# 소프트웨어공학

# 프로젝트 유닛 테스트 보고서

## - Title : Yoot Project Junit Test -

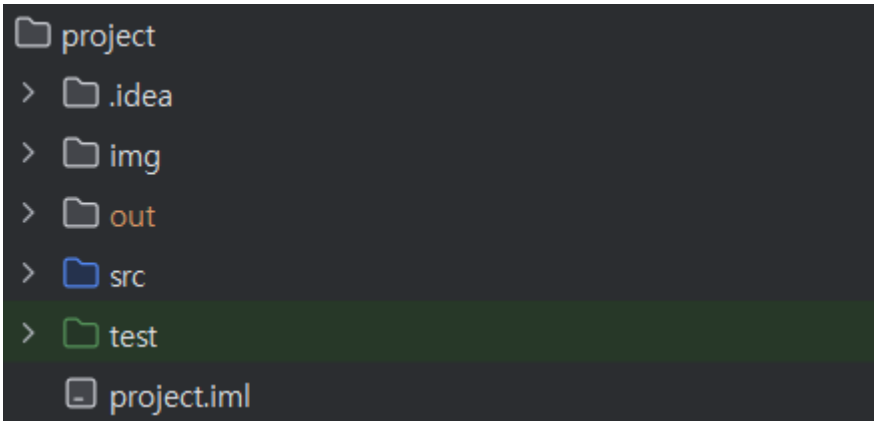
| | |
|---|---|
| **Class** | **소프트웨어공학, Class 01** |
| **Professor** | **이찬근** |
| **Team number** | **Team 18** |
| **Team Members** | **김찬중 (20213780)**<br>**박민용 (20213113)**<br>**엄태형 (20202029)**<br>**이정우 (20202021)**<br>**태아카 (20234483)** |
| **Author** | **김찬중** |

