Tyler Jones

Final Project: Write-Up

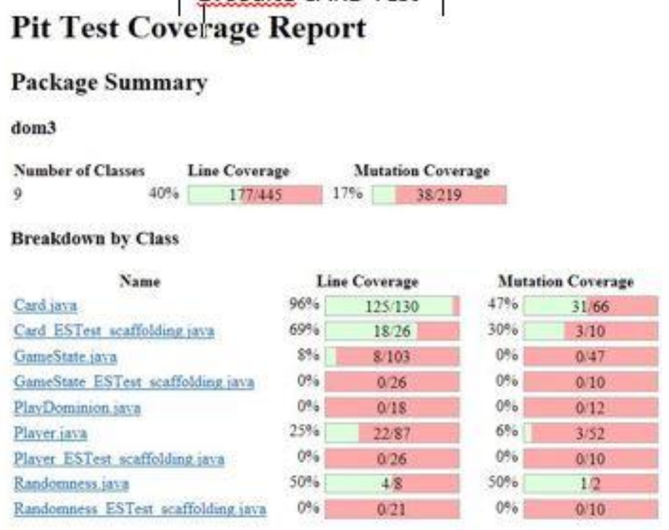
CS 362

3/19/17

PART B)

i) For this assignment, I developed and used PITest in Eclipse. PITest actually has an Eclipse plug-in that was extremely easy and great for me to use, however it had one downside. It is only able to generate reports for one test at a time. This was a simple enough work around for my own Junit tests, because I was able to put all of my tests into one test suite, therefore when PITest ran on my one test, it generated the mutation coverage for all my code like I want. However, for the EvoSuite auto test generation, this proved more difficult. Evosuite generates on a class by class basis. I wasn't able to consolidate all of these into one test file and so I was only able to run PITest on each class at a time. Therefore, I had to break it down into 4 separate PITest reports for my EvoSuite generated tests. Those can be seen below.

EvoSuite CARD Test



Pit Test Coverage Report

Package Summary

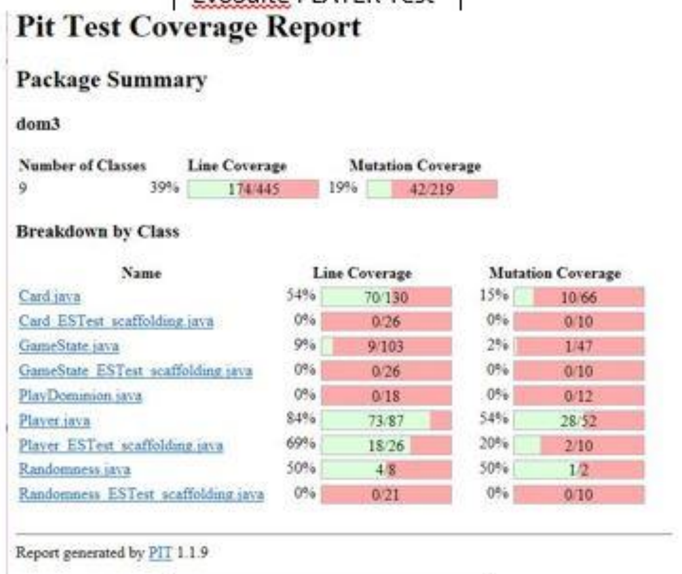
dom3

Number of Classes	Line Coverage		Mutation Coverage	
9	40%	177/445	17%	38/219

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
Card.java	96%	125/130	47%	31/66
Card_ESTest_scaffolding.java	69%	18/26	30%	3/10
GameState.java	8%	8/103	0%	0/47
GameState_ESTest_scaffolding.java	0%	0/26	0%	0/10
PlayDominion.java	0%	0/18	0%	0/12
Player.java	25%	22/87	6%	3/52
Player_ESTest_scaffolding.java	0%	0/26	0%	0/10
Randomness.java	50%	4/8	50%	1/2
Randomness_ESTest_scaffolding.java	0%	0/21	0%	0/10

