
SOFTWARE ENGINEERING

Term Project
Spring, 2019

Yutnori Game

Junseok Lee, 20152345

Jesoon Kang, 20170937

Kyeongtae Kim, 20175119

Keonwoo Kim, 20172762

a) Team Members & Roles

Member	Project Role
Jesoon Kang	Planning, Implementation (Control Part), Testing Code
Junseok Lee	Documentation, Implementation (Control Part, User Interface), Testing Code
Kyeongtae Kim	Planning, Implementation (Control Part), Testing Code
Keonwoo Kim	Planning, Implementation (Model Part), Testing Code

b) Vision

1. Introduction

1.1 Document Purpose and Scope

This section outline the vision for the Yutnoli Program. The purposes of this document are to:

- Gather and describe customer requests for software features
- Propose a solution
- Identify any constraints to the proposed solution
- Identify stakeholders and users
- Identify the software development team

2. Customer requests

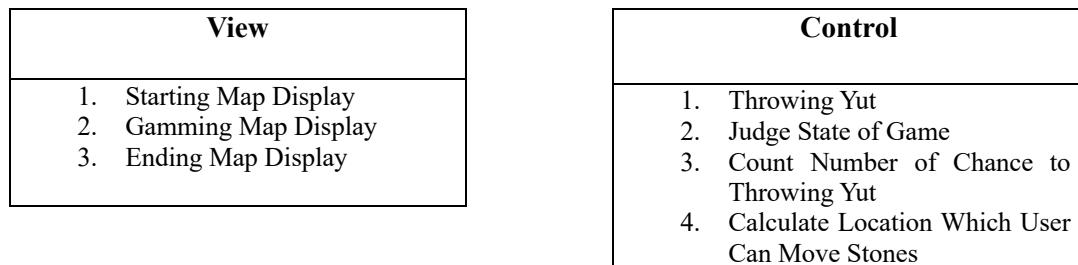
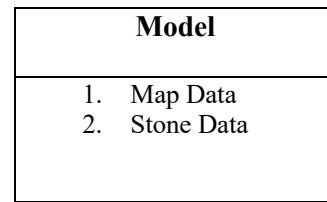
1. User wants to play Yutnoli
2. User can choose number of players and number of stones
3. User can see situation of the game visually
4. User can choose how to throw Yut
5. User can choose which stone to move

6. User can make group of stones.
7. User can catch the others stone
8. All stones are starts at starting position, and any player who moved all of his stones to ending position becomes winner. Winner player have to displayed.
9. After one game ended user can decide restart or end game.

3. Product features

3.1 MVC pattern.

- MVC pattern is well suits for designing game.



3.2 Test Oriented Programming.

- Make test codes for every classes.

4. Constraints

4.1 Programming language

- Java : One of the most famous Object Oriented Language
- Java Swing Library : Use for User Interface

5. Risk

5.1 Using Java Swing

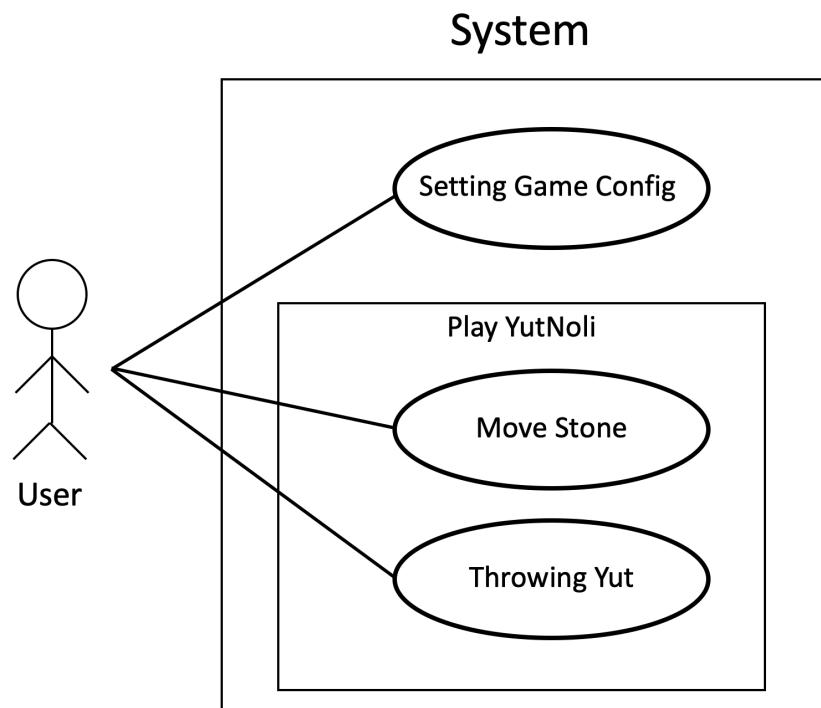
- Unfamiliar. Get used to it first before implements functions to reduce risk.

6. User Descriptions

Title	Role	Description of use
Game User	Playing game	Playing game - Using overall function for playing

c) Use Case Model

1) Use Case Diagrams



2) Use Case Descriptions

Scope : Desktop computer which can execute program made by java code.

Level : user goal

Primary Actor : Game User

Stakeholders and Interests :

-Game User : Wants to play game at fast and clean environment, without any bugs.

