

Term Project

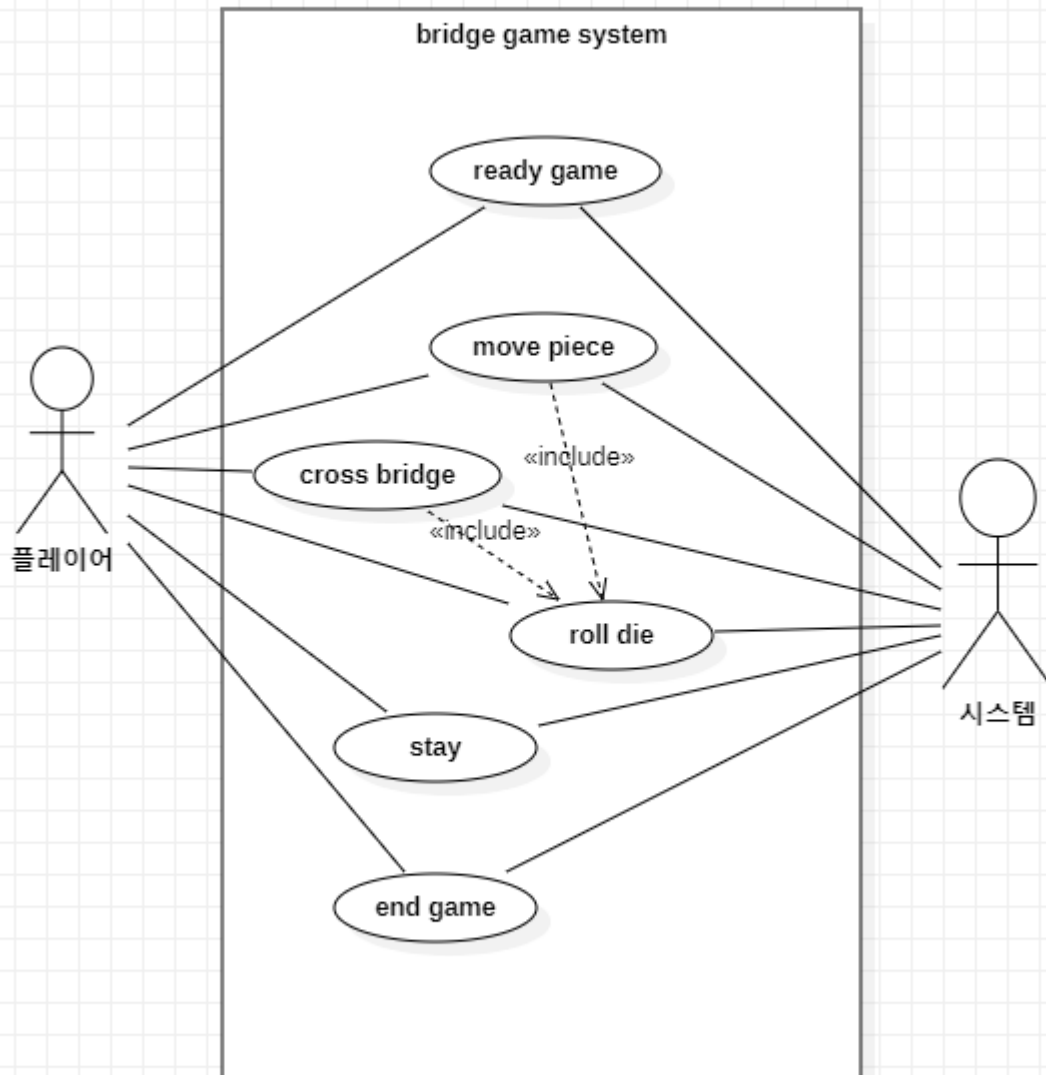
OOAD를 이용한 Bridge 게임 개발

소프트웨어공학 01분반

2022-06-10

1. 요구 정의 및 분석 산출물

(1) 유스케이스 다이어그램



위의 유스케이스 다이어그램은 bridge game system을 묘사하고 있습니다. 다이어그램에서 Actor는 human인 플레이어와 non human인 게임을 관장하는 시스템으로 구성되어 게임하면서 상호작용을 하고 있습니다. Bridge game system 내에는 너무 잘게 그리고 너무 크게 나누지 않으려고 노력한 유스케이스들이 있는데, 유스케이스 모두를 두 actor가 사용하는 모습을 확인할 수 있습니다. 그 중, 유스케이스 roll die는 cross bridge와 move piece가 공통으로 행동하는 유스케이스이므로 분리시켜 include 관계로 설정했습니다.

(2) 유스케이스 명세 4가지

● Use case 1: Move piece

Scope: system use case

Level: user-goal level

Primary Actor: 플레이어

Stakeholders and Interests:

- 플레이어: 점수를 최대한 벌 수 있도록, 특수 cell에 자신의 piece를 도달하고 싶어합니다. 또한 최대한 빨리 완주시키고 싶어합니다.
- 시스템: 유효한 방향 커맨드를 입력받고 싶어합니다.

Preconditions: 게임 준비를 마친 상태.

Postconditions: 플레이어의 piece가 의도한 cell에 위치한 상태.

Main Success Scenario:

1. 자신의 turn을 맞이한 게임 중인 플레이어가 자신의 piece를 움직이기 위해 주사위를 굴렀다.
2. 시스템은 해당 플레이어가 움직일 수 있는 수를 알려준다.
3. 게임 중인 플레이어는 자신의 piece를 움직일 방향 커맨드를 입력한다.
4. 시스템은 해당 piece를 커맨드대로 움직인다.
5. 시스템이 다음 turn으로 전환한다.

Extensions:

3a. 게임 중인 플레이어가 자신의 piece가 위치한 map의 cell에서 갈 수 없는 방향이 포함된 방향 커맨드를 입력한다.

➔ 시스템이 방향 커맨드를 다시 입력하게 한다.

➔ 올바른 방향 커맨드를 입력하면 piece move를 계속해서 진행합니다.

3b. 게임 중인 플레이어가 움직일 수 있는 수와 동일하지 않은 길이의 방향 커맨드를 입력한다.

➔ 시스템이 방향 커맨드를 다시 입력하게 한다.

➔ 올바른 길이의 방향 커맨드를 입력하면 piece move를 계속해서 진

행합니다.

3c. 어떤 플레이어의 piece가 완주하게 되면, board 위의 남은 piece들은 더 이상 뒤로 갈 수 없는데 뒤로 move하기 위한 방향 커맨드를 입력한다.

➔ 시스템이 방향 커맨드를 다시 입력하게 한다.

➔ 앞으로만 갈 수 있는 move의 방향 커맨드를 입력하면 piece move를 계속해서 진행합니다.

● Use case 2: Cross bridge

Scope: system use case

Level: user-goal level

Primary Actor: 플레이어

Stakeholders and Interests:

- 플레이어: 점수를 최대한 벌 수 있도록, 특수 cell에 자신의 piece를 도달하고 싶어합니다. 또한 최대한 빨리 완주시키고 싶어합니다.
- 시스템: 해당 플레이어의 piece가 bridge를 건널 수 있는 cell에 있는지 정확히 알고 싶어합니다. piece의 다음 move를 플레이어의 의도대로 정확히 준비하고 싶어합니다.

Preconditions: 게임 준비를 마친 상태이며 bridge를 건널 수 있는 cell에 piece가 위치한 상태.

Postconditions: 플레이어의 piece가 의도한대로 bridge를 건널 수 있도록 앞으로 move하게 되면 bridge의 건너 cell에 위치하도록 준비한 상태.

Main Success Scenario:

1. 자신의 turn을 맞이한 게임 중인 플레이어가 자신의 piece를 움직이기 위해 주사위를 굴렀다.
2. 시스템은 해당 플레이어의 piece가 bridge를 건널 수 있다고 알린다.
3. 게임 중인 플레이어는 bridge를 건너겠다고 한다.
4. 시스템은 해당 piece가 앞으로 move하게 되면 bridge의 건너 cell로 나아가도록 한다.

5. 시스템이 move piece를 한다.

Extensions:

3a. 게임 중인 플레이어가 bridge를 건너지 않고 원래 길로 가겠다고 한다.

➔ 시스템은 해당 piece의 다음 move를 특별히 다르게 설정하지 않고 move piece를 이어서 진행합니다.

● **Use case 3: End game**

Scope: system use case

Level: user-goal level

Primary Actor: 플레이어

Stakeholders and Interests:

- 플레이어: 다른 플레이어들보다 자신의 piece를 먼저 완주시켜 가장 높은 완주점수를 벌고 싶어합니다. 게임에서 우승하고 싶어합니다. 자신의 점수를 알고 싶어합니다.
- 시스템: 정확한 점수 계산을 하고 싶어합니다. Piece의 완주 상태를 정확히 기록하고 싶어합니다.