EvoSuite PLAYER Test



Pit Test Coverage Report

Package Summary

dom3

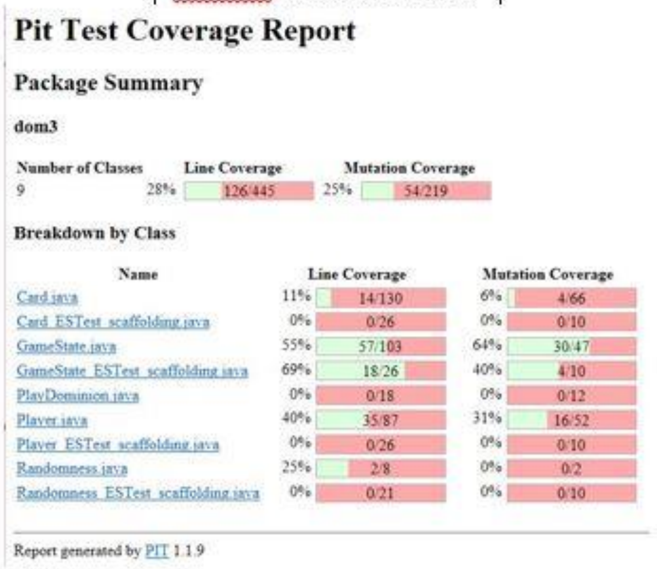
Number of Classes	Line Coverage		Mutation Coverage	
9	39%	174/445	19%	42/219

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
Card.java	54%	70/130	15%	10/66
Card_ESTest_scaffolding.java	0%	0/26	0%	0/10
GameState.java	9%	9/103	2%	1/47
GameState_ESTest_scaffolding.java	0%	0/26	0%	0/10
PlayDominion.java	0%	0/18	0%	0/12
Player.java	84%	73/87	54%	28/52
Player_ESTest_scaffolding.java	69%	18/26	20%	2/10
Randomness.java	50%	4/8	50%	1/2
Randomness_ESTest_scaffolding.java	0%	0/21	0%	0/10

Report generated by PIT 1.1.9

EvoSuite GAMESTATE Test



Pit Test Coverage Report

Package Summary

dom3

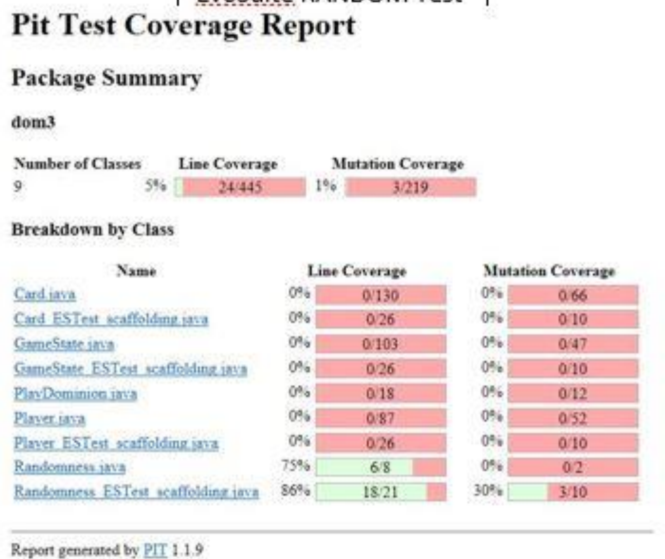
Number of Classes	Line Coverage		Mutation Coverage	
9	28%	126/445	25%	54/219

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
Card.java	11%	14/130	6%	4/66
Card_ESTest_scaffolding.java	0%	0/26	0%	0/10
GameState.java	55%	57/103	64%	30/47
GameState_ESTest_scaffolding.java	69%	18/26	40%	4/10
PlayDominion.java	0%	0/18	0%	0/12
Player.java	40%	35/87	31%	16/52
Player_ESTest_scaffolding.java	0%	0/26	0%	0/10
Randomness.java	25%	2/8	0%	0/2
Randomness_ESTest_scaffolding.java	0%	0/21	0%	0/10

