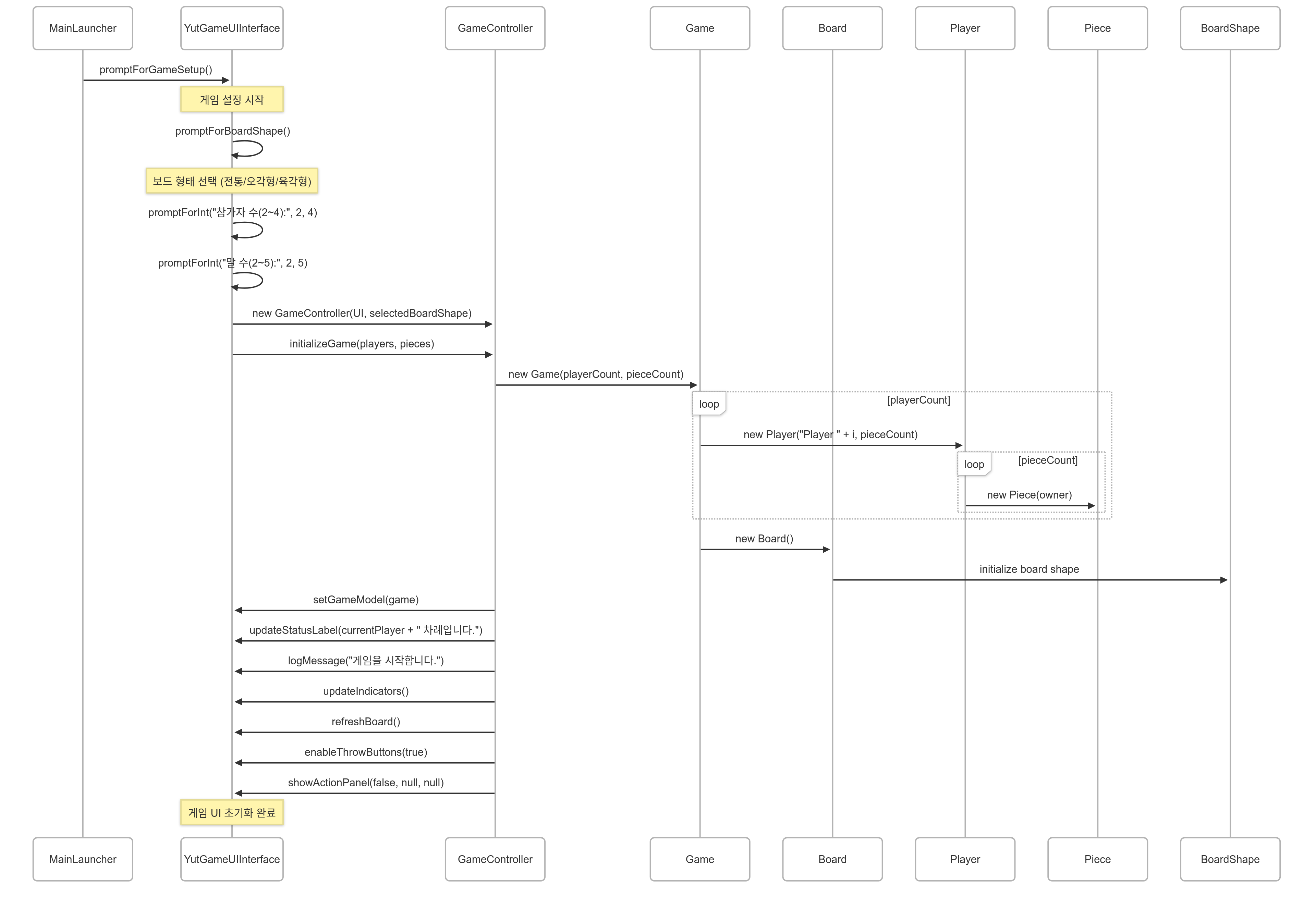
* **관심사 분리 (Separation of Concerns)**
  + UI 로직과 게임 로직의 완전한 분리
  + 각 계층의 독립적인 책임
  + 프레임워크별 UI 구현체 분
* **의존성 역전 원칙 (Dependency Inversion Principle)**
  + UI 컴포넌트들이 인터페이스에 의존
  + YutGameUIInterface 와 BoardPanelInterface를 통한 추상화
  + 구체적인 UI 구현체와의 결합도 최소화
* **인터페이스 분리 원칙 (Interface Segregation Principle)**
  + 각 인터페이스가 단일 책임을 가짐
* **개방-폐쇄 원칙 (Open-Closed Principle)**
  + 새로운 UI 프레임워크 추가 시 기존 코드 수정 불필요
  + AbstractBoardPanel을 통한 공통 기능 확장
  + 인터페이스 구현을 통한 새로운 기능 추가
* **단일 책임 원칙 (Single Responsibility Principle)**
  + PathManager: 경로 계산만 담당
  + YutThrower: 윷 던지기 로직만 담당
  + GameController: 게임 흐름 제어만 담당