Preconditions: 플레이어는 어느 정도의 piece move와 여러 번의 turn 획득이 반복된 상태.

Postconditions: 플레이어의 piece가 완주되어 점수가 정상적으로 계산되었으며 저장된 상태.

Main Success Scenario:

1. 마지막에서 두 번째로 남은 플레이어의 piece가 완주했다.
2. 시스템은 완주한 piece 및 완주하지 못한 piece의 점수도 계산하여 저장한다.
3. 시스템은 점수를 토대로 우승자가 누구인지 알려준다.

Extensions:

1a. 아직 게임을 완주하지 못한 piece가 board 위에 두 개 이상 남아있다.

➔ Board 위에 하나의 piece만 남을 때까지 piece move와 turn 전환을 반복합니다.

● **Use case 4:** Ready game

Scope: system use case

Level: user-goal level

Primary Actor: 플레이어

Stakeholders and Interests:

- 플레이어: 시작하는 게임을 이기고 싶어합니다.
- 시스템: 플레이어의 수를 정확히 반영하고 싶어합니다. 정상적으로 map을 로드하고 싶어합니다.

Preconditions: 게임 시작 버튼이 포함된 화면을 보여준 상태..

Postconditions: 게임이 시작된 상태.

Main Success Scenario:

1. 플레이어가 게임 시작 버튼을 누른다.
2. 플레이어가 함께 플레이할 플레이어의 수를 입력한다.
3. 시스템은 입력된 수 만큼 piece를 준비하여 map과 함께 보여주면서 게임 시작 준비를 마친다.

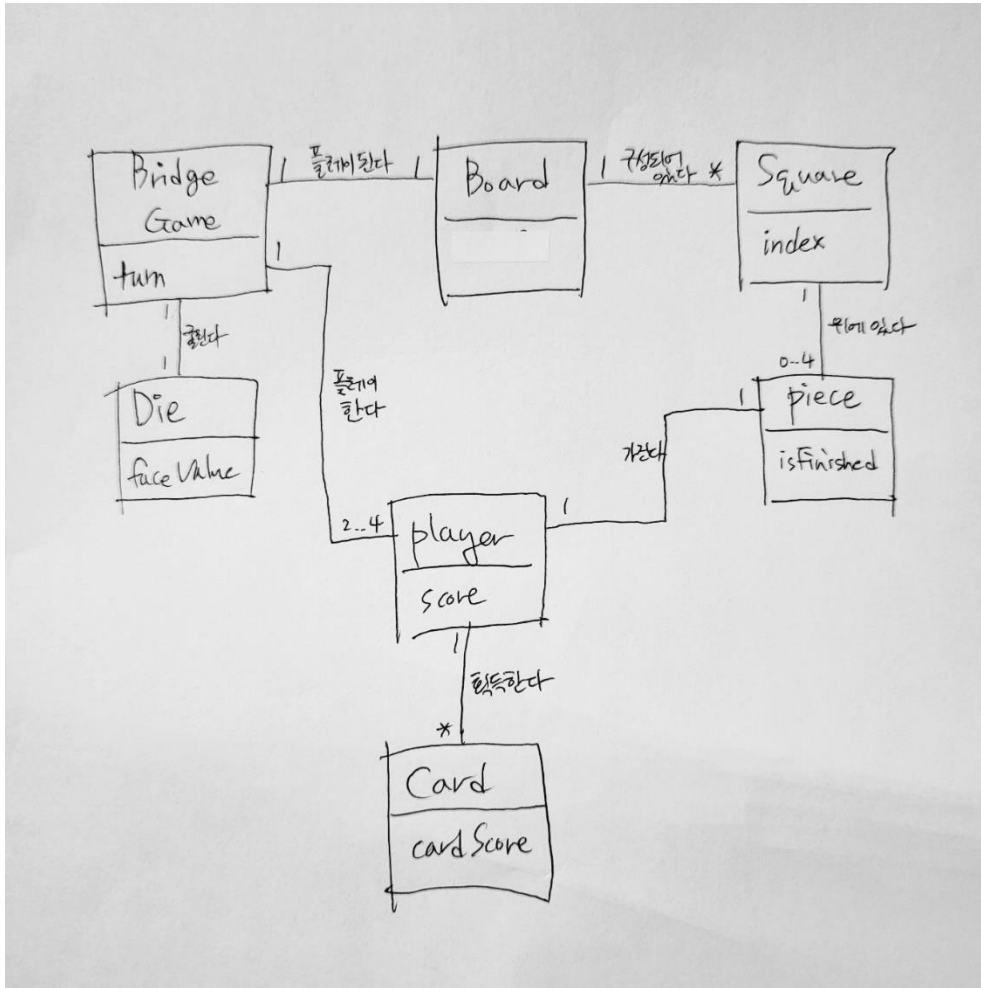
Extensions:

2a. 플레이어가 플레이어의 수를 2 ~ 4의 범위를 벗어난 수를 입력한다.

➔ 시스템이 플레이어의 수를 다시 입력하게 한다.

➔ 게임 준비를 이어서 진행합니다.

(3) 도메인 모델

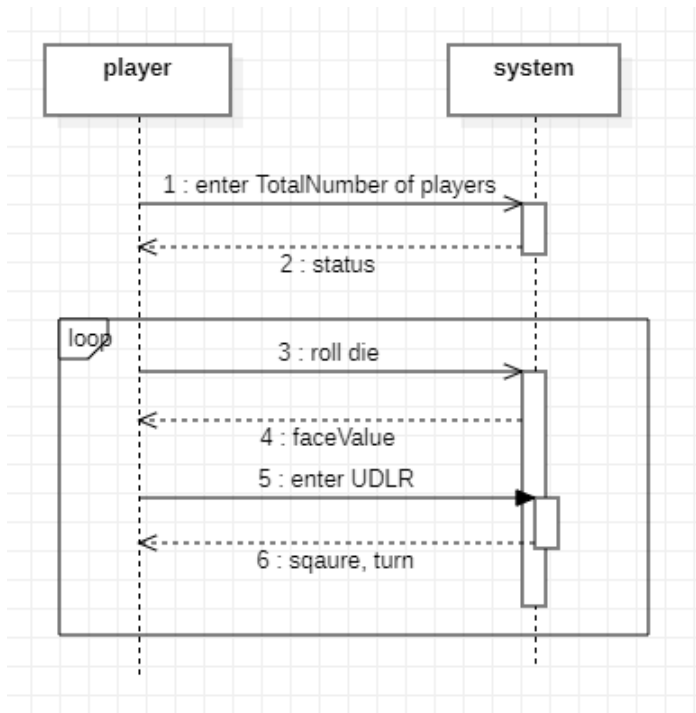


도메인 모델은 요구 정의 및 분석하는 단계에서 클래스 다이어그램의 원천이 될 수 있는 모델입니다. 강의에서 언급하신 모노폴리라는 이미 존재하는 도메인 모델을 참고하여 재사용했으며, 주어진 요구와 앞에서 작성한 유스케이스의 success scenario를 토대로 개념적인 클래스들과 현실의 객체들을 시각화할 수 있도록 중요한 것들 위주로 도메인 모델을 작성했습니다.

각 domain object들 사이의 association에 대해 설명하겠습니다. bridge game은 2~4명의 player가 플레이합니다. 또한 하나의 board 위에서 플레이 됩니다. 그리고 하나의 Die를 굴리며 진행됩니다. Board는 여러 개의 square들로 구성되어있으며 이 square위에 player의 piece가 존재할 수 있습니다. 또한 player는 게임 중에 여러 개의 card를 획득하게 됩니다.

각 domain object들의 attribute에 대해 설명하겠습니다. bridge game은 attribute로 turn, die는 faceValue, player는 score, card는 cardScore, square는 index 그리고 마지막으로 piece는 isFinished 라는 attribute를 가집니다.

(4) System Sequence Diagram



시스템이 받아야하는 외부 입력이 무엇인지 알아보기 위해 작성한 system sequence diagram입니다. 시스템은 actor로부터 플레이어의 수, 주사위를 굴릴 것인지, stay할 것인지, 그리고 방향 커맨드를 입력받는다는 것을 확인할 수 있습니다.

(5) Operation Contract

Operation: enter UDLR(command: String)

Cross References: Use cases: bridge game system

Preconditons: 움직일 수 있는 칸 수를 알고 있는 상태.