Report generated by PIT 1.1.9

EvoSuite RANDOM Test



Pit Test Coverage Report

Package Summary

dom3

Number of Classes	Line Coverage		Mutation Coverage	
9	5%	24/445	1%	3/219

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
Card.java	0%	0/130	0%	0/66
Card_ESTest_scaffolding.java	0%	0/26	0%	0/10
GameState.java	0%	0/103	0%	0/47
GameState_ESTest_scaffolding.java	0%	0/26	0%	0/10
PlayDominion.java	0%	0/18	0%	0/12
Player.java	0%	0/87	0%	0/52
Player_ESTest_scaffolding.java	0%	0/26	0%	0/10
Randomness.java	75%	6/8	0%	0/2
Randomness_ESTest_scaffolding.java	86%	18/21	30%	3/10

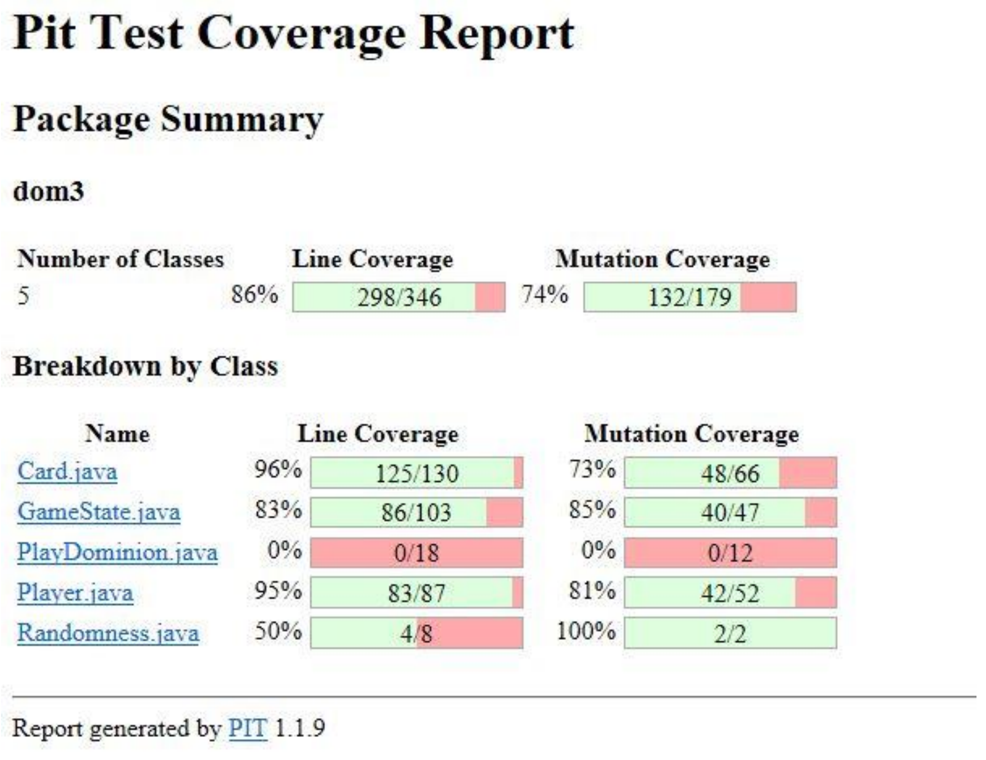
Report generated by PIT 1.1.9

The 4 main class reports can be seen above. In each test, the corresponding bar with the test it belongs to is the indicator of the mutation rate for that EvoSuite Generated test. For example, in my Evosuite Card test report, the Card.java percentage is the indicator that we care about. For Card it was 47%. If we were to consolidate all of the bars that indicate the mutation coverage for each test into one report, it would look something like this.