**2.상세 설계**

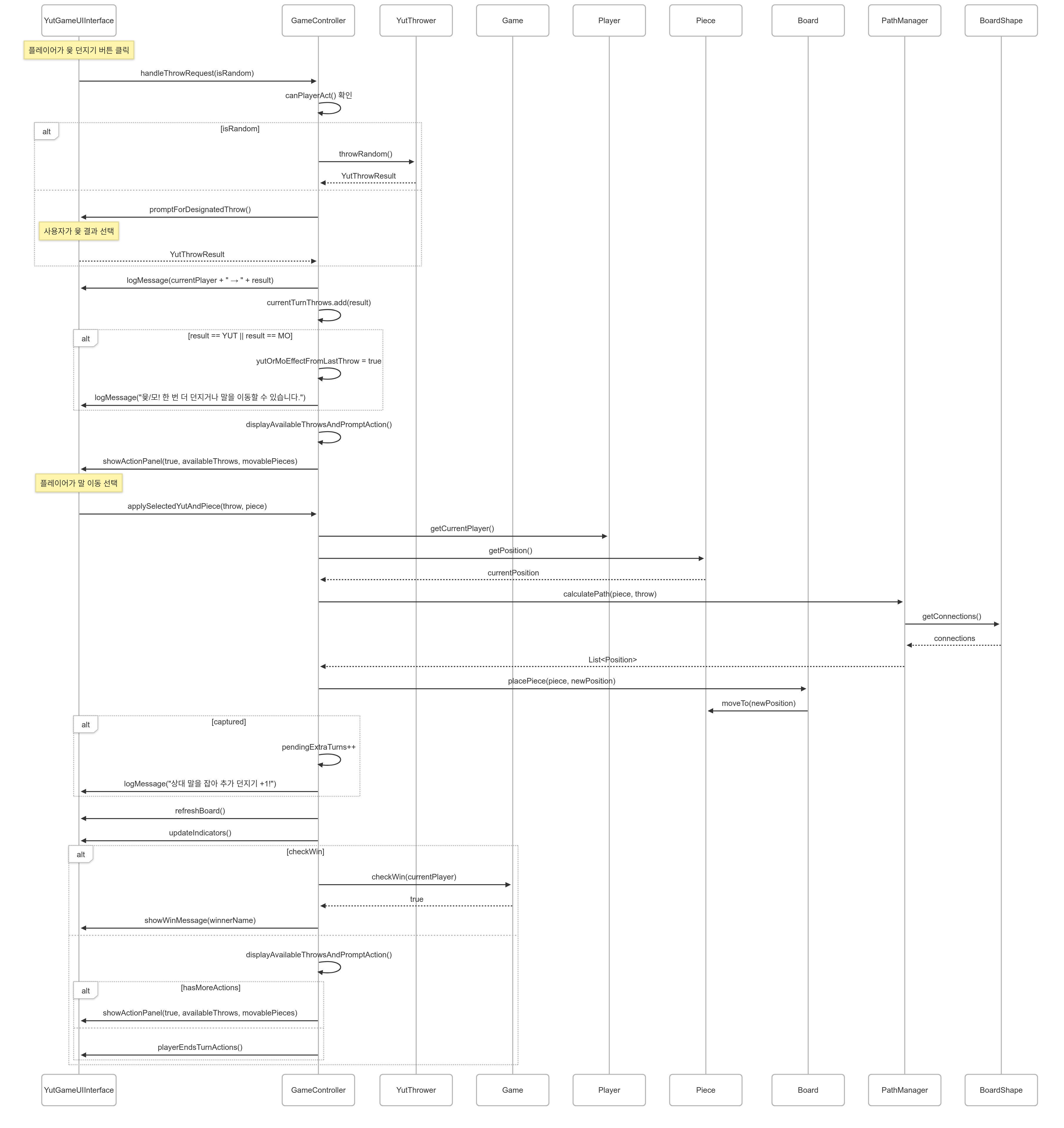
**2.1 Class Diagram**



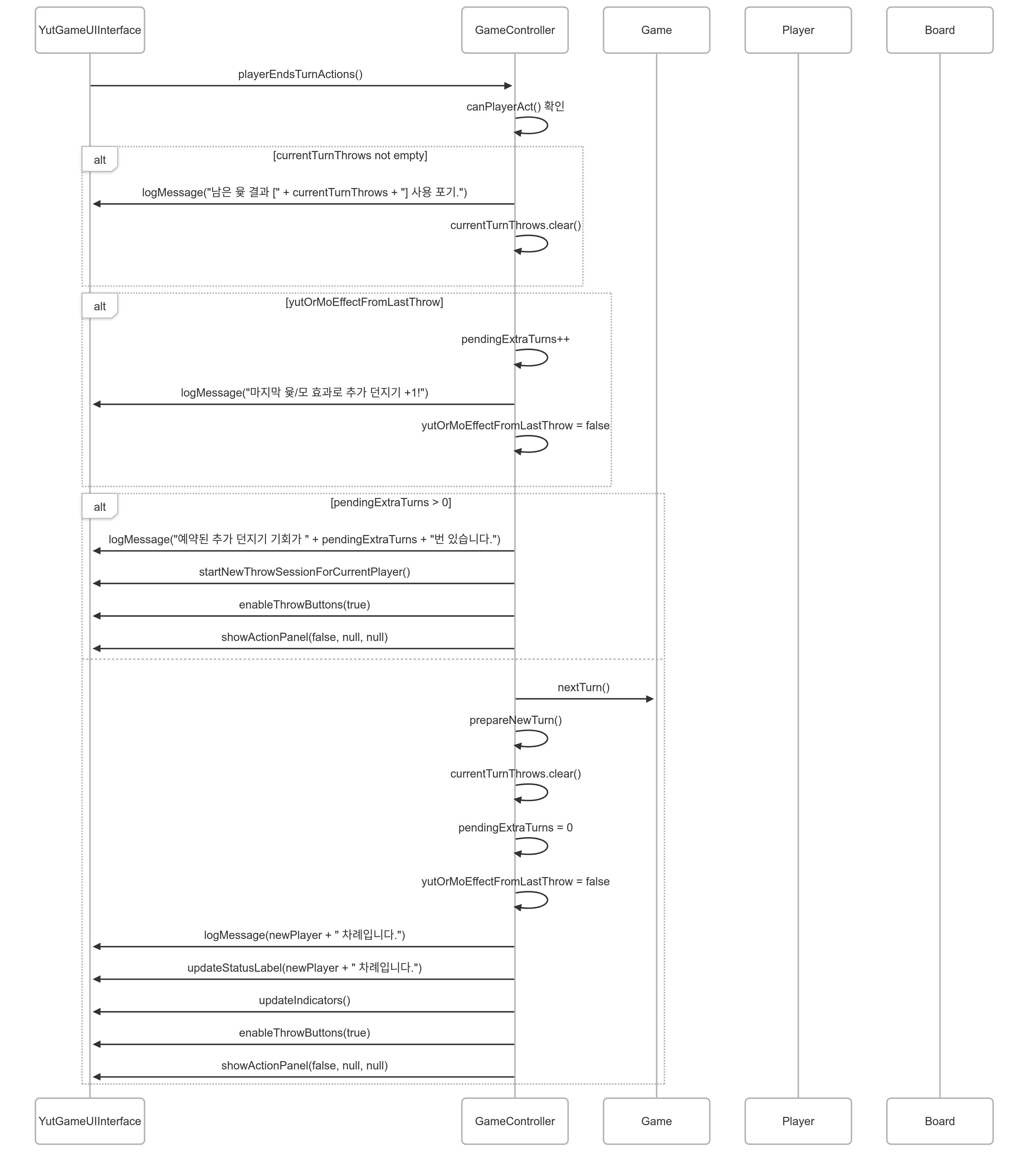
**2.2 Sequence Diagram - 게임 