# 목차

# 1. Introduction

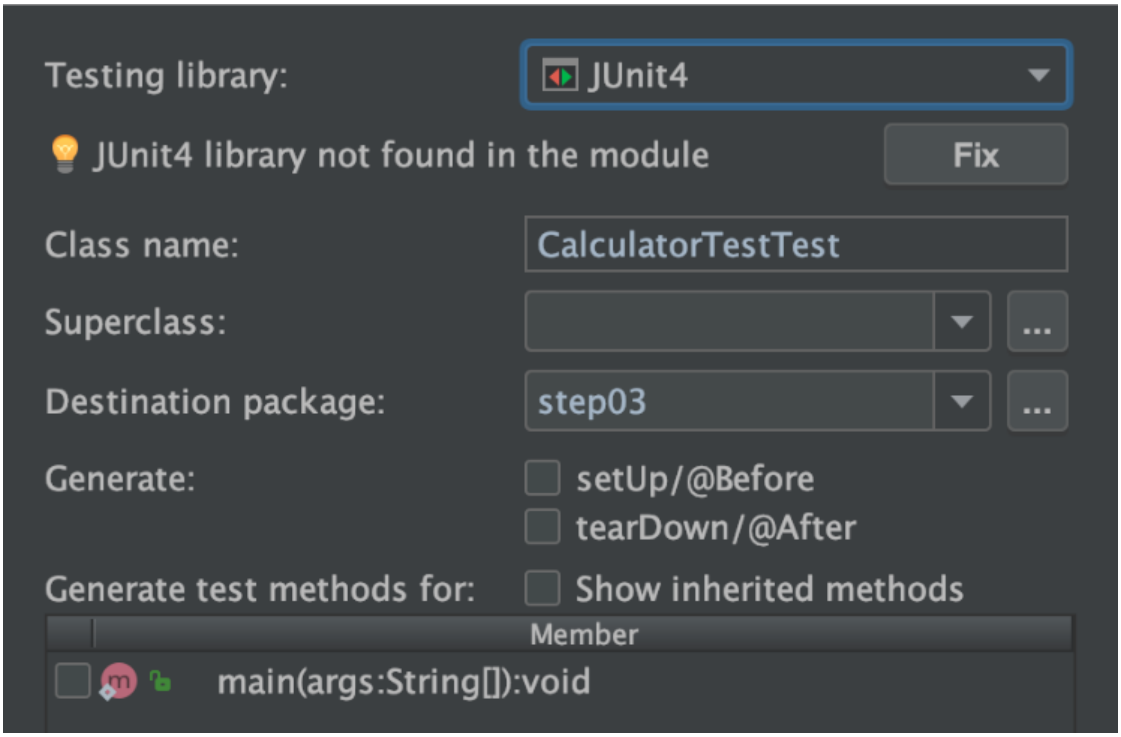
## A) Test 환경 구축

- 1) 테스트 폴더를 project 폴더 하위에 새로 추가해준다.




- 2) 생성된 폴더에서 모듈 세팅을 'Tests'로 지정하면 위와 같이 초록색 폴더로 바뀐다.

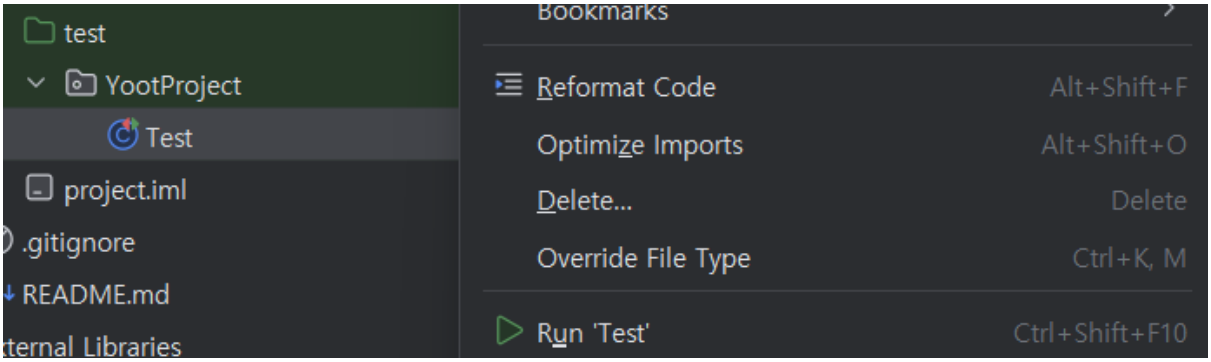
- 3) 테스트할 class 파일을 생성한다. 생성 시 Junit library의 dependency가 설정이 되어 있지 않으면 Fix 버튼을 눌러 종속성을 만족시켜준다.




- (위의 그림은 프로젝트와 무관하지만 동일하게 진행하면 됨)

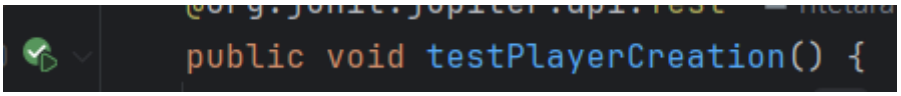
■ 4) 생성된 test class 파일은 테스트를 진행할 package와 동일한 명칭의 package 하위 폴더에 위치해야 한다.


■ 5) Test 코드를 작성 후 우클릭을 하면 아래의 사진과 같은 옵션이 뜨는데 Run 'Test'를 누르면 Junit test가 실행된다.




■ 6) Test가 진행되면 각 테스트가 성공 혹은 실패했는지를 색으로 표현해준다. 초록색은 성공을 빨간색은 실패를 나타낸다. Test class 자체가 초록색으로 빛나는 경우 모든 test case를 통과했다는 것을 알려준다. 테스트 진행 결과 및 src의 package 이름 그리고 테스트 진행 시간도 알려준다.