Card.java = 47%
Player.java 54%
Gamestate.java = 64%
Random.java = 0%

The evosuite generated tests only killed 54% of player mutants, 47% of card mutants, 64% of Gamestate mutants, and 0% of my random mutants, although the last could fluctuate for any given PITest generation depending on the mutants that it generates. These figures are what we will take into consideration when comparing to our own tests.

For my own unit test, it was much simpler to get an effective report from PITest, especially since I put my tests for all of my classes into one large test suite so it could be run from Eclipse without the issues seen above of having to generate 4 separate reports. The mutation coverage for my unit tests can be seen below.



Pit Test Coverage Report

Package Summary

dom3

Number of Classes	Line Coverage		Mutation Coverage	
5	86%	298/346	74%	132/179

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
Card.java	96%	125/130	73%	48/66
GameState.java	83%	86/103	85%	40/47
PlayDominion.java	0%	0/18	0%	0/12
Player.java	95%	83/87	81%	42/52
Randomness.java	50%	4/8	100%	2/2

Report generated by PIT 1.1.9

Based on these figures both for the EvoSuite generated tests, and my own unit tests, I would come to the initial conclusion that test generation software has an inherent flaw about it. Because I do not handle the test generation, nor do I handle the mutants generated by PITest, there is effectively nothing I can do to improve my mutation coverage for EvoSuite tests. Since mutation coverage is intended to indicate the overall quality of a test, it would be reasonable to conclude that EvoSuite's tests are less comprehensive and lower quality than writing them myself, at least for a Dominion implementation.

Writing my own unit tests was much more efficient at generating higher mutation coverage, and thus it can be concluded that writing your own unit tests is the best way to comprehensively cover your code in a way that results in an effective test environment.

My data shows that 74% of my overall mutants were killed. More specifically, 73% card mutants were killed, 85% GameState mutants were killed, 0% PlayDominion mutants were killed (this is because it doesn't need to be tested, it just plays a game), 81% of player mutants were killed, and 100% of random mutants were killed. All of these values are higher than the EvoSuite generated tests listed previously. Additionally, my code coverage from my unit tests was higher than the coverage generated by EvoSuite's tests.

In terms of mutants that survived, upon closer inspection, many of them were mutants that I was unable to control either due to randomness, or due to intermittent function behavior that doesn't have end result assert qualities. For example, within my buyCard function in my Player.java, I have a list of if statements that are based on the values of local variables within the function. I am unable to set up assert statements for the expected values of these local variables, and thus when PITest mutates my if conditions on them, their mutants survive. This was a problem I noticed throughout my code with surviving mutants. There weren't any mutants that survived in any of my classes due to me not having coverage or a test for it. Only problems like I previously stated resulted in survived mutants.

ii) Before deciding to use PITest through Eclipse, I only looked at using PITest through the maven command line. I resolved to use PITest without exploring other options due to my feeling like I understood the concepts behind it better than the other tools presented to us.