초기화**



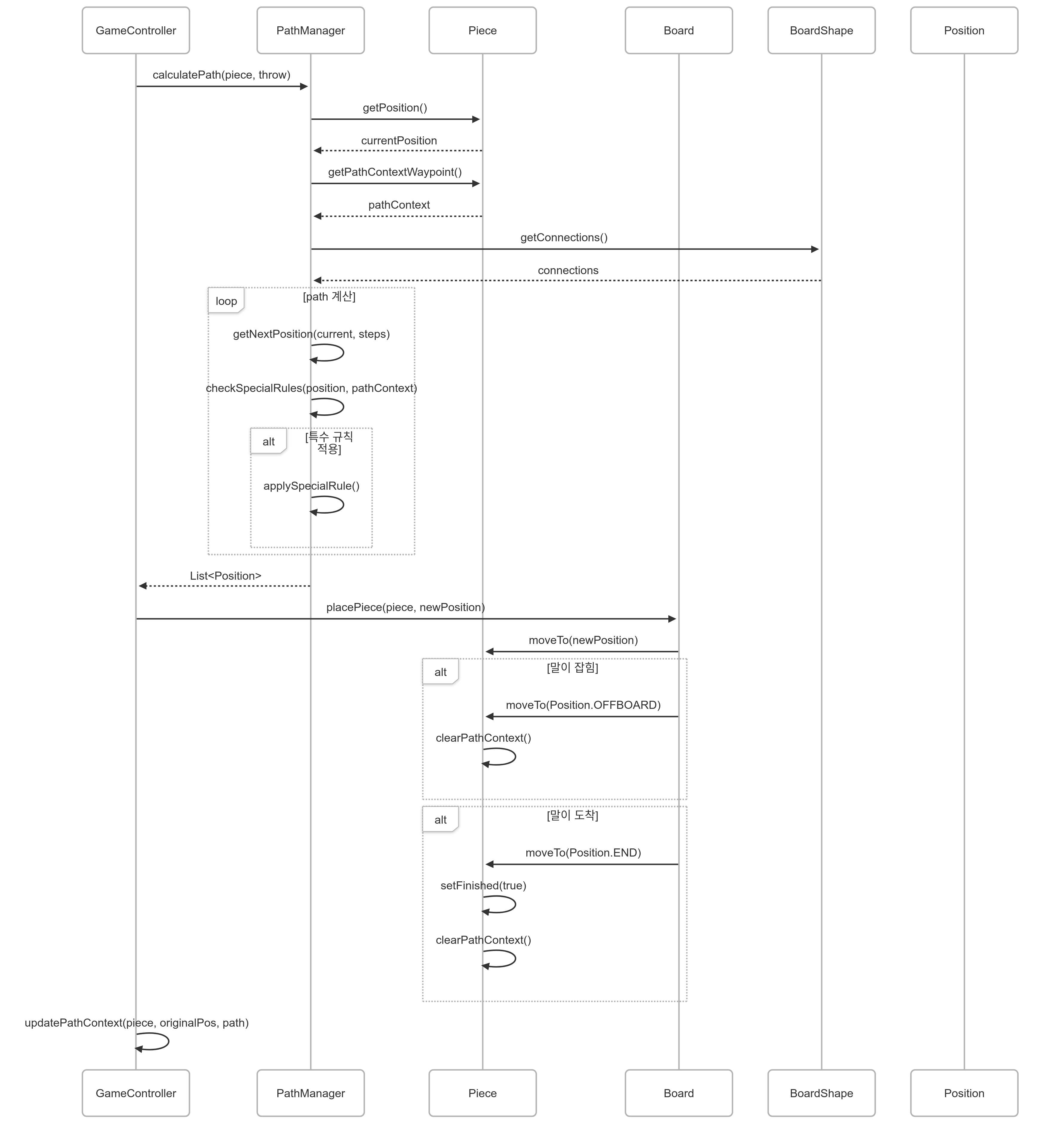
**2.3 Sequence Diagram - 윷 던지기 및 말 이동**



**2.4 Sequence Diagram - 턴 종료 및 다음 턴 시작**