(모든 테스트가 통과한 모습)




■ 테스트 진행 후 테스트 코드 파일에 들어가면 좌측 중단점을 찍는 부분에 초록색으로 체크 표시가 보일 것이다. 빨간색 'X' 표시는 assert의 결과가 False임을, 초록색 'V' 표시는 True가 return 되었음을 알려준다. Test class 하위의 모든 method의 선언부분의 좌측에는 위와 같이 성공 여부가 모두 표시되어 있다.

■ 모든 Junit test를 진행하기 위해선 test method 위에 '@Test'를 표시해주어야 한다. Machine이 notation을 보고 해당 method가 unit test인지 아닌지를 판단하게 된다.

## 2. Body

### B) 테스트 코드

```
@BeforeEach  htetarakr
public void setUp() {
    // 테스트 전에 항상 mockRouteManager 초기화
    mockRouteManager = new IBoardRouteManager() {  htetarakr
        @Override  3 usages  htetarakr
        public int getStartRoute() { return 0; }

        @Override  3 usages  htetarakr
        public int getStartPosition() { return 0; }

        @Override  2 usages  htetarakr
        public int getRouteLength(int routeIndex) { return 10; }

        // 경로 변경 없음
        @Override  1 usage  htetarakr
        public BoardRouteManager.RouteChange getRouteChange(int routeIndex, int positionIndex) { return null; }

        // 백도 처리: 음수 위치는 0으로 조정
        @Override  1 usage  htetarakr
        public int adjustForBackdo(int routeIndex, int positionIndex) { return Math.max(0, positionIndex); }
    };
}
```

- 위의 코드는 각 테스트 method가 실행되기 전에 초기화를 진행해주는 setup 함수이다. Junit 4 version까지는 @before decorator을 통해 setup 함수임을 알려주었지만 Junit 5 version부턴 @BeforeEach라는 표현을 통해 각 method가 call되기 전에 setup이 진행된다는 것을 더 명확하게 알 수 있게 해준다. Setup method에선 각 판별로 적용되는 규칙을 불러오기 위해 Interface를 선언해준다.
- getStartRoute의 경우 시작 시 밟고 있는 곳이 외곽 route인데 프로젝트 내에선 외곽 루트를 route 0로 표현한다.
- getStartPostion의 경우 시작 시 밟고 있는 위치의 point가 0이기 때문에 위와 같이 초기화해준다. 배열로 따지면 윷놀이 판의 좌표는 point[route][position]과 같은 형태로 표현할 수 있다. Row에 해당하는 것은 외곽 루트인지 아니면 중앙 루트인지를 나타낸다.
- getRouteLength의 경우 외곽 route를 기준으로 도착 지점까지의 거리이다. Junit test에선 이를 간단하게 하기 위해 길이를 10으로 설정했다. Piece가 움직인 거리가 10보다 커지거나 같으면 도착으로 취급한다.
- 경로 변경에 대한 함수 부분에서 Junit test 초기값으로 length를 10으로 주었기에 route 분기 없음을 나타내었다.

■ 백도가 좌표 값이 0인 곳에서 나타났을 경우 다시 0으로 바꾸어 주기 위해 adjustForBackdo 함수도 선언해주었다.

```
// 1. Player가 잘 생성되는가?
@org.junit.jupiter.api.Test  htetarakr
public void testPlayerCreation() {
    Player player = new Player( id: 1,  pieceNum: 2, mockRouteManager);
    assertEquals( expected: 1, player.getId()); // ID 확인
    assertEquals( expected: 2, player.getStandbyPieceCount()); // 대기 중 말 개수 확인
}
```

■ 첫번째 test method는 게임 setting 화면에서 인자를 받을 때 Player 인자가 Interface와 잘 연결되고 ID 및 Player가 소지하고 있는 piece의 개수가 잘 배정되는지 알아보는 method이다.

```
// 2. 말(Piece)이 잘 생성되는가?
@org.junit.jupiter.api.Test  htetarakr
public void testPieceCreation() {
    Player player = new Player( id: 1,  pieceNum: 1, mockRouteManager);
    assertTrue(player.createPiece()); // 말 생성 성공 여부
    assertEquals( expected: 1, player.getPieces().size()); // 생성된 말 개수 확인
}
```