Preconditions : Game User have to know about rule of Yutnoli.

Success Guarantee(or Postconditions) : Game is over. The winner is decided

Main Success Scenario (or Basic Flow) :

1. Game Users start game and choose order.
2. Game User select option for how to throw Yut(random or selected) and throw yut.
3. Add chance to throw Yut, if Yut position is Moe or Yut.

Game User repeats steps 2-3 until Game User uses all of his chances to throw Yut.

4. Game User decide which stone should move and where to go.
5. Add chance to throw Yut, if Game User's stone caught the other user's stone.

Game User repeats steps 2-5 until Game User uses all of his chances.

6. Change turn to other Game User.
7. If one of the Game User move all of his stones to end point, then game end.
8. Game User choose between two options, end game or restart game.

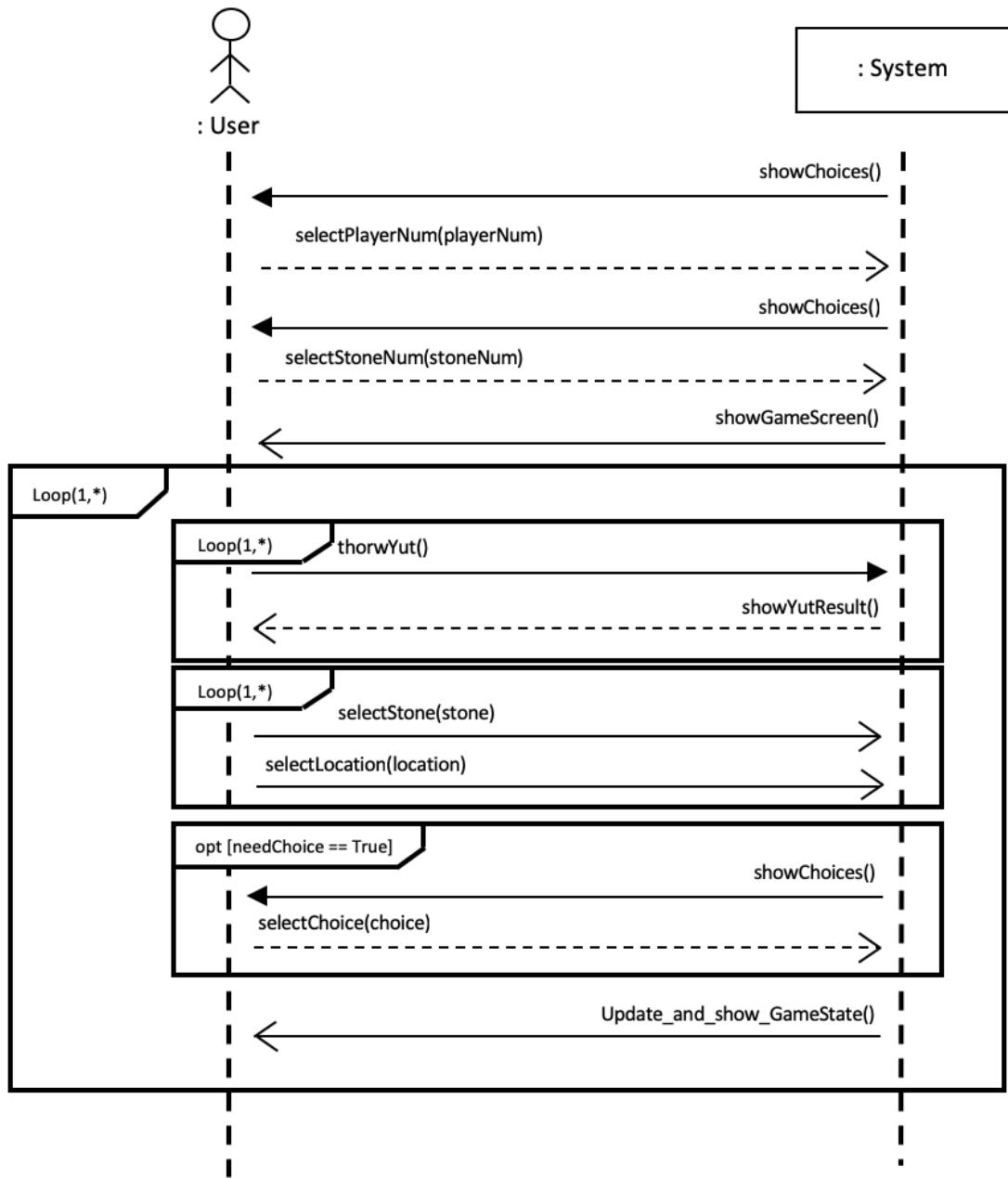
Game User repeats step 1-8 until Game User choose end game.

9. Program finish.

Extensions (or Alternative Flows) :

- a. If Game Users stone located at entrance of shortcut :
 1. Game User can choose only position that located at shortcut.
- b. If Game User caught enemy's stone at right in front of starting position and use Back-Doe move :
 1. That stone treated as which stone located at the end position.

3) System Sequence Diagrams (SSD)



4) Operation Contracts

Operation :	selectPlayerNum(playerNum : integer)
Cross References :	Setting Game Configuration
Preconditions :	YutNuli Instance is exist Program on game initializing Statement
Postconditions :	<ul style="list-style-type: none"> ○ 게임을 진행할 플레이어 수가 정해진다. ○ 플레이어 인스턴스가 생성된다. ○ 윷놀이 판 모델과 association 연결.