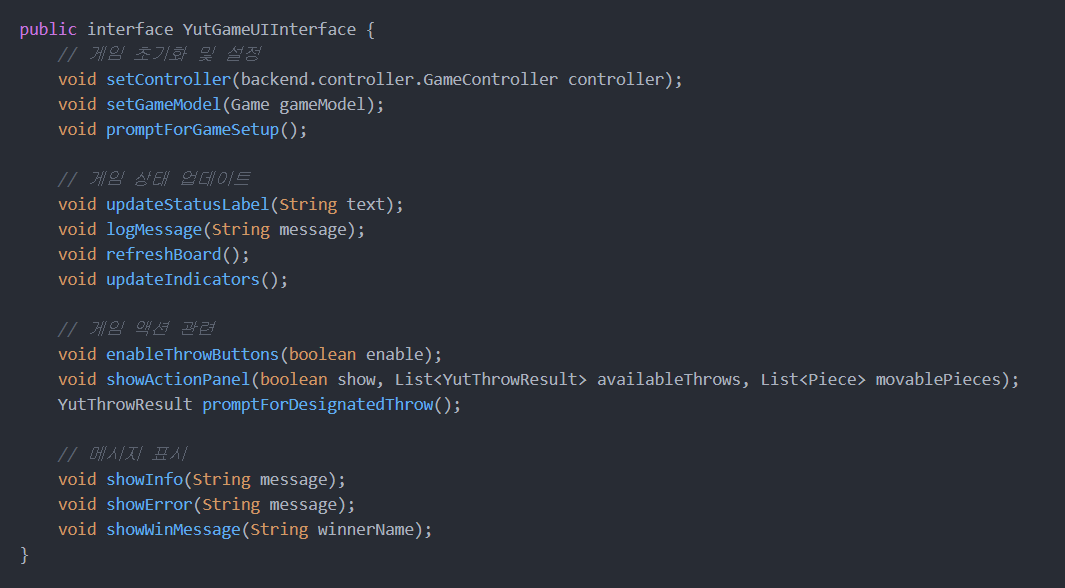
**2.5 Sequence Diagram - 말 이동 및 경로 계산**



**3. UI 교체 설계의 핵심**

**3.1 인터페이스 기반 추상화**

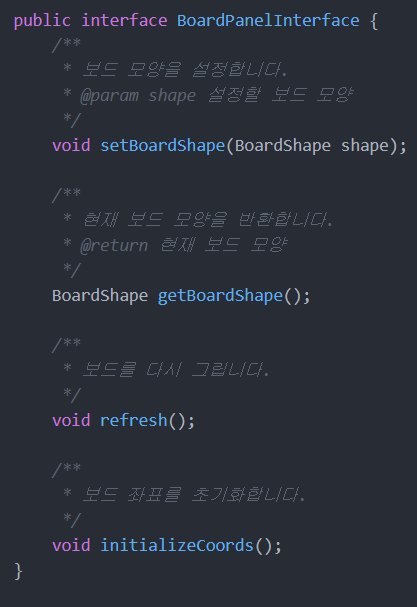
YutGameUIInterface는 UI 프레임워크에 독립적인 계약을 정의합니 다.



* UI 관련 모든 기능을 인터페이스로 정의
* 구체적인 UI 구현체와의 결합도 최소화
* 새로운 UI 프레임워크 추가 시 인터페이스만 구현하면 됨