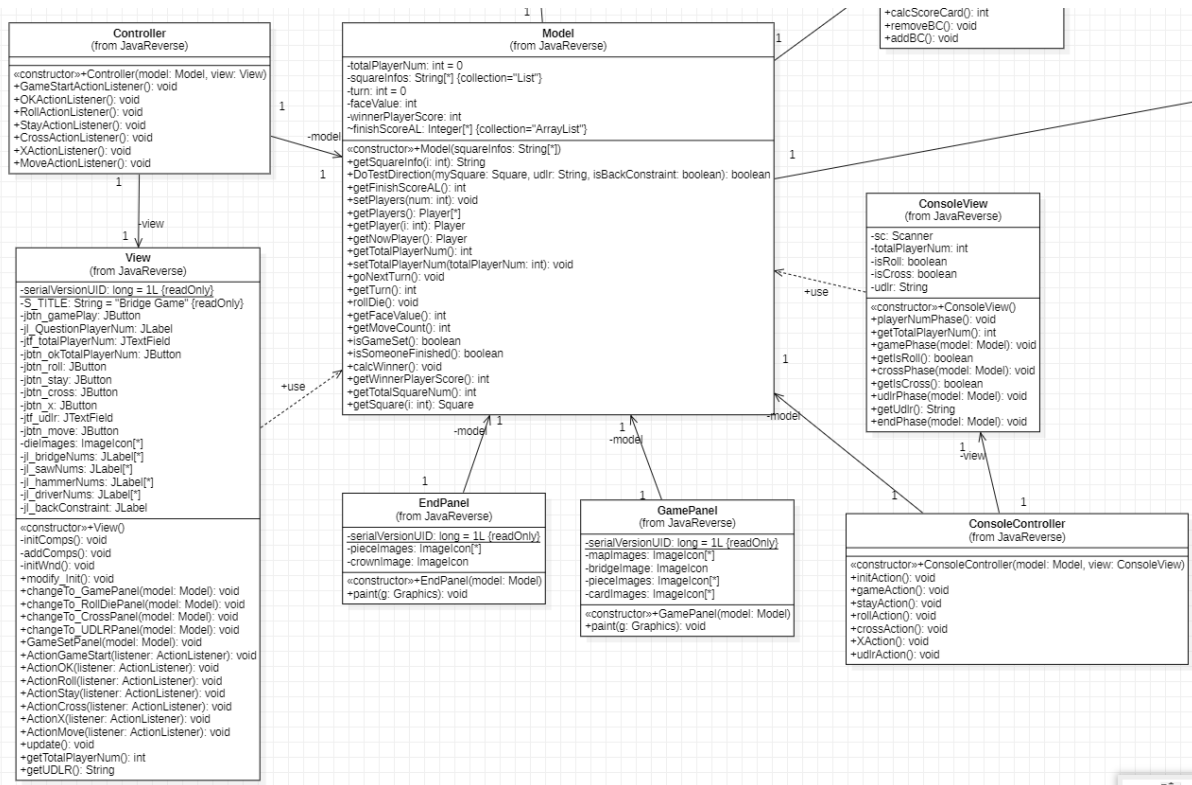
Postcondtions: 플레이어의 piece의 square index가 시스템으로부터 받은 square로 정정된다. Bridge game의 turn이 정정된다. 만약 특수 cell에 piece가 위치한다면 플레이어의 card instance가 생성되어 추가된다. 만약 bridge cell을 밟는다면 다음 turn에 플레이어 piece가 위치한 square가 isCross가 true로 정정된다.

Operation Contract는 도메인 모델에서 언급하지 못한 특수 cell과 card에 대한 내용을 추가하므로써 조금 더 이해를 도와줍니다. 또한 시스템 작동의 결과로써 객체의 변화가 어떻게 진행되는지 상세히 파악할 수 있습니다.

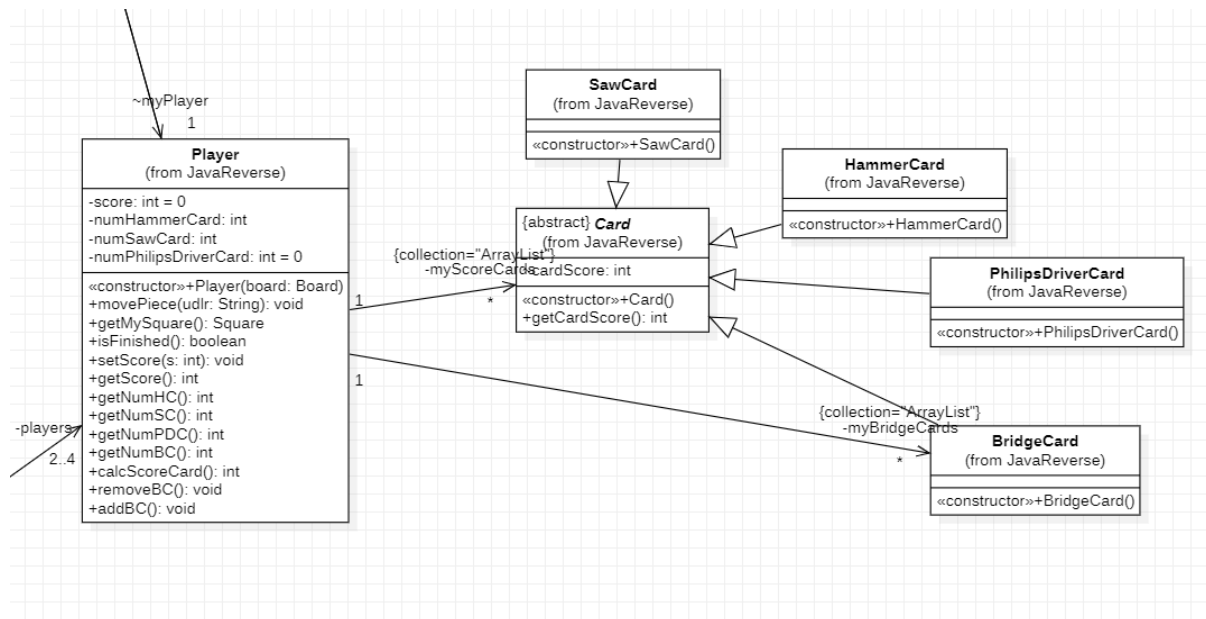
2. 설계 산출물

(1) Class Diagram

다이어그램이 거대하다 보니 가시성이 확보되지 않아, 부분들을 확대하여 캡처 후 작성하였음을 알립니다.



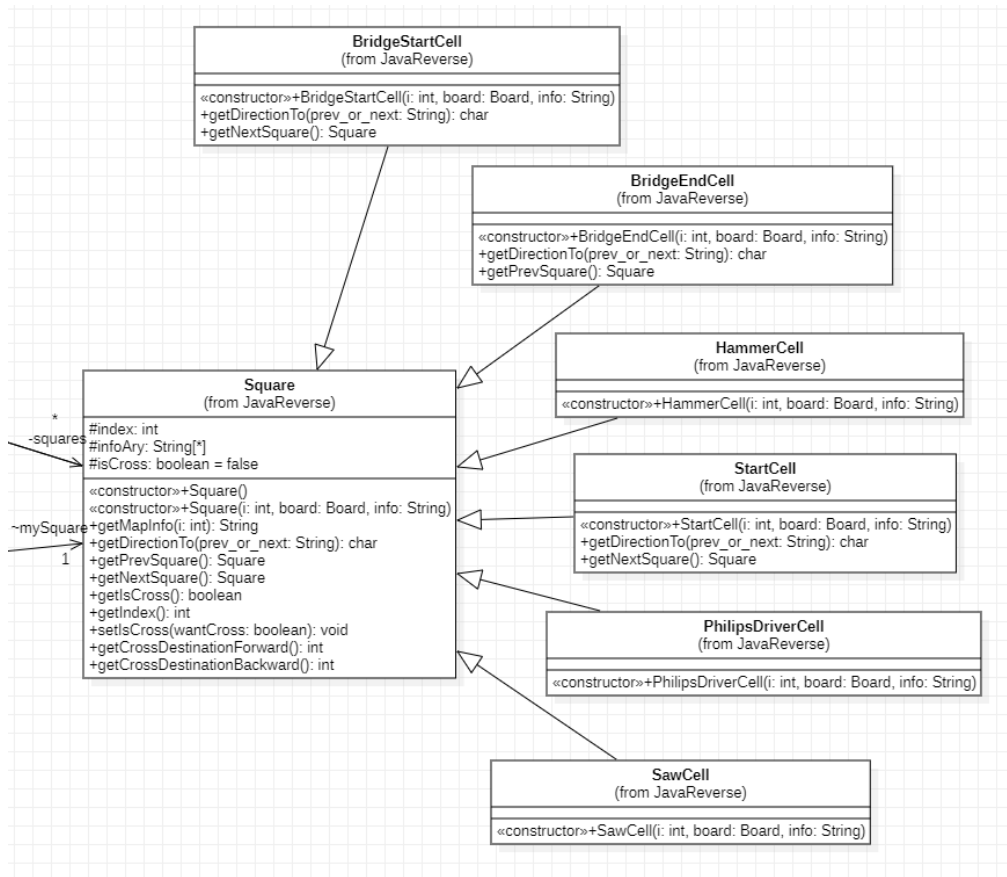
먼저, MVC 설계에 해당하는 클래스들을 보여주는 class diagram의 부분입니다. Non-UI인 model 클래스가 전체 game을 관장하는 시스템의 역할을 하고 있습니다. View와 consoleView는 model에서 데이터를 참고해서 UI를 표시하기 위해 model을 잠깐 사용하므로 의존 관계를 가지는 모습을 볼 수 있습니다. Controller는 view와 model을 모두 연관 관계를 가지는데, 이는 controller에서 이벤트를 처리하는 것은 물론, view에서 받아온 user input과 model의 데이터를 이용해 연산을 하고 model에 다시 변화된 값을 get/set해주기 때문입니다.