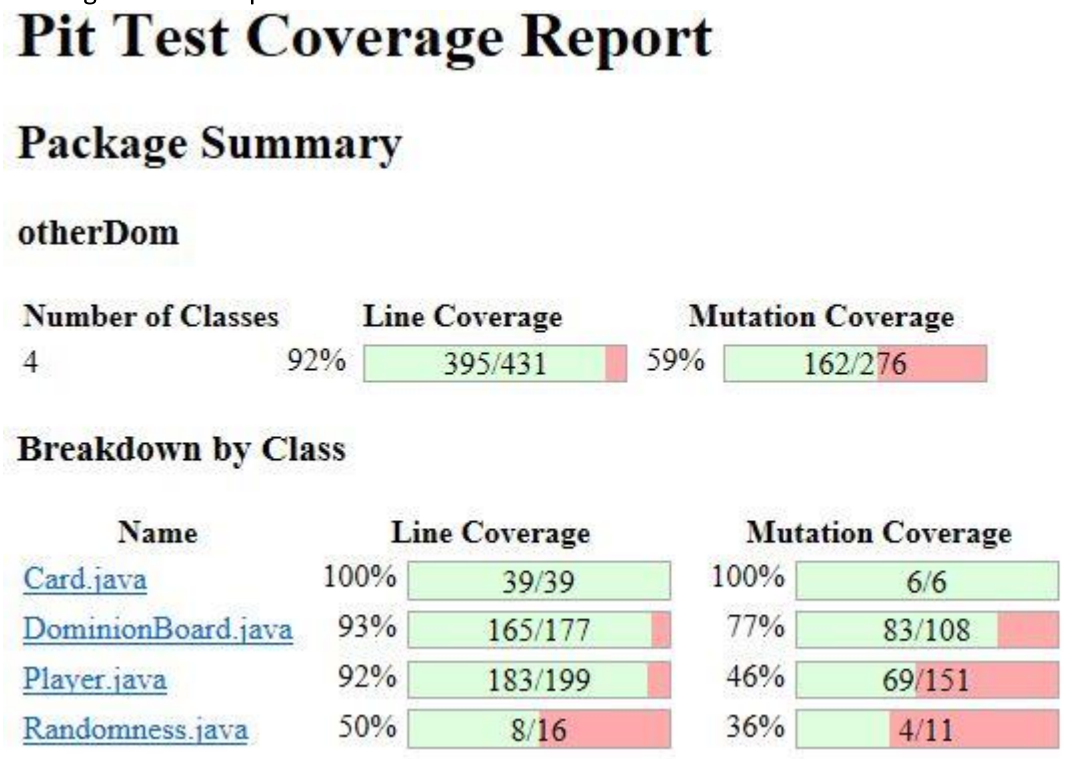
iii) The only problem I faced when using this tool was with EvoSuite. Initially, when I tried to use it from maven on my EvoSuite tests to compile one mutation coverage report for all my classes at once, as would be desirable, it wasn't able to find my EvoSuite tests. It was only once I made it into the PIT Eclipse plugin that I was able generate a coverage report for my EvoSuite tests, however this came with the previously described limitation of only being able to generate coverage on one class at a time.

iv) Once PITest was installed properly, it worked pleasantly well on all my tests. It gave me effective error messages during development to tell me what is going wrong with my most recently implemented test. I would recommend PITest through Eclipse to anyone wanting to generate a mutation coverage report for their tests.

PART B)

v) A) Testing Dominion has turned out to be a pretty frustrating experience. Not due to the nature of Dominion, but rather due to all of the software contingencies and the setting up of our environment that we were pretty much left to figure out for ourselves. Once the environment has been set up for mvn, Eclipse, cobertura, EvoSuite, and PITest, all working with each other and on your code properly, testing Dominion has been a fun and informative experience.

B) The Dominion code that I picked to test was very well written and documented. It was slightly different than my own, mostly in the way that we handled the GameState and GameBoard functions. "hellwegk"'s code was very clean, easy to read, and had great code coverage. PITest's report of her code can be seen below



Pit Test Coverage Report

Package Summary

otherDom

Number of Classes	Line Coverage		Mutation Coverage	
4	92%	395/431	59%	162/276

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
Card.java	100%	39/39	100%	6/6
DominionBoard.java	93%	165/177	77%	83/108
Player.java	92%	183/199	46%	69/151
Randomness.java	50%	8/16	36%	4/11

As can be seen, her code coverage is very high, and her mutation coverage is also sufficient. She has a substantial amount of surviving mutants from DominionBoard, Player, and Randomness. While she has 100% mutation coverage for her Card class.

vi) The only problems I faced when generating tests for my classmate's code was figuring out how to initialize variables in order to test them. The way she set it up versus how I set it up was fairly different in nature so I had to look at her tests in order to get a better idea of how to go about it

vii) I faced little to no problems testing my class mate's code. The implementation of Dominion was clean, correct, and easy to read, in addition to her UnitTest for this assignment being easy to read and take from as well in order to test things myself.