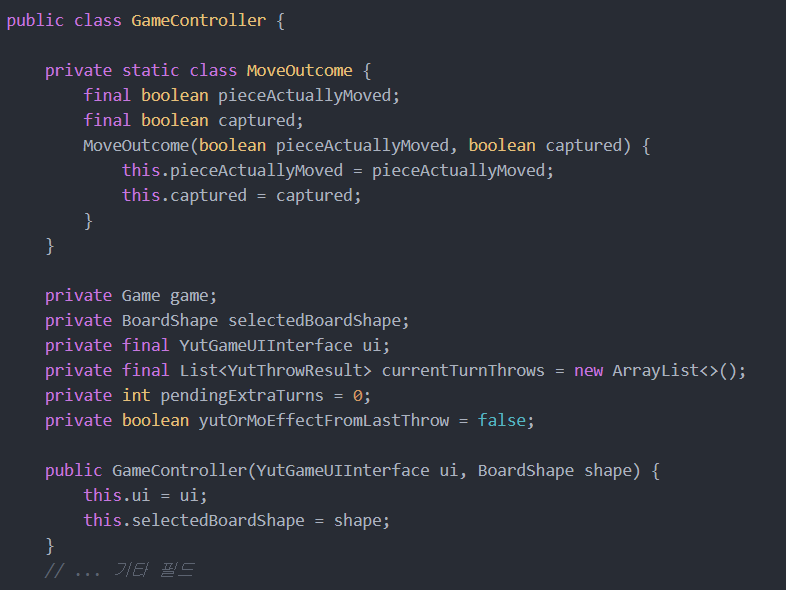
**3.2 보드 패널 추상화**

**BoardPanel** 추상 클래스는 UI 프레임워크별 구현체들의 공통 기능을 제공합니다.



* 게임 보드 렌더링 로직을 추상화
* 프레임워크별 구현체 분리 (Swing/JavaFX)
* 공통 기능은 추상 클래스에서 구현

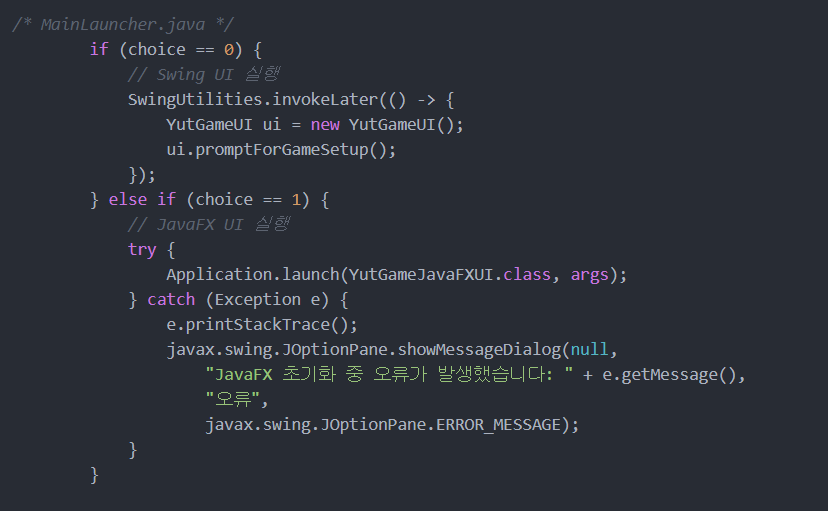
**3.3 컨트롤러 분리**



* UI 로직과 게임 로직의 완전한 분리
* 컨트롤러가 UI 인터페이스에만 의존

**4. UI 교체 시나리오 분석**

**4.1 Swing에서 JavaFX로 전환**



* 런타임에 UI 선택 가능
* 기존 코드 수정 없이 새로운 UI 추가

**4.2 새로운 UI 프레임워크 추가 시**

* YutGameUIInterface 구현
* BoardPanelInterface 구현
* AbstractBoardPanel 상속
* 기존 코드 수정 불필요

**5. 설계 패턴 적용**

**5.1 MVC (Model-View-Controller) 패턴**

* **Model**: 게임 도메인 모델 (Game, Board, Player, Piece)
* **View**: UI 구현체 (YutGameUI, YutGameJavaFXUI, ... )
* **Controller**: 게임 컨트롤러 (GameController)

**5.2 전략 패턴 (Strategy Pattern)**

* 다양한 보드 렌더링 전략 구현
* 런타임에 전략 교체 가능

**5.3 템플릿 메서드 패턴 (Template Method Pattern)**

* 공통 알고리즘 구조 정의
* 하위 클래스에서 구체적인 구현

**5.4 의존성 주입 패턴 (Dependency Injection)**

* 의존성을 외부에서 주입
* 결합도 감소 및 테스트 용이성 향상

**6. 설계의 이점**

**6.1 유지보수성 향상**

* UI 로직과 게임 로직의 명확한 분리