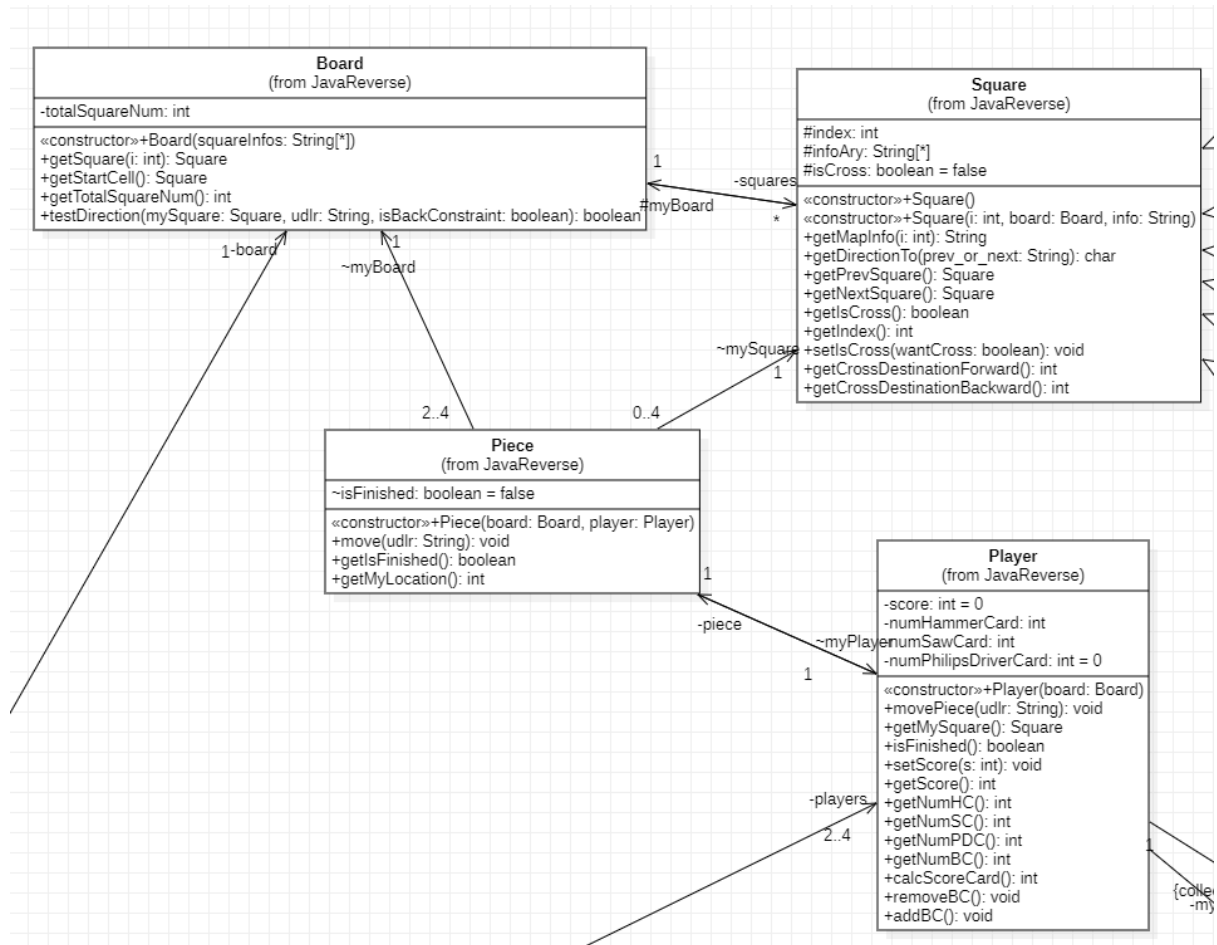
다음으로, player 클래스와 해당 플레이어가 획득한 card를 표현하는 card 클래스 및 해당 클래스로 일반화가 가능한 4개의 종류의 card 클래스들이 보여집니다. 이를 일반화 관계를 뜻하는 표기법을 보아 확인할 수 있습니다.

Card를 abstract 클래스로 설계함으로써 여러 번 작성해야할 뻔 했던 정보들을 모아서 한 번만 작성할 수 있고, 수정할 때도 추상 클래스에서만 수정할 수 있어서 관리를 보다 쉽게 할 수 있을 뿐만 아니라 card의 모든 것을 재사용할 수 있다는 장점이 있습니다.

또한 플레이어는 카드를 여러 개 가지므로 1 - * 의 연관관계를 가집니다.



Square 클래스도 마찬가지로, abstract 클래스는 아니지만, 일반화 관계임을 나타내는 표기법을 확인할 수 있습니다. 상속개념을 사용함으로써 재사용, 쉬운 관리 그리고 확장성이라는 장점을 가집니다.



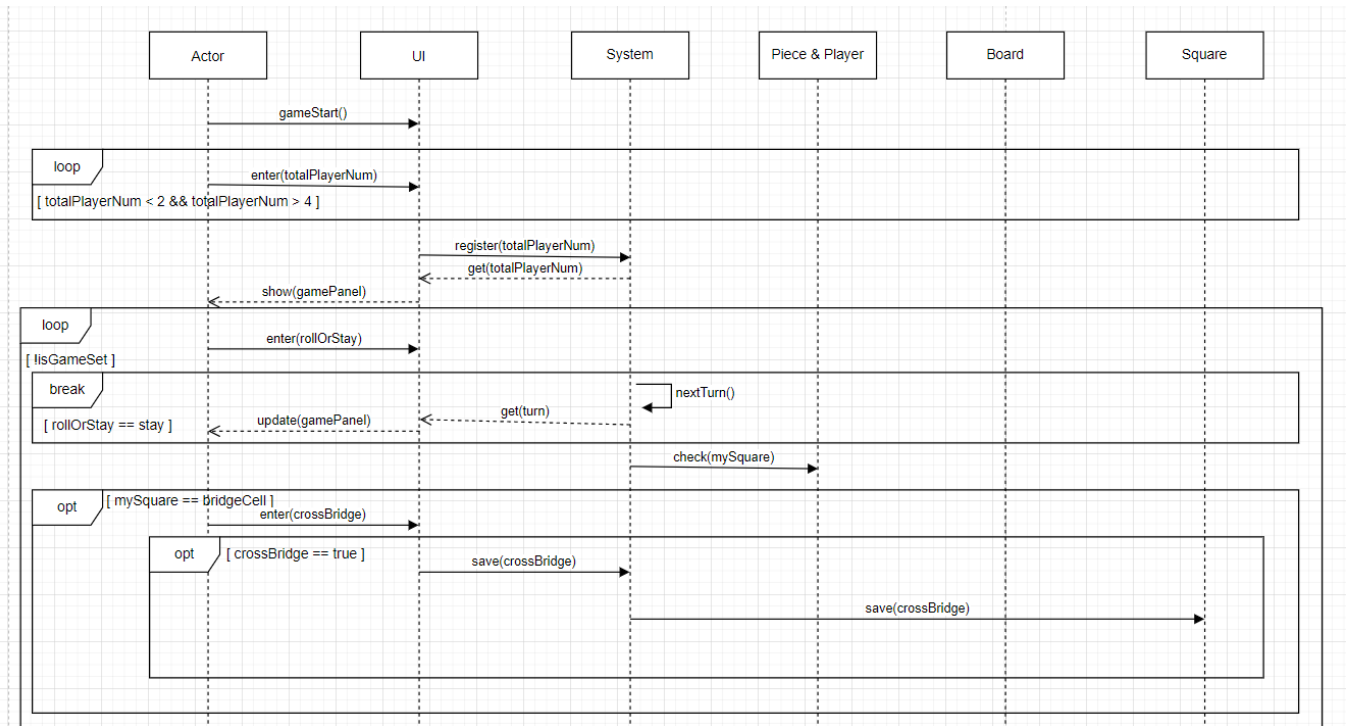
다음으로, 주요 클래스들의 관계를 살펴보겠습니다.