Operation :	selectStoneNum(stoneNum : integer)
Cross References :	Setting Game Configuration
Preconditions :	<ul style="list-style-type: none"> ○ 시스템이 게임 초기화 상태에 있다. ○ 플레이어 인스턴스가 생성 되어 있다.
Postconditions :	<ul style="list-style-type: none"> ○ 플레이어가 가질 돌의 갯수가 정해진다. ○ 돌 인스턴스가 생성된다. ○ 플레이어 인스턴스와 association 연결.

Operation :	throwYut()
Cross References :	Throwing Yut
Preconditions :	<ul style="list-style-type: none"> ○ 플레이어가 던질 차례가 되었다. ○ 윷을 던질 기회가 남아있다.
Postconditions :	<ul style="list-style-type: none"> ○ 윷을 던져 나온 값을 얻는다.

Operation :	selectStone(stone : Class)
Cross References :	Move Stone
Preconditions :	<ul style="list-style-type: none"> ○ 옮길 수 있는 말이 하나 이상 남아있다 ○ 돌을 옮길 윷 던진 결과가 남아 있다.
Postconditions :	<ul style="list-style-type: none"> ○ 옮길 돌 인스턴스가 선택되어 진다.

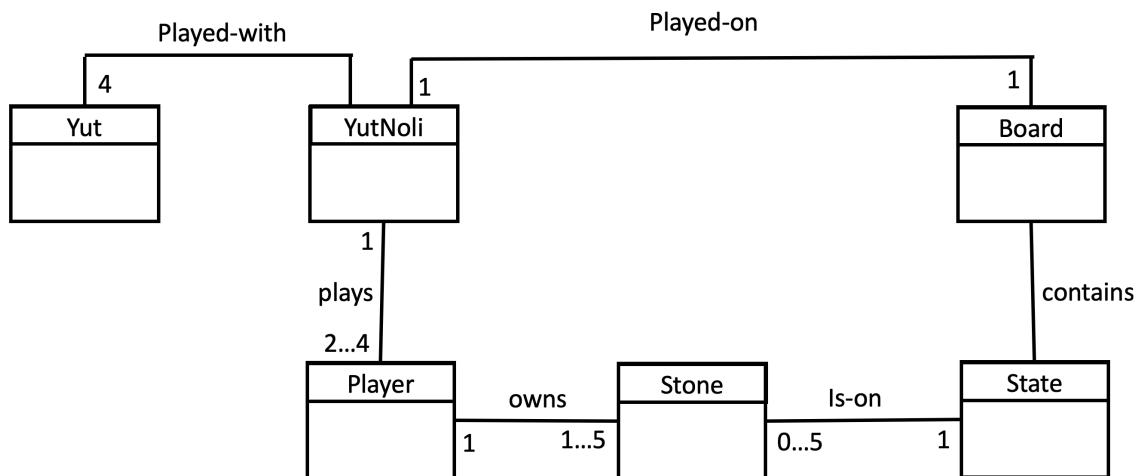
Operation :	selectLoacation(state : Class)
Cross References :	Move Stone
Preconditions :	<ul style="list-style-type: none"> ○ 플레이어가 옮길 돌이 선택되어져 있다 ○ 돌을 옮길 윷 던진 결과가 남아 있다.
Postconditions :	<ul style="list-style-type: none"> ○ 유효한 위치라면 돌이 옮겨진다 ○ 유효하지 않은 위치라면 selectLocation 다시 실행 ○ 돌이 옮겨졌다면, 해당 윷 판의 state와 association 연결. 다음 플레이어의 throwYut 차례가 된다. ○ 해당 위치에 옮기기 위한 선택지가 필요하다면 (마지막 골인 지점), 'selectChoice' 호출

Operation :	selectChoice (choice : integer))
Cross References :	MoveStone
Preconditions :	<ul style="list-style-type: none"> ○ 돌이 옮겨질 위치가 정해 졌고, 소모할 윷 결과를 선택해야한다. (마지막 골인 지점)

Postconditions :	<ul style="list-style-type: none"> ○ 돌이 옮겨진다. ○ 돌 인스턴스가 사라진다. ○ 다음 플레이어의 throwYut 호출
-------------------------	---

d) Analysis & Design, Implementation, and Test Document

1) Domain Model



2) How MVC concept was applied

- 게임 개발에 있어서 MVC모델을 적용하게 되면 매우 효율적으로 프로그램 개발을 진행할 수 있다. 그 이유는 게임 자체가 MVC 패턴과 매우 유사한 구조를 갖고 있기 때문이다. 따라서 이러한 MVC 패턴의 장점을 효율적으로 활용하기 위해서 프로그램을 Model, View, Control 3가지 Package로 나누어 게임의 기능을 분류하여 개발하였다.
- 우선, 게임이 작동하기 위해서는 전반적인 게임의 상황을 담고 있는 부분이 필요하다. MVC 패턴에서는 MODEL이 이와 같은 기능을 한다고 판단하여, 게임 데이터를 저장하는 부분을 모두 MODEL Package 안에 구현하였다. 이 Model Package 안의 클래스에서 생성되는 객체들은 View 와 Control Package에 있는 클래스의 객체들을 일절 갖고 있지 않다. 즉, View 와 Control로부터 독립적이며 그들이 요청하는 데이터에 대한 명령만 수행할 뿐, View 와 Control에 어떠한 영향도 끼치지 않는다.
- 다음으로, View Package는 유저에게 user interface를 제공하며, 유저로부터 받은 입력을 Controller에게 전달하는 역할을 한다. View Package의 Class들은 게임의 상황에 대해 어떠한 판단도 하지 않으며, Controller에서 오는 신호와, Model에 있는 정보만을 가지고