* 각 컴포넌트의 단일 책임 원칙 준수
* 코드 재사용성 향상

### **6.2 확장성 극대화**

* 새로운 UI 프레임워크 추가 용이
* 기존 코드 수정 최소화
* 인터페이스 기반 설계로 유연성 확보

### **6.3 테스성 증대**

* UI와 게임 로직의 독립적 테스트 가능
* 목(Mock) 객체를 통한 테스트 용이
* 단위 테스트 작성 용이

**7. 성능 및 메모리 최적화**

**7.1 메모리 관리**

* 효율적인 데이터 구조 사용
* 불필요한 객체 생성 최소화

**7.2 렌더링 최적화**

* 더블 버퍼링을 통한 화면 깜빡임 방지
* 부분 갱신을 통한 성능 향상

## **8. 향후 확장 가능성**

### **8.1 새로운 UI 프레임워크 지원**

* **Web UI**: React, Vue.js 기반 웹 인터페이스
* **Mobile UI**: Android, iOS 네이티브 앱
* **Desktop UI**: Electron, WPF 등

### **8.2 게임 기능 확장**

* **네트워크 멀티플레이**: 온라인 대전 기능
* **AI 플레이어**: 컴퓨터 상대방 추가
* **게임 기록**: 플레이 기록 저장 및 분석

### **8.3 접근성 개선**

* **다국어 지원**: i18n 인터페이스 추가
* **접근성 지원**: 시각/청각 장애인을 위한 UI
* **커스터마이징**: 사용자 정의 테마 및 설정

**9. 결론**

**9.1 설계의 성공 요인**

* 인터페이스 기반 추상화
* 명확한 계층 분리
* 확장 가능한 구조

**9.2 향후 개선 방향**

* 더 많은 UI 프레임워크 지원
* 성능 최적화
* 사용자 경험 개선

**9.3 프로젝트의 의의**

* 객체지향 설계 원칙의 실천
* 유지보수 가능한 코드 구조
* 확장 가능한 아키텍처

이러한 설계를 통해 UI 프레임워크 변경에 대한 유연성을 극대화하고, 향후 확장성과 유지보수성을 보장할 수 있도록 했습니다.