게임은 2명에서 4명 사이의 플레이어만 사용가능한 것을 확인할 수 있습니다. 그리고 이 플레이어는 각각 하나씩의 piece를 가지는 것을 연관 관계으로 보여줍니다.

board에는 플레이어와 마찬가지로 2~4개의 piece만 올라갈 수 있습니다. 그리고 board는 여러 개의 square(=cell)로 구성됩니다. 이 각각의 square 위에는 0~4개 사이의 piece가 올라갈 수 있다는 것을 위의 클래스 다이어그램의 연관 관계에서 알 수 있습니다.

(2) Sequence Diagram

먼저, 마찬가지로 다이어그램이 커서 글자가 잘 안보이는 관계로 부분들로 잘라서 보여드리는 점 양해 부탁드립니다.

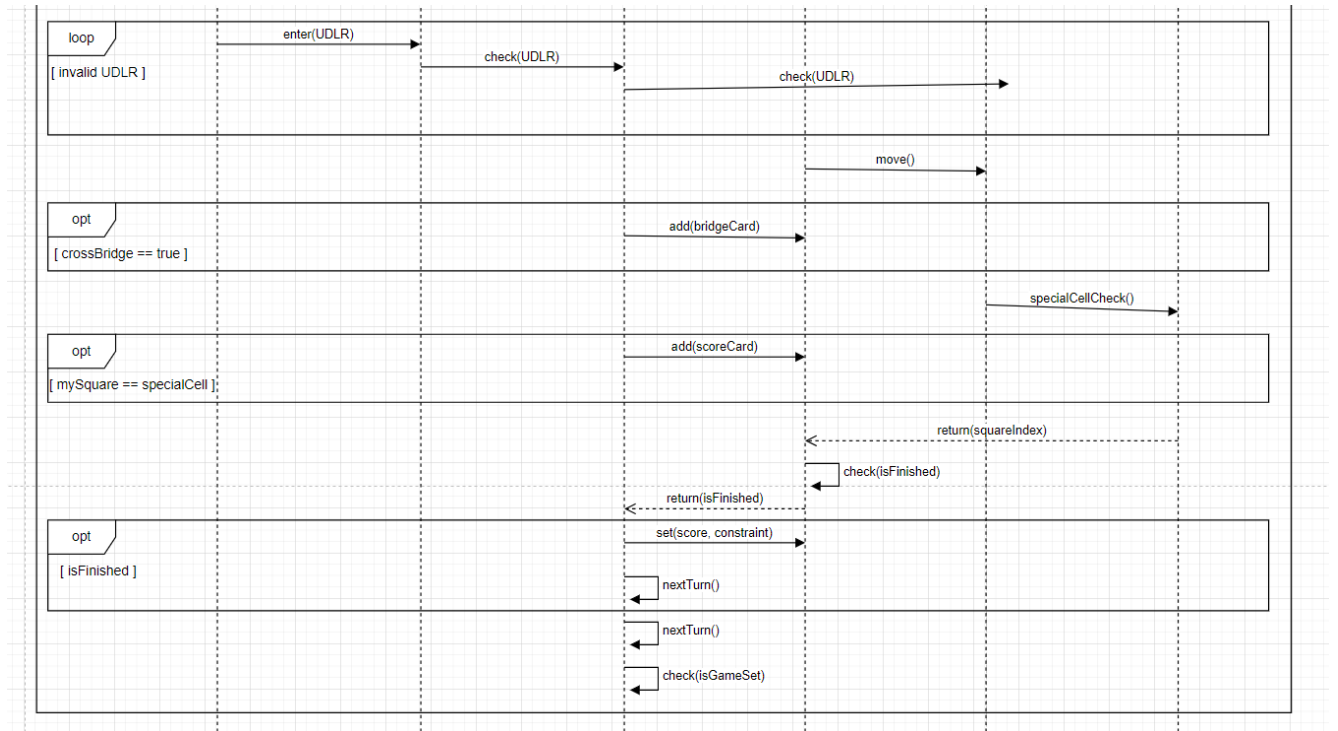


첫 번째 부분입니다.

Actor는 UI를 통해 게임을 시작합니다. 그리고 총 플레이어 수를 입력받는데, 이때 설정된 범위 외의 입력을 받게 되면 반복해서 입력을 요청합니다. 범위 내 입력을 받으면, 시스템에 해당 입력 값을 저장하고 이를 반영한 game panel을 actor에게 보여줍니다.

게임은 isGameSet이 true로 바뀌기 전까지 반복합니다. Actor는 roll 할건지, stay 할건지 결정하여 UI에 입력합니다. 만약 stay를 선택했다면, turn을 다음으로 넘긴 뒤, game panel을 업데이트 하고 break 아래를 수행하지 않은 채 반복문을 타고 다시 roll 할건지, stay 할건지 물어보는 페이지로 도달합니다.

Roll을 선택했다면 지금 자신의 square가 무슨 cell 위에 있는지 확인합니다. 만약 bridge cell 위에 있다면 cross 할 것인지 아닌지를 입력합니다. 만약, cross를 선택했다면 그 선택을 저장한 뒤 게임을 이어 진행합니다.



두 번째 부분입니다.

Roll을 선택했고, 그 다음 방향 커맨드를 입력받습니다. 입력받은 UDLR은 유효한지 검사하여 유효한 입력을 받을 때까지 반복하여 입력하도록 유도합니다.

만약 유효한 입력을 했다면, piece를 move합니다.

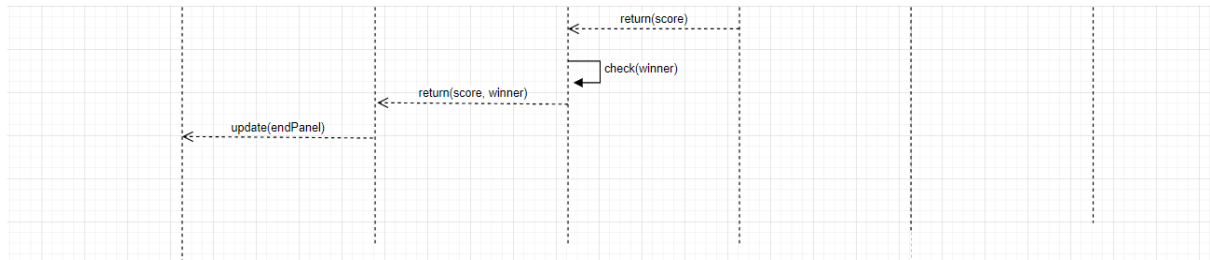
그 후, 이전에 bridge를 cross하기로 결정했었다면, bridge card를 받게 됩니다.

그리고 도달한 square가 특별한 cell인지 확인합니다. 만약 특별한 cell이라면 scoreCard를 획득하게 됩니다.

그 다음, 도달한 square의 index를 확인하여 해당 piece가 완주했는지 검사합니다. 만약 완주했다면, 점수를 설정하고 더 이상 다른 piece들이 뒤로 가지 못하게 제약을 걸어줍니다. 그리고 turn이 더 이상 해당 플레이어에게 오지 못하도록 처리해줍니다.

Move가 마무리되면 다음 플레이어에게 turn을 넘겨줍니다.

또한 게임이 끝났는지 검사합니다.



마지막 부분입니다.

진행되던 게임이 끝나서 반복문을 빠져나왔다면, score와 winner를 알아낸 뒤, UI를 업데이트 하고 해당 정보들을 actor에게 보여주면서 게임을 종료합니다.

3. 구현 산출물

(1) 소스코드 및 실행화일

먼저, 모든 소스코드를 문서에 담기보다 프로그램의 설계를 엿볼 수 있는 핵심적인 코드만 캡처하여 담았음을 알립니다.

```
import java.io.IOException;

public class Main {
    public static void main(String[] args) throws IOException {
        Scanner sc = new Scanner(System.in);
        List<String> squareInfos;

        squareInfos = Files.readAllLines(Paths.get("default.map"));

        System.out.print("Do you want to play in GUI? (Y/N) : ");
        String strValue = sc.next();
        if (strValue.equals("y") || strValue.equals("Y")) {
            View view = new View();
            Model model = new Model(squareInfos);
            new Controller(model, view);
        } else {
            ConsoleView view = new ConsoleView();
            Model model = new Model(squareInfos);
            new ConsoleController(model, view);
        }
    }
}
```

이것은 메인함수가 있는 클래스입니다. 프로그램이 시작되면 default.map 파일을 디폴트로 로드하고, GUI 사용 여부를 입력받습니다.