NOTE: The assignment calls for BugsReport.pdf to be submitted on Canvas. It states this file needs to be submitted to Canvas as well, but there is only one Canvas submission slot. I have placed both files separately in my GitHub submission, but BugsReport.pdf is copy pasted at the end of this document due to the Canvas limitation.

Tyler Jones

Bugs Report

3/19/17

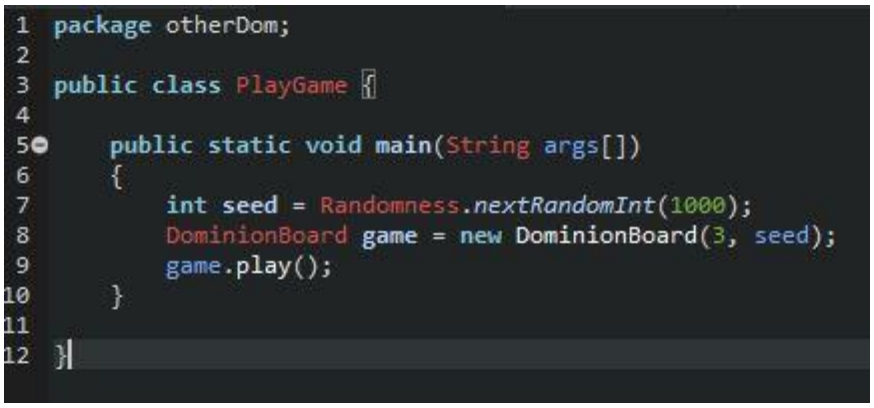
CS 362: Final Project

For Part-B of this assignment I used the ONID username from GitHub "**hellwegk**" for finding bugs.

I found two large bugs in her implementation.

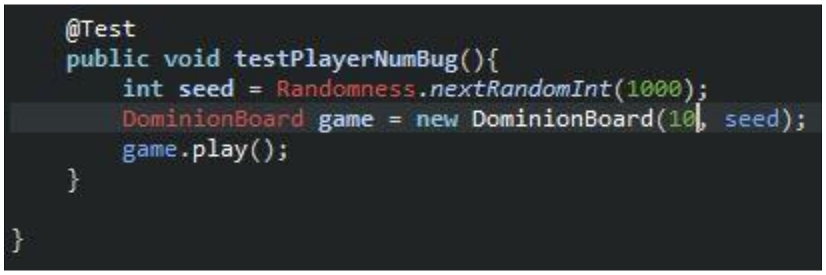
BUG 1

The first bug I found was very high level and easy to spot. In order to find both bugs for this assignment, I started from the highest level possible and moved down. I looked at her PlayDominion.java file to see how she actually called a game and played it fully. I noticed that she passed a number and a seed into a new Gameboard and then called play on the Gameboard. This can be seen below.

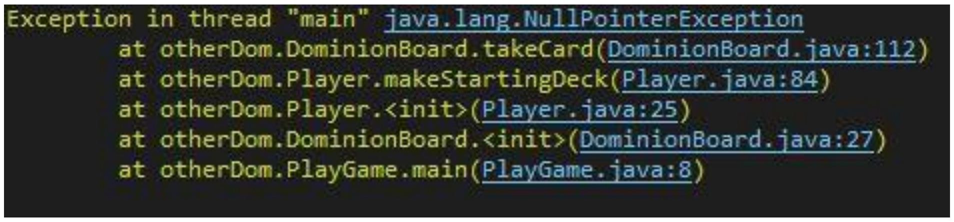


```java
package otherDom;

public class PlayGame {

    public static void main(String args[])
    {
        int seed = Randomness.nextRandomInt(1000);
        DominionBoard game = new DominionBoard(3, seed);
        game.play();
    }

}
```

I figured I would try to edit the number of players passed into the first argument of DominionBoard to a number that isn't valid for a Dominion game. I changed it to 10, and it was then I found my first bug. When calling anywhere from 2-4 as this argument, the game plays successfully, as it should, but the game gives me a NullPointerException when I made this value too large, as the code isn't configured to handle that many players. My input and her output that revealed the bug can be seen below.