private 및 default로 두고 get, set function을 정의함으로써 의존성을 최대한 줄였고, MVC 패턴을 이용하여 view는 Model의 객체를 최소한 적게 생성하여 모델의 데이터를 query하기만 하고, Controller 역시 view 및 model의 객체를 최소한 적게 생성하여 dependency를 줄이기 위해 노력했습니다. 또한 테스트 코드를 짜다 각 함수들의 크기가 커지면서 하는 일이 많아지게 되면, unit test를 하기 힘들어지게 된다는 것을 느껴 최대한 한 메소드 및 한 클래스가 single responsibility를 가지도록 design 했습니다.

5) How Object-Oriented Analysis & Design Applied

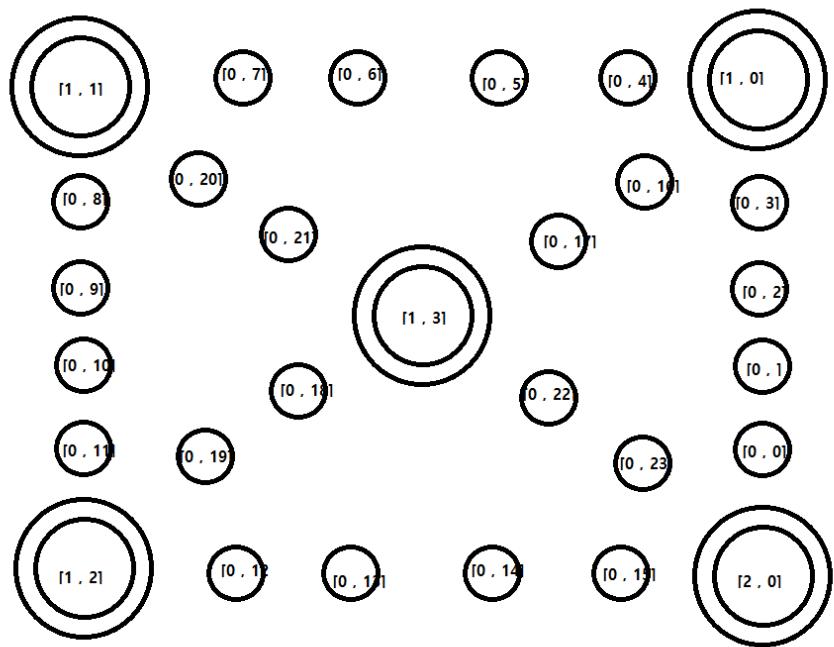
프로그램을 개발하면서 객체지향 언어의 특성을 최대한으로 활용하기 위해 노력했다.

먼저 객체지향 프로그래밍에서 가장 중요하다고 할 수 있는 Polymorphism을 활용하였다. View에서 Control에게 전달되는 신호는 크게 2가지, 작게는 3가지로 분류 될 수 있다. 2가지 분류는 말을 움직이는 State에 대한 신호와, 윗을 던지는 Throw에 대한 신호이다. 이때 Throw 신호는 랜덤으로 던지는 Random Throw 와 이용자가 윗의 형태를 지정하는 Selected Throw로 나눌 수 있다. 이때 두가지의 던지는 동작을 처리하는 과정은 매우 유사하기 때문에 Control 부분에서 신호를 받는 함수를 overload하였다.

```
public void recieveThrowAction() {
    this.throwOption = 0;
    mainFlow(0);
}

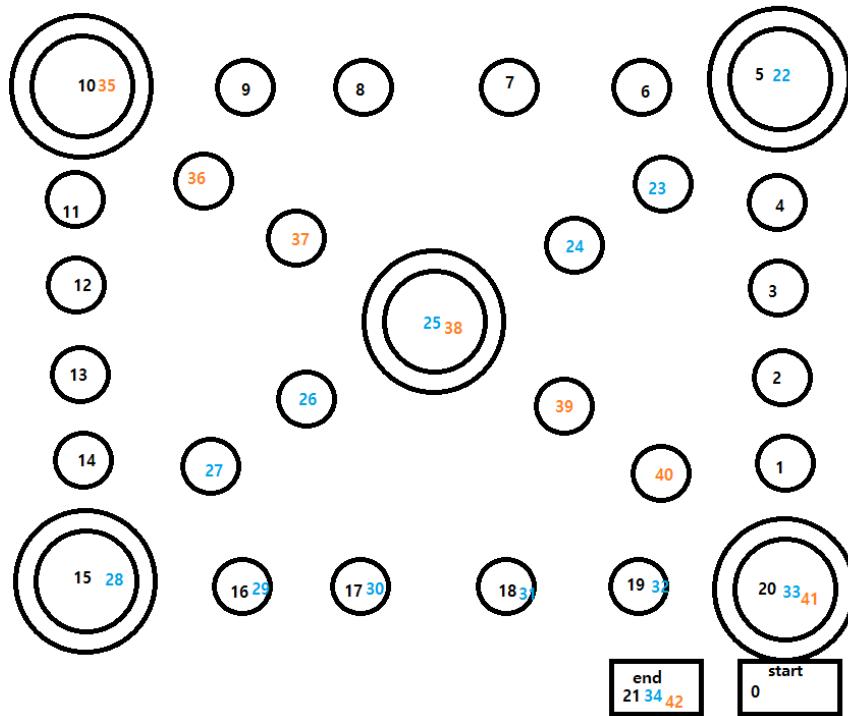
public void recieveThrowAction(int selected) {
    this.throwOption = 1;
    this.selected = selected;
    mainFlow(0);
}
```