이 프로그램은 UI와 응용 로직을 분리하여 MVC 아키텍처를 활용한 설계 및 구현을 했기 때문에, UI 부분의 View와 controller가 이벤트 핸들러로 서로 상호작용을 하며 controller는 model의 데이터를 get/set하면서 상호작용을 하지만, model은 UI부분을 알 수 없는 구조입니다. 이와 같은 내용은 위의 메인함수 클래스에서 확인할 수 있습니다.


```
public class View extends JFrame {
```

```
    for (int i = 0; i < model.getTotalPlayerNum(); i++) {
        jl_bridgeNums[i] = new JLabel(String.valueOf(model.getPlayer(i).getNumBC()));
        jl_bridgeNums[i].setBounds(this.getWidth() - 375, 150 + y, 50, 50);
        jp_game.add(jl_bridgeNums[i]);
        jl_sawNums[i] = new JLabel(String.valueOf(model.getPlayer(i).getNumSC()));
        jl_sawNums[i].setBounds(this.getWidth() - 275, 150 + y, 50, 50);
        jp_game.add(jl_sawNums[i]);
        jl_hammerNums[i] = new JLabel(String.valueOf(model.getPlayer(i).getNumHC()));
        jl_hammerNums[i].setBounds(this.getWidth() - 175, 150 + y, 50, 50);
        jp_game.add(jl_hammerNums[i]);
        jl_driverNums[i] = new JLabel(String.valueOf(model.getPlayer(i).getNumPDC()));
        jl_driverNums[i].setBounds(this.getWidth() - 75, 150 + y, 50, 50);
        jp_game.add(jl_driverNums[i]);

        y += 100;
    }

    jbtn_roll.setBounds(this.getWidth() - 500, this.getHeight() - 100, 200, 50);
    jbtn_stay.setBounds(this.getWidth() - 250, this.getHeight() - 100, 200, 50);

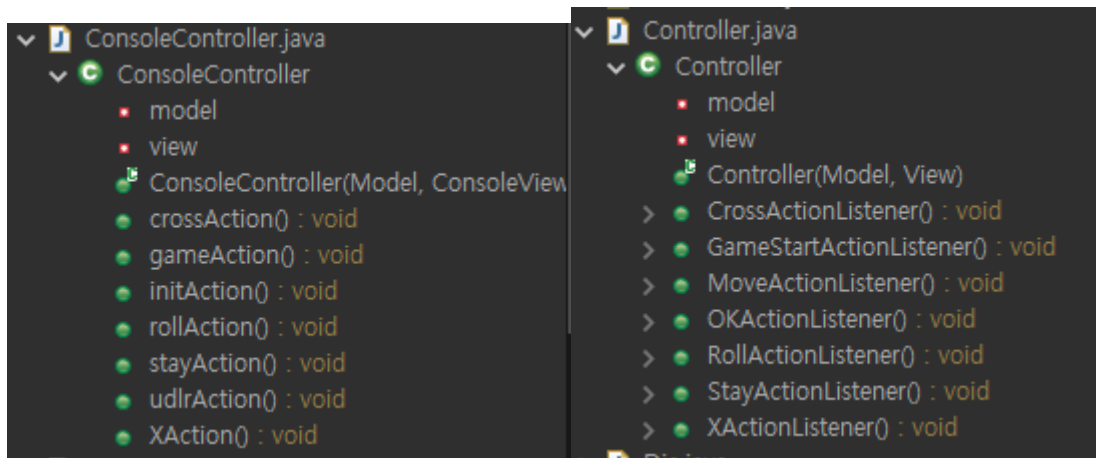
    jp_game.add(jbtn_roll);
    jp_game.add(jbtn_stay);
    add(jp_game);
}
```

```
public class Controller {
    private Model model;
    private View view;

    public Controller(Model model, View view) {
        this.model = model;
        this.view = view;

        GameStartActionListener();
        OKActionListener();
        RollActionListener();
        StayActionListener();
        CrossActionListener();
        XActionListener();
        MoveActionListener();
    }
}
```

그리고, View 클래스는 JFrame으로서 x y 좌표에 따라 화면에 그려 보여주기만 하는 역할을 담당하고, Controller 클래스는 이벤트 액션에 따른 반응을 읽고, 그 내용을 계산하며 model에 get/set을 하며 뇌의 역할을 수행합니다. 이는 model과 UI를 구분지었을 뿐만 아니라 view와 controller도 구분지었음을 보여줍니다.



또한 콘솔 버전의 UI와 그래픽 버전의 UI가 가지는 controller가 서로 비슷한 것을 확인할 수 있습니다. 이는 UI가 바뀌더라도 로직 이하의 계층을 변경없이 아주 쉽게 재사용했기 때문입니다. 실제로 GUI를 구현하고 나서, 콘솔을 구현하였는데, 콘솔을 구현할 때는 아주 빠르게 마무리할 수 있었습니다.

```
public abstract class Card {  
    int cardScore;  
  
    public Card() {  
    }  
  
    public int getCardScore() {  
        return cardScore;  
    }  
}
```

```
private ArrayList<Card> myScoreCards = new ArrayList<Card>();
```

Card 클래스는 여러 종류의 카드가 상속하고 있는 abstract class입니다. 이를 이용해 서로 다른 종류의 카드를 한 arraylist에 저장할 수 있었습니다. 이는 추후 더 많은 종류의 카드를 추가할 수 있으므로, 확장성에 유리한 객체 지향적인 설계를 했음을 증명합니다.

```
public class StartCell extends Square{
```

```
public class BridgeStartCell extends Square{
```

```
public class BridgeEndCell extends Square{
```

```
public class HammerCell extends Square {
```

...

```
public class Board {  
    private int totalSquareNum;  
    private Square[] squares;  
  
    public Board(List<String> squareInfos) {  
        totalSquareNum = squareInfos.size();  
  
        squares = new Square[totalSquareNum];  
        squares[0] = new StartCell(0, this, squareInfos.get(0));  
        for (int i = 1; i < totalSquareNum; i++) {  
            switch (squareInfos.get(i).charAt(0)) {  
                case 'B':  
                    squares[i] = new BridgeStartCell(i, this, squareInfos.get(i));  
                    break;  
  
                case 'b':  
                    squares[i] = new BridgeEndCell(i, this, squareInfos.get(i));  
                    break;  
  
                case 'P':  
                    squares[i] = new PhilipsDriverCell(i, this, squareInfos.get(i));  
                    break;  
  
                case 'H':  
                    squares[i] = new HammerCell(i, this, squareInfos.get(i));  
                    break;  
  
                case 'S':  
                    squares[i] = new SawCell(i, this, squareInfos.get(i));  
                    break;  
  
                default:  
                    squares[i] = new Square(i, this, squareInfos.get(i));  
            }  
        }  
    }  
}
```

마찬가지로, Square 클래스도 여러 Cell들이 상속 관계에 있는 클래스입니다. 이를 이용해 서로 다른 종류의 Cell들을 Board 클래스에서 하나의 array안에 저장할 수 있었습니다. Square 정보에 따라 초기에 한 번만 switch문을 이용해 객체를 생성해 저장해두게 되면, 한 줄의 메소드 호출로도 각 cell마다 다르게 override된 메소드를 이용할 수 있습니다. 이는 코드의 간편성과 함께 확장성에도 유리하다고 할 수 있습니다.

(2) 프로그램 사용 방법

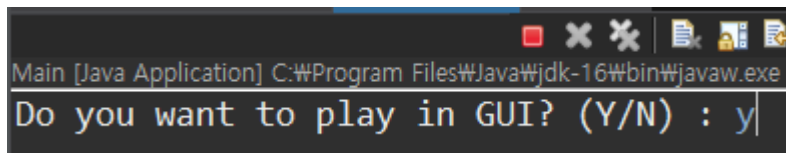
먼저 이클립스에서 프로젝트를 만듭니다.

그리고 제출한 디렉토리 'src_20184286' 안의 모든 소스파일들을 생성한 프로젝트의 src디렉토리에 붙여넣기 합니다.

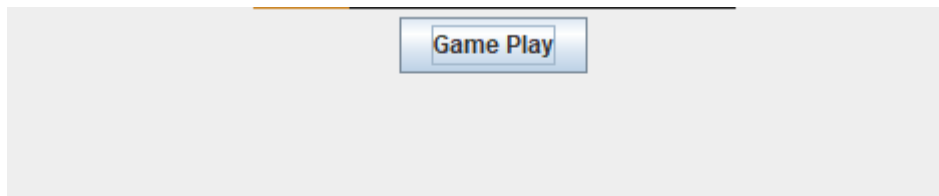
또한 함께 제출된 디렉토리 'data_20184286' 안의 데이터 파일들은 생성한 프로젝트의 디렉토리에 붙여넣습니다.

그리고 나서 이클립스에서 Main.java를 찾은 뒤, run 버튼을 눌러 게임을 시작합니다. 종료버튼은 따로 없으며 alt + F4를 이용하여 종료합니다.