■ 두번째 test method는 piece가 잘 생성되는 지를 알아보는 method이다. 게임 setting 화면에서 handler가 create 함수를 call하는 데 성공했는지를 알려주고 생성한 piece의 개수가 조건과 맞는 지 비교를 통해 piece 생성의 에러 여부를 찾는 method이다. (사진 크기 이슈로 다음 페이지로 넘어가서 다음 method를 설명한다.)

```java
// 3. 도/개/걸/윷/모/백도 이동이 잘 적용되는가?
@org.junit.jupiter.api.Test  htetarakr
public void testPieceMovement() {
    Player player = new Player( id: 1,  pieceNum: 1, mockRouteManager);
    player.createPiece();
    Piece piece = player.getPieces().get(0);

    int route = piece.getRouteIndex();
    int pos = piece.getPositionIndex();

    // 앞으로 이동 (ex. 걸: +4칸)
    assertTrue(player.movePieceAt(route, pos,  move: 4));
    assertEquals( expected: pos + 4, piece.getPositionIndex());

    // 백도 (ex. -1칸 이동)
    pos = piece.getPositionIndex();
    assertTrue(player.movePieceAt(route, pos,  move: -1));
    assertEquals( expected: pos - 1, piece.getPositionIndex());
}
```

- 3번째 test method는 움직임에 대한 test이다. 가장 중요한 test로 말이 잘 움직이는지 판단하는 code이다.
- 우선 Player을 1명, piece를 1개를 생성한다. 판의 모양은 움직임 자체에 영향을 주지 않기에 default로 선택된 사각형 모양의 판이 형성될 것이다.
- 초기 route와 post는 모두 0이다. movePieceAt의 경우 route, pos를 move 값에 따라 조정하는 함수이다.
- '걸'이 나오면 +4 칸을 움직이게 되고 초기 조건에 따라 route = 0, pos = 4가 될 것이라는 것을 예상할 수 있다. 이후 백도가 나오게 된다면 route = 0, pos = 3이 될 것이라는 것을 예상할 수 있다. 만약 movePieceAt method가 작동이 잘 안된다면 위의 예상 값과 실제 값이 다르게 나와 False를 return할 것이다.

```java
// 4. 같은 사용자의 말이 잘 병합되는가? (Grouping)
@org.junit.jupiter.api.Test  htetarakr
public void testPieceGrouping() {
    Player player = new Player( id: 1,  pieceNum: 2, mockRouteManager);
    player.createPiece();
    player.createPiece();
    Piece p1 = player.getPieces().get(0);
    Piece p2 = player.getPieces().get(1);

    // 두 말을 같은 위치로 이동
    p2.setRouteAndPosition(p1.getRouteIndex(), p1.getPositionIndex());

    // 병합 확인
    assertTrue(player.checkAndMergeStack());
    assertEquals( expected: 1, player.getPieces().size()); // 하나로 병합되어야 함
    assertEquals( expected: 2, player.getPieces().get(0).getPoint()); // 포인트도 누적되어야 함
}
```