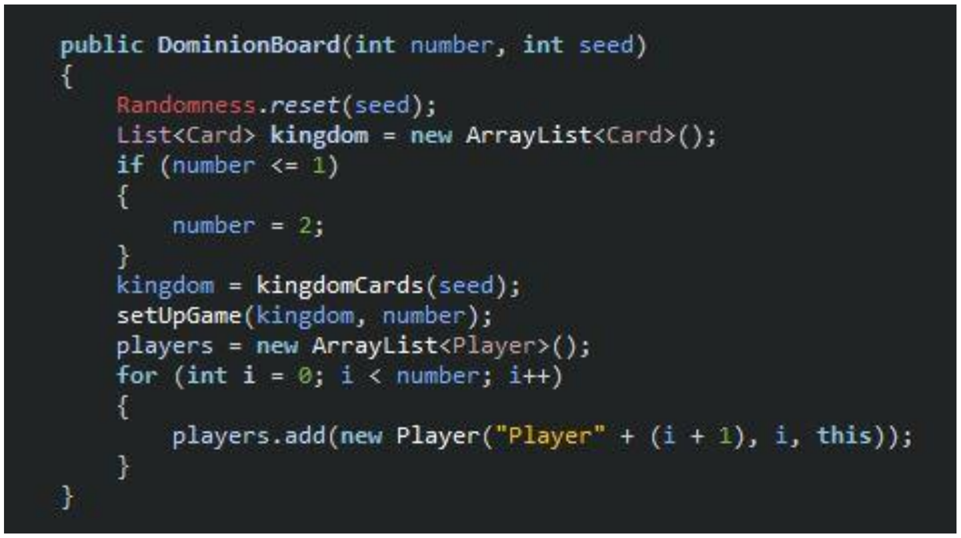


```
    @Test
    public void testPlayerNumBug(){
        int seed = Randomness.nextRandomInt(1000);
        DominionBoard game = new DominionBoard(10, seed);
        game.play();
    }

}
```



```
Exception in thread "main" java.lang.NullPointerException
	at otherDom.DominionBoard.takeCard(DominionBoard.java:112)
	at otherDom.Player.makeStartingDeck(Player.java:84)
	at otherDom.Player.<init>(Player.java:25)
	at otherDom.DominionBoard.<init>(DominionBoard.java:27)
	at otherDom.PlayGame.main(PlayGame.java:8)
```

The solution to her bug would be when she initializes her game object that she set up constraints to be able to handle a number that is too large. She has if statements within her code to handle numbers that are too small (0-1), but not that are too large. It is an easy fix but one that needs to be made. The area where the second if statement for values that are too large should be added can be seen below.

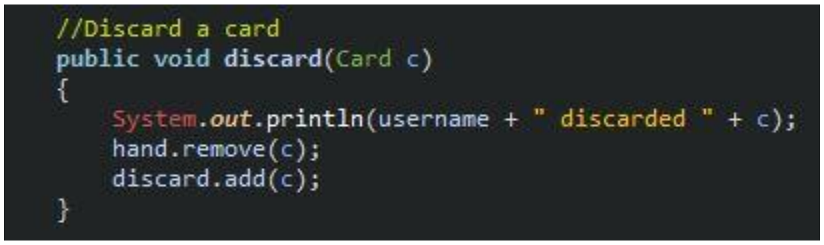


```
    public DominionBoard(int number, int seed)
    {
        Randomness.reset(seed);
        List<Card> kingdom = new ArrayList<Card>();
        if (number <= 1)
        {
            number = 2;
        }
        kingdom = kingdomCards(seed);
        setUpGame(kingdom, number);
        players = new ArrayList<Player>();
        for (int i = 0; i < number; i++)
        {
            players.add(new Player("Player" + (i + 1), i, this));
        }
    }
```

By simply adding a **if(number >4){ number = 4 };** statement, that is very similar to her existing if statement for values that are too small, this bug could be fixed.