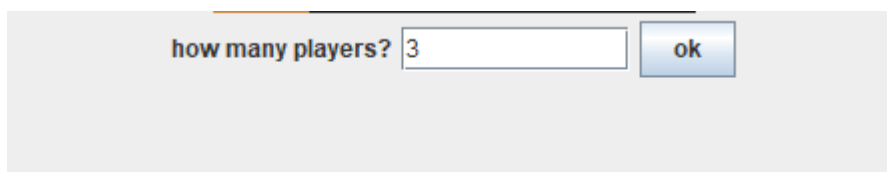
(3) 테스트 결과



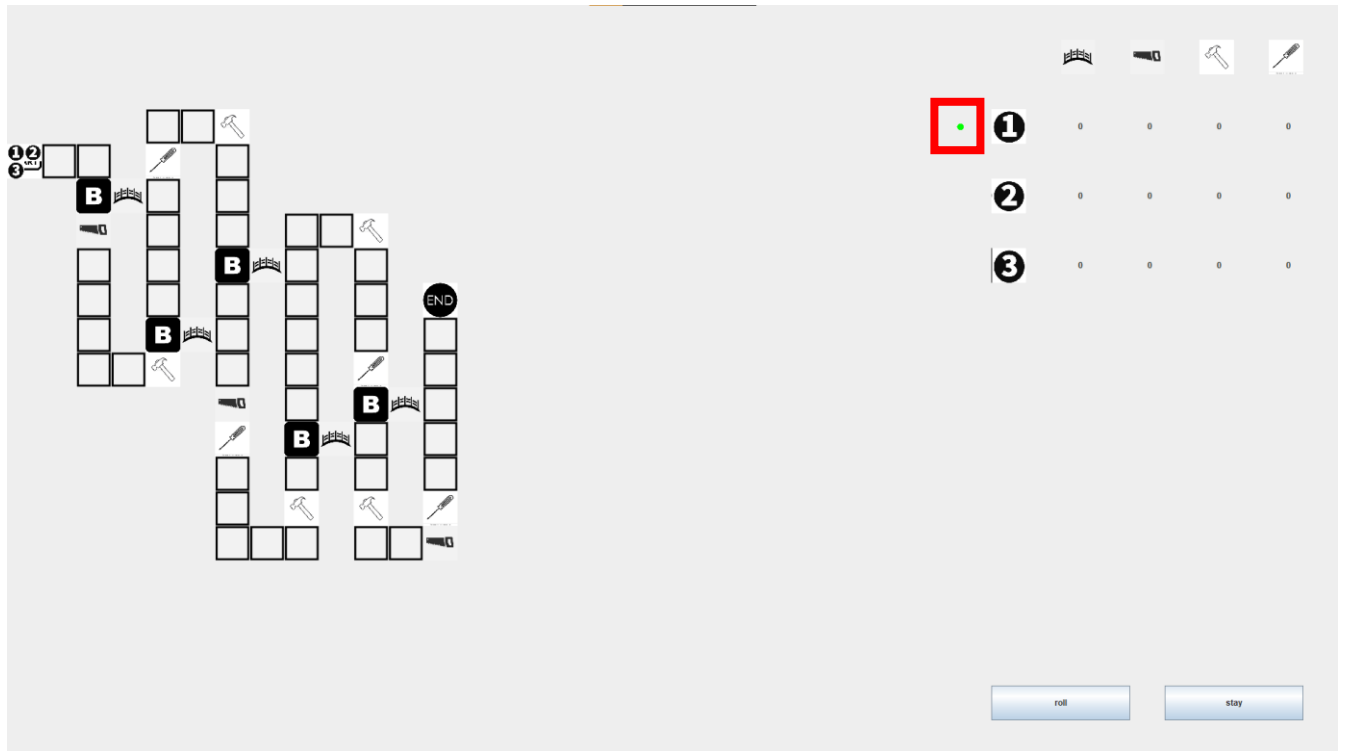
GUI로 플레이합니다.



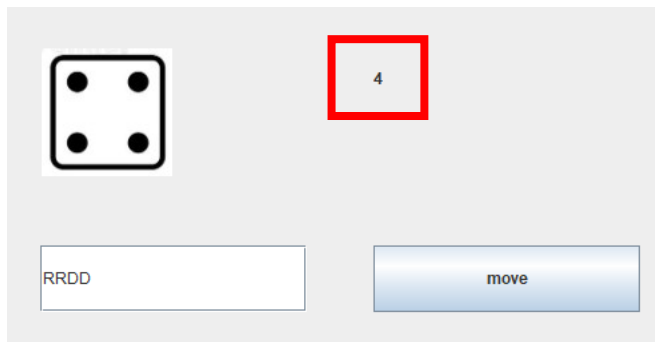
게임 시작 버튼을 누릅니다.



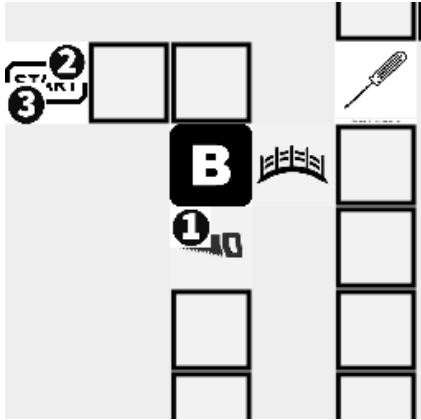
플레이할 플레이어의 수를 입력하고 OK버튼을 누릅니다.







본격적인 게임 화면입니다. 좌측에 맵과 1,2,3번의 piece들이 보여집니다. 우측에는 입력된 플레이어의 수의 플레이어들이 보여지는 상태창과 roll 버튼, stay 버튼이 존재합니다. Turn에 대한 정보는 초록색 동그라미로 상태창에서 확인할 수 있습니다.



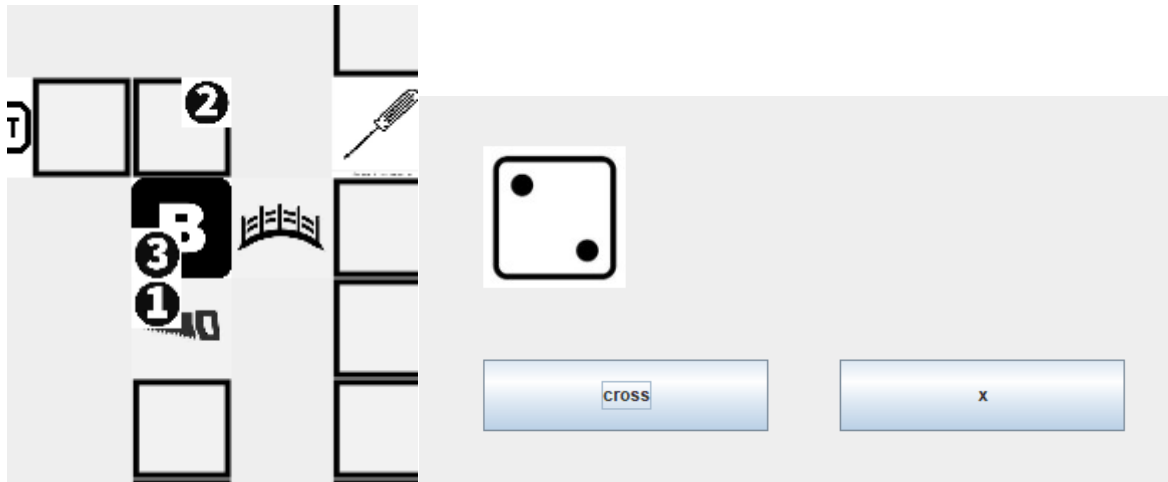
Roll 버튼을 누른 뒤, 주사위 모양과 함께 움직일 수 있는 칸 수를 표시해줍니다. 이를 참고하여 방향 커맨드를 입력하고 move 버튼을 누릅니다.



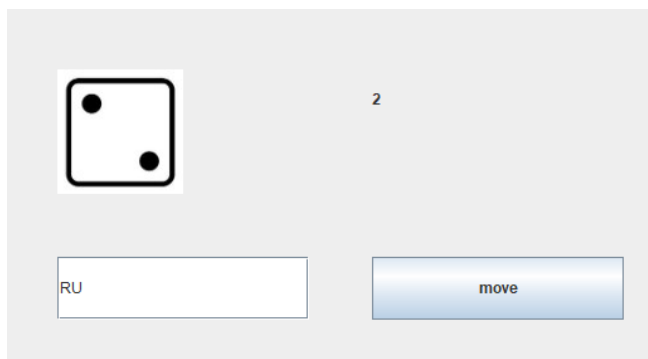
1번 piece가 입력된 방향 커맨드(RRDD)대로 이동했음을 알 수 있습니다.