- 윷놀이에선 자신의 말 위에 다른 자신의 말을 겹칠 수 있다. 이를 grouping이라고 한다. Project에서의 grouping 동작은 덧셈 연산을 통해 이루어진다.
- getPieces.size()는 나가 있는 piece 차지하고 있는 크기이다. 만약 board 위에 piece 위 좌표가 다르다면 이 method의 return 값은 나가 있는 piece의 개수가 된다.
- Grouping이 된 경우 나가 있는 piece의 좌표가 하나로 특정되므로 return 값이 1이 된다. Piece는 각각의 고유 piece_ID를 배정받는다. 위의 시나리오에선 0번 piece위에 1번 piece가 도착하는 경우이다. 이 경우 piece_ID = 0인 piece의 point는 자신의 point + 다른 말의 point (grouping도 고려)가 된다. 자신의 point는 default로 1이 배정되어 있고 다른 piece도 마찬가지로 1이 배정되어 있다. 따라서 예상 return 값은 2가 된다.

```java
// 5. 다른 사용자의 말을 잘 잡는가? (Capture)
@org.junit.jupiter.api.Test  htetarakr
public void testCaptureOpponentPiece() {
    Player attacker = new Player( id: 1,  pieceNum: 2, mockRouteManager);
    Player defender = new Player( id: 2,  pieceNum: 2, mockRouteManager);
    attacker.createPiece();
    defender.createPiece();

    Piece attackerPiece = attacker.getPieces().get(0);
    Piece defenderPiece = defender.getPieces().get(0);

    // 같은 위치로 이동시켜 잡기
    attackerPiece.setRouteAndPosition(defenderPiece.getRouteIndex(), defenderPiece.getPositionIndex());
    assertTrue(defender.captureOpponentPiece(defenderPiece.getRouteIndex(), defenderPiece.getPositionIndex()));

    assertEquals( expected: 0, defender.getPieces().size()); // 잡혔으므로 0
    assertEquals( expected: 2, defender.getStandbyPieceCount()); // 대기 말 수 증가
}
```

- 이 test method는 상대 방의 piece를 잘 잡는지 test하는 method이다. 우선 player을 2명을 생성한다. 각각의 player은 2개의 piece를 가지고 있다.
- 자신의 piece가 상대의 piece를 capture한다는 것은 attack하는 piece가 defend하는 piece의 좌표 위로 가는 것과 동일하다. 이 경우 움직임을 따로 구현하는 것보단 attack 입장에서 defend 입장의 좌표로 setting하는 것이 test 구조가 간단하게 된다.
- 이 test에서 주목해야 하는 것은 잡힌 입장에서의 piece의 size 그리고 대기하고 있는 piece의 개수의 갱신이다. Defend의 piece 하나가 board위에 있었고 결국 잡혔기에 예상되는 defend 측의 piece size는 0이 된다. 초기 값으로 piece를 2개 배정받았기에 다시 대기하는 piece의 개수도 2개 될 것으로 예상할 수 있다.

```java
// 6. 게임 종료 조건 (모든 말을 도착시켰는지) 잘 판별하는가?
@org.junit.jupiter.api.Test  htetarakr
public void testGameFinishCondition() {
    PlayConfig config = new PlayConfig();
    config.setPlayerNum(1);
    config.setPieceNum(1);
    config.setBoardShape(5); // 오각형 보드

    // mockRouteManager를 사용하는 게임 객체 생성
    PlayGame game = new PlayGame(config, mockRouteManager);
    Player player = game.getPlayers().get(0);
    player.createPiece();
    Piece piece = player.getPieces().get(0);

    int len = mockRouteManager.getRouteLength(0);
    piece.setRouteAndPosition( route: 0,  pos: len - 1); // 거의 도착 지점
    player.movePieceAt( x: 0,  y: len - 1,  move: 1); // 도착 처리
    player.checkAndHandleArrival(); // 점수 및 말 제거 처리

    assertEquals( expected: 1, player.getScore()); // 점수가 누적되어야 함
    assertTrue(game.didSomeoneWin()); // 승리자 판별 확인
}
```