두번째는 상속을 활용하였다. 윗놀이 프로그램을 개발하면서 고민을 했던 부분 중의 하나는 윗판의 모양을 표현하는 방법이었다. 윗판의 State들은 유사해 보이지만 그 위치에 따라 역할이 살짝 다르다. 일반 State들과 다르게 모서리와 가운데 있는 State는 말이 움직일 수 있는 위치와 방향을 제한한다. 따라서 각각의 위치에 따라 State를 구분하기 위해서 2가지 인수로 State를 표현하였다. 하나의 인수는 State의 종류를 표현(일반 State, 모서리-중앙 State, 시작 State 등)하고, 다른 인수는 해당 종류의 State 중 몇 번 째 인지 나타낸다.



하지만 이는 프로그래머가 보기에 매우 직관적이라 이해하기 편하지만 State간 계산을 할 때에는 예외 처리가 많아서 알고리즘이 복잡해 진다.

따라서 State간 거리 등의 계산을 할 때 편리하게 하기 위해서 연속된 숫자로 이루어진 표현법을 구상하였다.



하나의 State가 여러 개의 번호를 부여 받은 경우는 말이 코너에서 새로운 경로로 분기하는 경우를 쉽게 계산하기 위함이다.

이 두가지 경우로 프로그램 개발을 진행하면 두가지 표현법에 자유로운 전환이 필요했고, 이러한 전환을 해주는 함수를 담은 클래스를 만들어, 관련된 연산을 수행하는 클래스가 상속받아 사용할 수 있도록 하였다.

```
public class StateExpressionChange {  
    public int convertOr2Seq(int state, int num) {  
        public int[] convertSeq2Or(int num)  
    }  
    public class DeleteMove extends StateExpressionChange{
```

다음으로 객체지향 프로그래밍에서 Coupling 을 최대한 줄이기 위해서, Class 간 각자의 역할을 확실히 구분하여 프로그래밍을 진행하였다.

Model Package안의 Class의 경우 게임의 중요한 데이터를 담고 있기 때문에 다른 Class에서 해당 데이터를 직접 접근하여 변경하는 경유 게임 진행의 심각한 오류를 발생시킬 수 있다. 따라서 Model Package 안의 Class 내 정보를 담당하는 모든 변수를 Private으로 선언하고, 해당 변수에 대한 get, set function 을 구현하여, [데이터의 무결성을 확보](#)하였다.

게임이 진행되면서 컨트롤을 담당하는 객체가 게임 데이터를 변경하게 되면 이 변경된 정보는 다른 객체들이 바라보는 정보에도 모두 적용되어야 한다. 이를 적용하기 위해서 하나의 변경이 발생하면 관련된 모든 객체에게 신호를 보내어 정보를 변경하게 하면 불필요한 연산이 많아져 비효율적일 수 있다. 다른 객체들이 정보를 하나의 객체를 바라보고 있다면, 변경된 정보를 전달하는 불필요한 과정을 생략할 수 있다. 이에 [Singleton Pattern](#) 을 적용하였다. 모델 객체를 생성할 때 만약 만들어진 객체가 없다면 새로 생성해서 반환하고, 이미 있다면 그 객체를 반환한다 이를 통해서 다른 객체들이 하나의 모델을 바라보게 할 수 있다.

```
public static GameData getInstance() {  
    if(gameData == null) {  
        gameData = new GameData();  
    }  
    return gameData;  
}
```