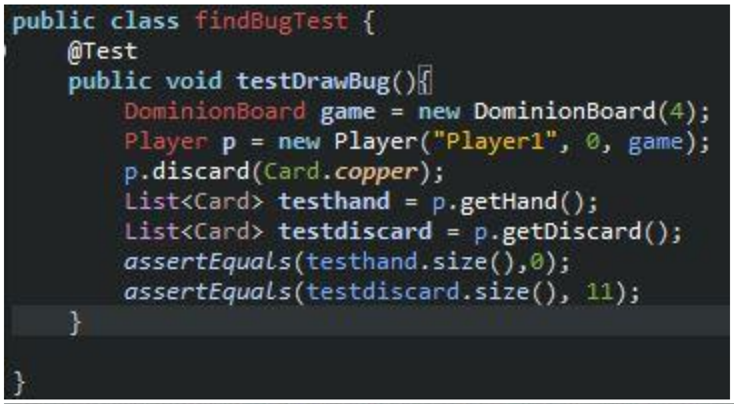
BUG 2

The second bug I found was more subtle than the first, but is still a bug. I was looking through her code as to how she handled drawing and discarding cards. I found her code that allows her to discard a card and that can be seen below.

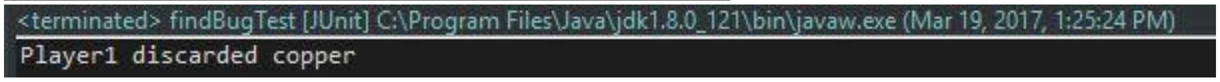


```
    //Discard a card
    public void discard(Card c)
    {
        System.out.println(username + " discarded " + c);
        hand.remove(c);
        discard.add(c);
    }
```

Clearly, her code doesn't have any conditionals setup for the necessary state of the hand and discard piles at the time of discarding. This function just takes the current state of the hand, and removes the passed card, and takes the current state of the discard and adds the passed card. I took this as a bug. I wrote a unit test that lets me generate a player, and lets me discard a card that I have not yet added to my hand. Her code allows me to remove from my hand a card that I don't have. My unit test and the successful output of the test can be seen below.



```
public class findBugTest {
    @Test
    public void testDrawBug(){
        DominionBoard game = new DominionBoard(4);
        Player p = new Player("Player1", 0, game);
        p.discard(Card.copper);
        List<Card> testhand = p.getHand();
        List<Card> testdiscard = p.getDiscard();
        assertEquals(testhand.size(),0);
        assertEquals(testdiscard.size(), 11);
    }

}
```



```
<terminated> findBugTest [JUnit] C:\Program Files\Java\jdk1.8.0_121\bin\javaw.exe (Mar 19, 2017, 1:25:24 PM)
Player1 discarded copper
```

In my unit test, I used Junit asserts in order to test the size of my hand and the size of my discard pile after discarding a copper card. This test passes, implying the hand doesn't empty to below 0 and doesn't catch an error, and the discard size increases from 10 to 11. The problem with this implementation is that because copper was clearly never added to player P's hand in this unit test, the discard behavior shouldn't exist.

Like in Bug 1, the fix for this bug would be to add conditional statements within her code to make sure that the card being discarded is first in the player's hand. This doesn't ever result in an output that wouldn't compile, like Bug 1 does, but rather this bug is subtle and simply undesired behavior for what would be considered a sound discard functionality.