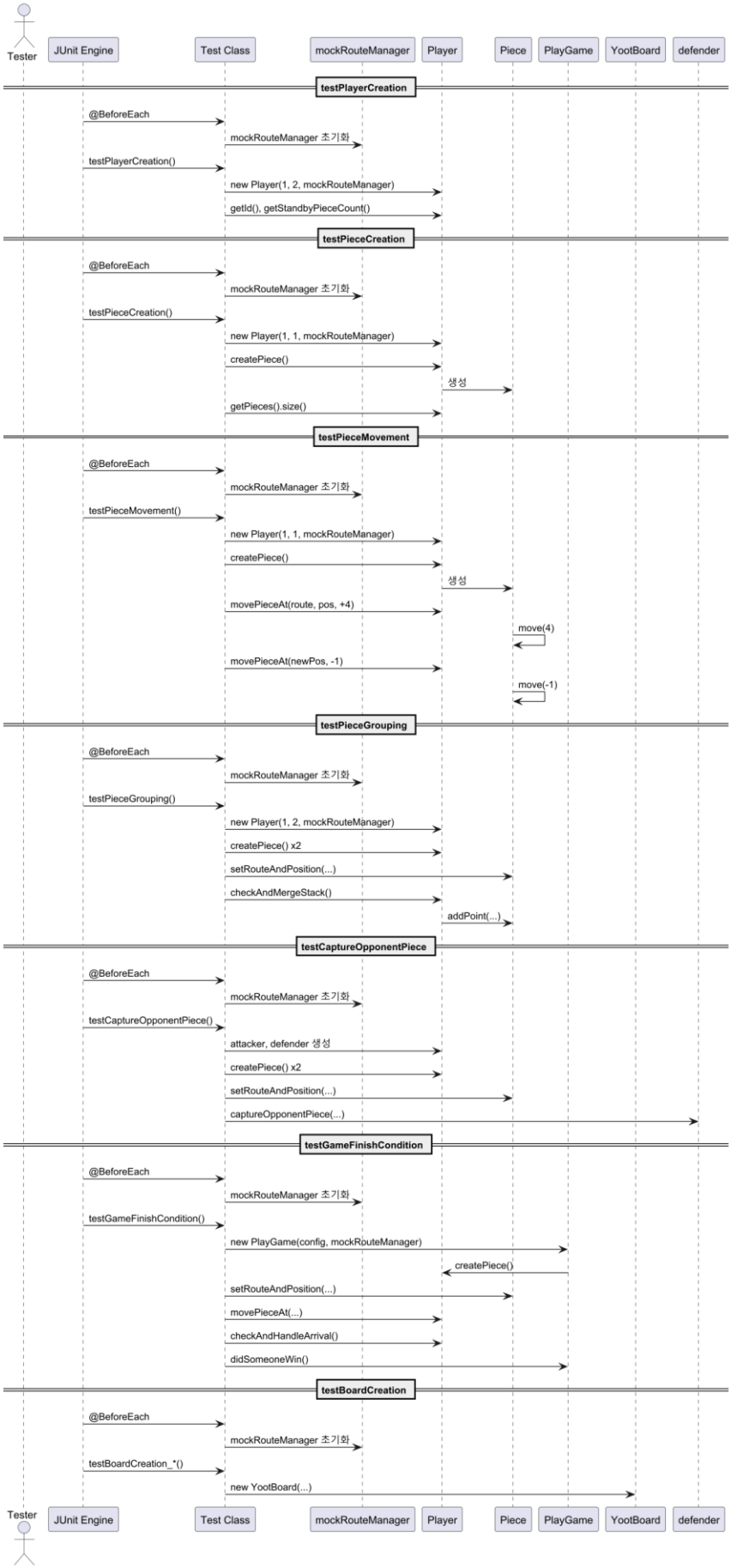
- 이 test method는 승리 조건이 잘 작동하는 지 알아보는 test method이다. Player을 1명 생성하고 piece는 1개 주어진다. 게임은 5각형 board 위에서 진행한다고 가정한다.
- Test의 시나리오는 도착 지점에서 한 칸을 움직였을 때 바로 승리 조건을 만족하는지 판단하는 것이다.
- Piece가 도착 지점에서 한 칸을 이동한 후 player의 score가 1이 증가되었는지 확인하고 didSomeoneWin method를 call하여 승리자를 판별하게 된다.

```java
// 7-1. 사각형 보드 생성
@org.junit.jupiter.api.Test  htetarakr
public void testBoardCreation_Square() {
    YootBoard board = new YootBoard( playerNum: 2,  pieceNum: 2,  boardShape: 4);
    assertNotNull(board);
}

// 7-2. 오각형 보드 생성
@org.junit.jupiter.api.Test  htetarakr
public void testBoardCreation_Penta() {
    YootBoard board = new YootBoard( playerNum: 2,  pieceNum: 2,  boardShape: 5);
    assertNotNull(board);
}

// 7-3. 육각형 보드 생성
@org.junit.jupiter.api.Test  htetarakr
public void testBoardCreation_Hexa() {
    YootBoard board = new YootBoard( playerNum: 2,  pieceNum: 2,  boardShape: 6);
    assertNotNull(board);
}
```