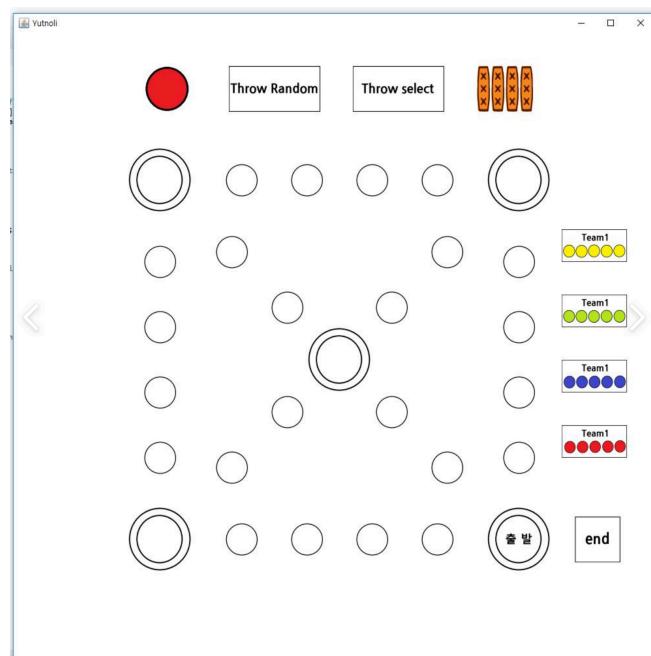
6) Usage of Program & Screen shots



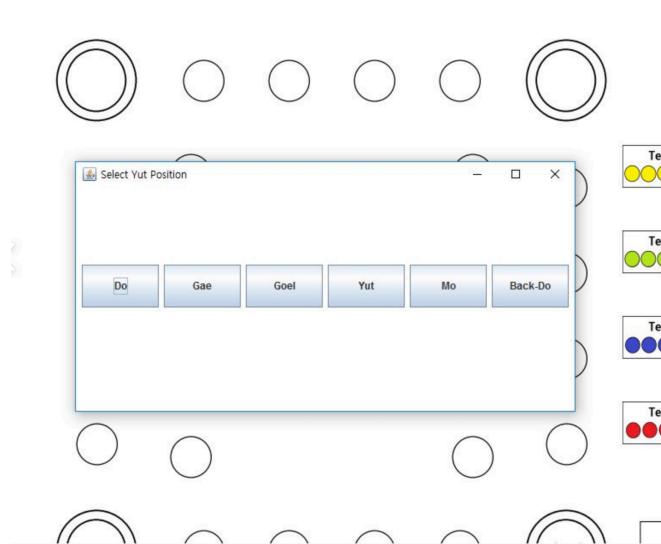
< 1. 플레이어 수 선택 >



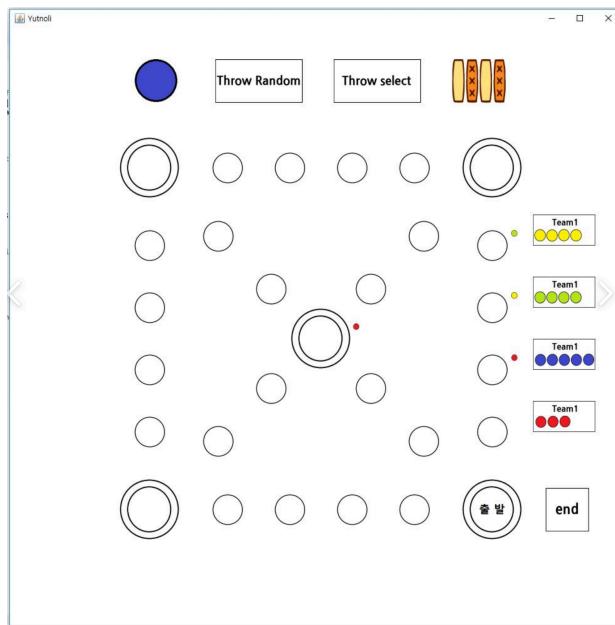
<2. 말 수 선택 >



<3. 랜덤 던지기 / 선택 던지기 선택 >

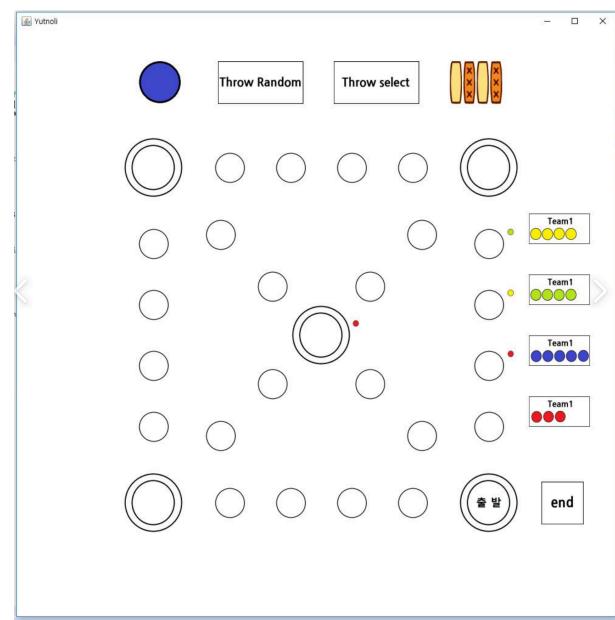


<4. 선택 던지기 시, 윷 결과 선택 >



<5. 플레이어나, 자신의 말의 위치를 클릭하여 옮길 말 선택 >

- 플레이어 선택 시, 새로운 말이 선택됨



<6. 말을 옮길 위치 선택 >



< 7. 골인 지점에서, 윷 던진 결과가 여러 개일 시, 골인에 사용 될 움직임 선택 >



< 8. 게임 오버, 재시작 / 프로그램 종료 선택 >



< 9. 재시작 선택 시, 게임 설정 다시 세팅 >

6) Functional Unit test cases & results

- use junit4 in a eclipse.
- we reference a document for enviroment settting to use junit4.
: <https://lee-mandu.tistory.com/398>

6-1) Unit test cases

테스트는 개발 중간 중간에 몇몇 필요한 핵심 부분에서만 테스트 케이스를 미리 만들어 검증하는 방식을 택했습니다. 아래는 저희가 이용한 테스트 케이스들 및 그 실행 결과입니다.