				
1	0	1	0	0
2	0	0	0	0
3	0	0	0	0

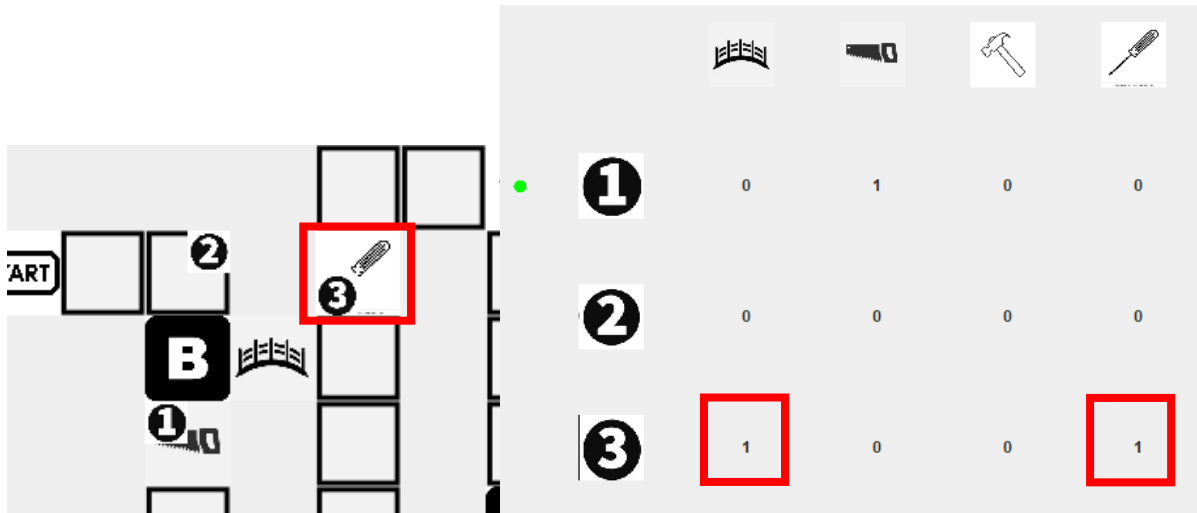
다음 turn으로 넘어간 모습과, 그 전에 1번 piece가 saw cell에 도달하면서 saw card를 한 장 획득하였음을 확인할 수 있습니다.



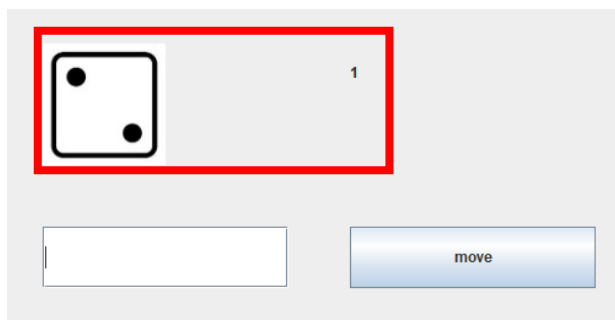
반복해서 진행이 되다가 3번 piece가 bridge를 건널 수 있는 cell에 도달했습니다. 따라서 cross 버튼과 x 버튼이 화면에 보여집니다.



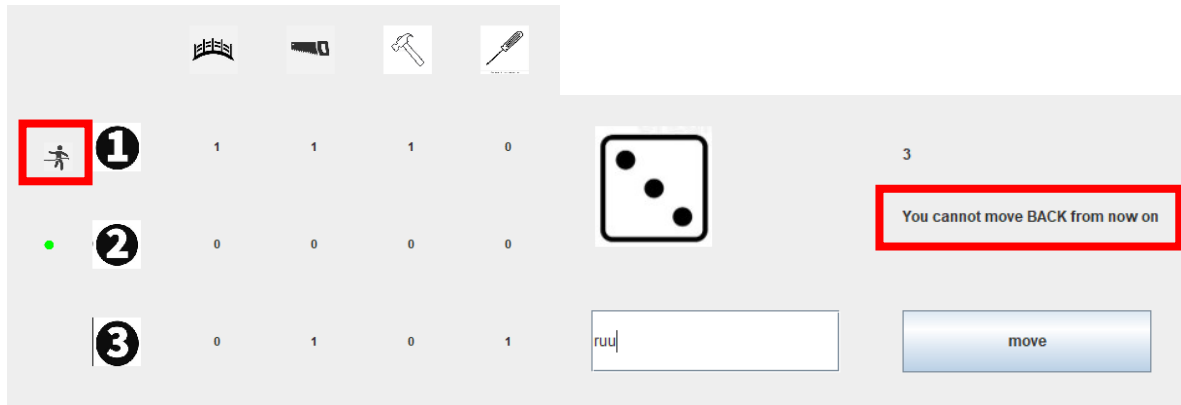
Cross 하기 위해 cross 버튼을 누른 후 그에 맞추어 방향 커맨드를 입력해줍니다.



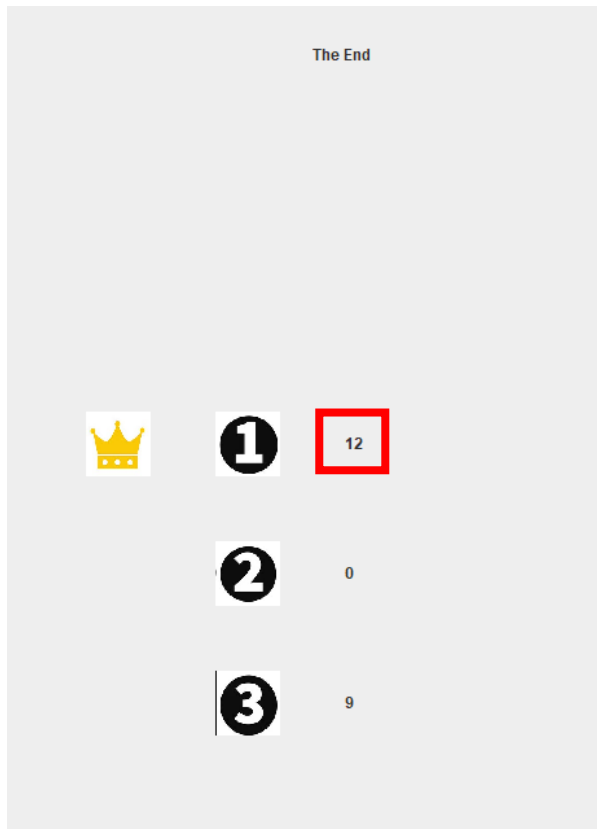
3번 piece가 bridge를 건너며 입력한 방향 커맨드(RU)대로 이동했습니다. 이 과정에서 bridge card를 하나 받게 되었고 driver cell을 밟으며 driver card를 획득하였음을 확인할 수 있습니다.



게임이 계속해서 진행되다가 캡쳐한 3번 플레이어 turn의 모습입니다. 주사위의 눈은 2이지만, 실제 움직일 수 있는 칸은 1입니다. 이는 bridge card를 한 장 가지고 있으므로 2 - 1의 1칸만을 이동할 수 있게 되었음을 보여줍니다. Bridge card는 stay를 하면 사라집니다.



마찬가지로 진행이 계속 되다가, 1번 piece가 완주를 하게 되어서, 상태창에는 완주 표시와 함께 더 이상 turn을 받지 않게 됩니다. 한 편, 다른 플레이어들은 이제 더 이상 뒤로 move할 수 없게 됨을 알립니다.



3번 piece가 완주를 해서, board 위에는 이제 하나의 piece인 2번 piece 밖에 남지 않게 되어서, 게임은 종료됩니다. 이 때, 가지고 있던 카드와 완주 순위 점수를 합쳐서 점수가 보여지고, 가장 높은 점수를 받은 1번 플레이어에게 왕관 표시로 우승자를 알려주며 게임을 종료된 모습을 확인할 수 있습니다.

```
Do you want to play in GUI? (Y/N) : n
How many players? : 4
turn 1
now your piece is on index 0
your BC:0, HC:0, SC:0, PDC:0
roll die or stay? (r/s) : r
die's faceValue : 1, your move count : 1
enter udlr : r
turn 2
now your piece is on index 0
your BC:0, HC:0, SC:0, PDC:0
roll die or stay? (r/s) : s
turn 3
now your piece is on index 0
your BC:0, HC:0, SC:0, PDC:0
roll die or stay? (r/s) : s
turn 4
now your piece is on index 0
your BC:0, HC:0, SC:0, PDC:0
roll die or stay? (r/s) : s
turn 1
now your piece is on index 1
your BC:0, HC:0, SC:0, PDC:0
roll die or stay? (r/s) : r
die's faceValue : 6, your move count : 6
enter udlr : rddddd
|turn 2
now your piece is on index 0
your BC:0, HC:0, SC:0, PDC:0
roll die or stay? (r/s) :
```

다음은, 콘솔로 구현한 bridge game입니다. 비록 맵은 구현하지 못했지만, 그 외의 동작은 정상적으로 GUI와 동일히 작동하는 모습을 보입니다. 맵을 따로 옆에 두고 참고하여 플레이한다면 똑같이 게임을 즐길 수 있습니다.

Piece의 위치는 index로 알 수 있고, 현재의 turn, 각 플레이어의 카드 현황, roll or stay, 움직일 수 있는 칸 수 그리고 방향 커맨드를 입력 받을 수 있음을 위의 캡처화면에서 확인할 수 있습니다.

이상입니다. 감사합니다.