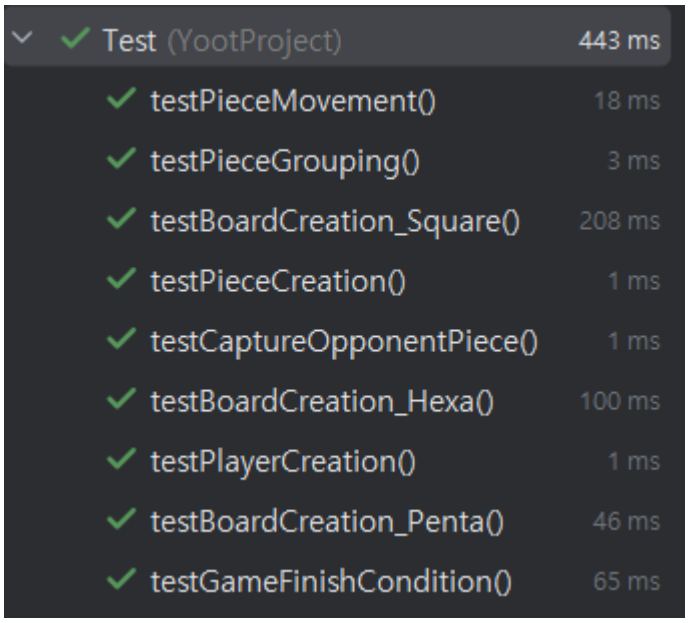
- 마지막 test는 각 모양별 board가 잘 생성되는 지이다. Board를 생성할 때 parameter을 3개를 넘긴다. Parameter가 잘 넘어갔으면 생성이 되었을 것이고 그렇지 않은 경우 할당을 실패하여 Null이 될 것이다. 이러한 시나리오를 고려하여 assserNotNull(board) method를 이용하였다.

### C) 테스트 진행과정

- 모든 test method가 실행되기 전엔 @BeforeEach decorator로 지정된 setup method를 거치게 된다.
- 각 test method는 실행 순서가 sequence하게 실행된다. (가상머신이 sequence를 임의로 바꾸어 실행하기도 한다.)
- 다음 페이지에 Junit test에 대한 Sequence diagram이 있다.


Tester
JUnit Engine
Test Class
mockRouteManager
Player
Piece
PlayGame
YootBoard
defender
testPlayerCreation
@BeforeEach
mockRouteManager 초기화
testPlayerCreation()
new Player(1, 2, mockRouteManager)
getId(), getStandbyPieceCount()
testPieceCreation
@BeforeEach
mockRouteManager 초기화
testPieceCreation()
new Player(1, 1, mockRouteManager)
createPiece()
생성
getPieces().size()
testPieceMovement
@BeforeEach
mockRouteManager 초기화
testPieceMovement()
new Player(1, 1, mockRouteManager)
createPiece()
생성
movePieceAt(route, pos, +4)
move(4)
movePieceAt(newPos, -1)
move(-1)
testPieceGrouping
@BeforeEach
mockRouteManager 초기화
testPieceGrouping()
new Player(1, 2, mockRouteManager)
createPiece() x2
setRouteAndPosition(...)
checkAndMergeStack()
addPoint(...)
testCaptureOpponentPiece
@BeforeEach
mockRouteManager 초기화
testCaptureOpponentPiece()
attacker, defender 생성
createPiece() x2
setRouteAndPosition(...)
captureOpponentPiece(...)
testGameFinishCondition
@BeforeEach
mockRouteManager 초기화
testGameFinishCondition()
new PlayGame(config, mockRouteManager)
createPiece()
setRouteAndPosition(...)
movePieceAt(...)
checkAndHandleArrival()
didSomeoneWin()
testBoardCreation
@BeforeEach
mockRouteManager 초기화
testBoardCreation_*()
new YootBoard(...)
Tester
JUnit Engine
Test Class
mockRouteManager
Player
Piece
PlayGame
YootBoard
defender