< GameData 클래스의 initGameData 함수가 올바르게 작동하는지 검증하는 테스트 케이스. >

```
public class TestGameData {  
    private GameData gameData; // test fixture variable.  
  
    /* set up method using @Before syntax. */  
    /* @Before methods are run before each test. */  
    /* initializes test fixture variables(gameData). */  
    @Before  
    public void runBeforeEachTest(){  
        System.out.println("Before testing.");  
        gameData = GameData.getInstance();  
    }  
  
    /* check if initGameData works correctly. */  
    @Test  
    public void testInitGameData(){  
        for(int i = 1; i <= 3; i++) {  
            for(int j = 1; j<=5; j++) {  
                gameData.initGameData(i, j);  
                assertEquals(i, gameData.getNumOfPlayers());  
                assertEquals(j, gameData.getNumOfStones());  
            }  
        }  
    }  
}
```

```
/* tear-down method using @After. */
/* @After methods are run after each test. */
/* frees test fixture variables(gameData). */
@After
public void runAfterEachTest() {
    System.out.println("After testing.");
    gameData = null;

    /* execute garbage collection. */
    System.gc();
}

}
```

PlayerData 클래스는 우리 응용에서 immutable 클래스이고 다른 클래스에서 그 멤버 변수를 참조하므로 그 필수 멤버들이 이후의 협업 개발 과정에서 없어지지는 않았는지, 초기화는 규칙을 정한 대로 잘 되었는지를 검증하는 것이 필요함. 따라서 이를 검증하는 테스트 케이스를 만들었습니다. 또한, findMove()함수가 올바르게 작동하는지 검증하는 테스트 케이스를 만들었습니다.

```

public class TestPlayerData {
    private PlayerData playerData; // test fixture variable.

    /* set up method using @Before syntax. */
    /* @Before methods are run before each test. */
    /* initializes test fixture variables(playerData). */
    @Before
    public void runBeforeFindMoveTest(){
        System.out.println("Before testing.");
        playerData = new PlayerData();
    }

    /* check if essential member variables of PlayerData Class exists
     * and if shortly after creating the object, the member variables are initialized correctly. */
    @Test
    public void testInitialMemberVars(){
        assertEquals(9, playerData.getTeam());
        assertEquals(0, playerData.getNumber());
        assertEquals(1, playerData.getThrowCount());
        assertEquals(0, playerData.getFinish());
    }

    /* check if findMove works correctly. */
    @Test
    public void testFindMove(){
        ArrayList<Integer> moves = playerData.getMoves();

        moves.add(1);
        moves.add(3);
        moves.add(4);

        assertEquals(true, playerData.findMove(1));
        assertEquals(false, playerData.findMove(2));
        assertEquals(true, playerData.findMove(3));
        assertEquals(true, playerData.findMove(4));
        assertEquals(false, playerData.findMove(5));
    }

    /* tear-down method using @After. */
    /* @After methods are run after each test. */
    /* frees test fixture variables(playerData). */
    @After
    public void runAfterFindMoveTest() {
        System.out.println("After testing.");

        playerData = null;

        /* execute garbage collection. */
        System.gc();
    }
}

```

앞에서 언급한 PlayerData 클래스와 마찬가지로 StateData 클래스 역시 우리 응용에서 immutable 클래스이고 다른 클래스에서 그 래퍼 변수를 참조하므로 그 필수 래퍼들이 이후의 협업 개발 과정에서 없어지지는 않았는지, 초기화는 규칙을 정한 대로 잘 되었는지를 검증하는 것이 필요했습니다. 따라서 이를 검증하는 테스트 케이스를 만들었습니다.

6-2) Unit test results.

각각의 테스트 케이스 실행 결과는 아래와 같습니다.

Finished after 0.024 seconds

Runs: 1/1 Errors: 0 Failures: 0

▶ test.unit.TestGameData [Runner: JUnit 4] (0.000 s)

```
1 package test.unit;
2
3④ import Model.GameData;
4
5
6 public class TestGameData {
7     private GameData gameData; // test fixture variable.
8
9     /* setUp method using @Before syntax. */
10    /* @Before methods are run before each test. */
11    /* initializes test fixture variables(gameData). */
12    @Before
13    public void runBeforeEachTest(){
14        System.out.println("Before testing.");
15        gameData = GameData.getInstance();
16    }
17
18
19
20
21
22
23
24
25
26
27
```

Finished after 0.02 seconds

Runs: 1/1 Errors: 0 Failures: 0

▶ test.unit.TestStateData [Runner: JUnit 4] (0.000 s)

```
1 package test.unit;
2
3④ import Model.StateData;
4
5
6 public class TestStateData {
7     private StateData stateData; // test fixture variable.
8
9     /* setUp method using @Before syntax. */
10    /* @Before methods are run before each test. */
11    /* initializes test fixture variables(stateData). */
12    @Before
13    public void runBeforeEachTest(){
14        System.out.println("Before testing.");
15        stateData = new StateData();
16    }
17
18
19
20
21
22
23
24
25
26
27
```

테스트 케이스에 맞게 tested class 및 tested method가 올바르게 되어 있지 않다면 되어있지 않다면 아래와 같이 오류를 냅니다.