### D) 테스트 결과




- Test 결과 모든 Junit test에서 예외 없이 성공했음을 볼 수 있다.
- Test 시간은 443ms이고 test 중에서 가장 오래 걸린 것은 동적 할당이 필요한 board creation이다. 여기서 의문인 것이 왜 사각형 board가 가장 오래 걸리는 것일 것이다. 그 이유는 사각형 원판의 경우 Design (코드 디자인이 아닌 외형적 디자인)에 가장 많이 신경을 썼고 img 파일을 많이 불러오기 때문에 가장 오래 걸린 것이다.

## 3. Conclusion (Junit Version간 문법의 차이)

- Junit test를 진행할 때 test package의 version을 꼭 알아야 한다.
- 진행한 Junit test의 경우 수업시간에 나온 decorator인 @Before이 아닌 Junit test 5 version의 표현 방식인 @BeforeEach를 사용한다.
- 이전 버전과는 다르게 Junit 5는 Jupiter 노트북의 package를 import해야한다.
- 기존의 Junit 4 버전과 5 버전의 코드를 혼용하여 작성하고 싶은 경우 junit-vintage-engine 의존성을 코드 파일에 추가적으로 migration 해야 한다.
- 다음 페이지는 Notion에 정리한 Junit 4과 Junit 5 버전의 큰 차이점에 대해 정리한 내용이다.

## ✅ 1. 주요 애너테이션 비교

| 기능 | JUnit 4 | JUnit 5 (`org.junit.jupiter.api`) |
|---|---|---|
| 테스트 메서드 | `@Test` | `@Test` |
| 테스트 전 설정 (각 메서드 전) | `@Before` | `@BeforeEach` |
| 테스트 후 정리 (각 메서드 후) | `@After` | `@AfterEach` |
| 전체 테스트 전 1회 실행 | `@BeforeClass` (static 필요) | `@BeforeAll` (static or non-static) |
| 전체 테스트 후 1회 실행 | `@AfterClass` (static 필요) | `@AfterAll` (static or non-static) |
| 예외 테스트 | `@Test(expected=...)` | `assertThrows()` 메서드 사용 |
| 타임아웃 설정 | `@Test(timeout=...)` | `assertTimeout()` 메서드 사용 |
| 테스트 무시 | `@Ignore` | `@Disabled` |

■ 위와 같이 Annotation의 경우 더 자세하게 바뀌었다.

1. **JUnit Platform**
   - 테스트 실행을 위한 런타임 플랫폼 (예: IDE, build tool 등과 연동)
2. **JUnit Jupiter**
   - JUnit 5의 **새 애너테이션 기반 API** 제공 (테스트 작성 시 사용하는 부분)
3. **JUnit Vintage**
   - 기존 JUnit 3 & 4 테스트를 실행할 수 있도록 지원하는 호환 모듈

■ Junit 5의 구성 요소는 위와 같이 크게 세 가지로 나뉘게 된다.

■ 또한 Junit 5의 경우 Back-End 개발자의 편의성을 고려해 Gradle, Maven builder을 통해 다양한 assertion, Tag filtering, Parameterized test를 지원한다.

■ Junit 5를 사용할 때 builder가 Spring boot 기반이 아니더라도 일반적인 pure Java에서도 돌아간다.