아래는 PlayerData 클래스의 램버 변수 default 값을 바꾸었을 때 junit이 이를 잡아내는 화면입니다.

The screenshot shows an IDE interface with two panes. The left pane displays the 'JUnit' view with the message 'Finished after 0.029 seconds' and statistics: 'Runs: 2/2', 'Errors: 0', and 'Failures: 0'. The right pane shows the source code for 'TestPlayerData.java'.

```

1 package test.unit;
2
3 import Model.PlayerData;
4
5 public class TestPlayerData {
6     private PlayerData playerData; // test fixture variable.
7
8     /* setup method using @Before syntax. */
9     /* @Before methods are run before each test. */
10    /* initializes test fixture variables(playerData). */
11    @Before
12    public void runBeforeFindMoveTest(){
13        System.out.println("Before testing.");
14        playerData = new PlayerData();
15    }
16
17    /* check if essential member variables of PlayerData Class exists
18     * and if shortly after creating the object, the member variables are ini
19     * @Test
20     public void testInitialMemberVars(){
21         assertEquals(9, playerData.getTeam());
22         assertEquals(0, playerData.getNumber());
23         assertEquals(1, playerData.getThrowCount());
24         assertEquals(0, playerData.getFinish());
25     }
26
27
28
29
30     */
31
32
33
34
35

```

Original.

```

public PlayerData() {

    this.team = 9;

    this.number = 0;

    this.throwCount = 1;

    this.finish = 0;

}

```

Code changed.

```

public PlayerData() {

    this.team = 1;

    this.number = 0;

    this.throwCount = 1;

    this.finish = 0;

}

```

바뀐 코드에서 default 값을 우리 응용에서 정한 값과 다르게 바꾸었기 때문에 테스트에 실패했습니다.

The screenshot shows the Eclipse IDE interface. On the left, the Project Explorer and JUnit view are visible. The JUnit view shows a successful run with 2/2 tests passed, 0 errors, and 1 failure. The failure is in the testFindMove test, which took 0.000 seconds. On the right, the code editor displays the PlayerData.java file. The code defines a class with member variables team, number, throwCount, and finish, and a moves ArrayList. It includes a comment mapping move values (0-4) to move names (do, gae, geo1, yut, mo). The constructor initializes team to 1, number to 0, throwCount to 1, and finish to 0. The findMove method iterates over the moves list to find a match. A failure trace is shown at the bottom, indicating an AssertionError was thrown.

```

public class PlayerData{
    int team;
    int number;
    int throwCount;
    int finish;
    ArrayList<Integer> moves = new ArrayList<Integer>();
    /*
     * 0 = do
     * 1 = gae
     * 2 = geo1
     * 3 = yut
     * 4 = mo
     */
    public PlayerData() {
        this.team = 1;
        this.number = 0;
        this.throwCount = 1;
        this.finish = 0;
    }
    public boolean findMove(int move) {
        Iterator<Integer> it = moves.iterator();
        while(it.hasNext()) {
            int val = (int)it.next();
            if (val == move) {
                return true;
            }
        }
        return false;
    }
}

```

e) Project Management Report

1) Github project repository Link

<https://github.com/yutnoli/yutnoli>

2) Project Progress History

프로그램 개발 과정 중에 초반 iteration에서 risk가 가장 높은 것부터 해결하여 개발을 진행하면서 risk가 줄어들 수 있도록 고려했다. 윷놀이 게임의 알고리즘은 수학의 연산이고, 알고 있는 부분을 활용하는 것이지만 UI의 경우 사용하기로 결정한 JAVA SWING이 생소하고, 해당 문법을 모르면 개발 진행이 불가하기 때문에 UI가 가장 큰 risk라고 판단하였다.

또한 UI같은 경우 Model과 Control의 테스트 코드로 작동할 수 있기 때문에 빠른 View의 구현이 결과적으로 프로그램 개발의 risk를 크게 줄일 수 있다고 판단하였다. 이에 EX와 같이 전반적인 부분의 반복 개발을 통해 새로운 생산물을 산출해 내면서도 UP와 같이 초반에 risk가 큰 View 부분 개발에 집중하였다. 그 예로 Control과 Model의 구현이 미미 하더라도 View에 해당하는 부분만 따로 디렉토리를 만들어 독립적으로 개발을 진행 후 합치는 방법을 택하였다. 이에 프로그램 개발 과정이 훨씬 수월해지는 효과를 얻을 수 있었다.

3) Experience

기존의 해오던 프로그램 개발 같은 경우는 개발을 진행하던 도중 커다란 문제가 종종 발생하여 모든 개발 프로세스가 멈추거나, 새로운 방법을 도입하기 위해 개발이 룰백 되는 경우가 있었다. 하지만 미리 팀원들과 회의를 진행하면서 리스크를 예상하고 수월한 쪽을 판단하며 개발을 하니, 개발을 저해하는 요소가 줄어들거나, 해당 요소가 발생되어도 수정 범위 안에서 발생되어 금방 주된 개발 과정을 진행할 수 있었다. 물론 문서작업을 하면서 개발 이외의 부분에 시간을 소요한다는 느낌도 들었지만 해당 문서가 개발 과정을 이해하는데 도움을 주고, 그에 따른 생산성이 증가할 수 있겠다는 생각이 들었다.