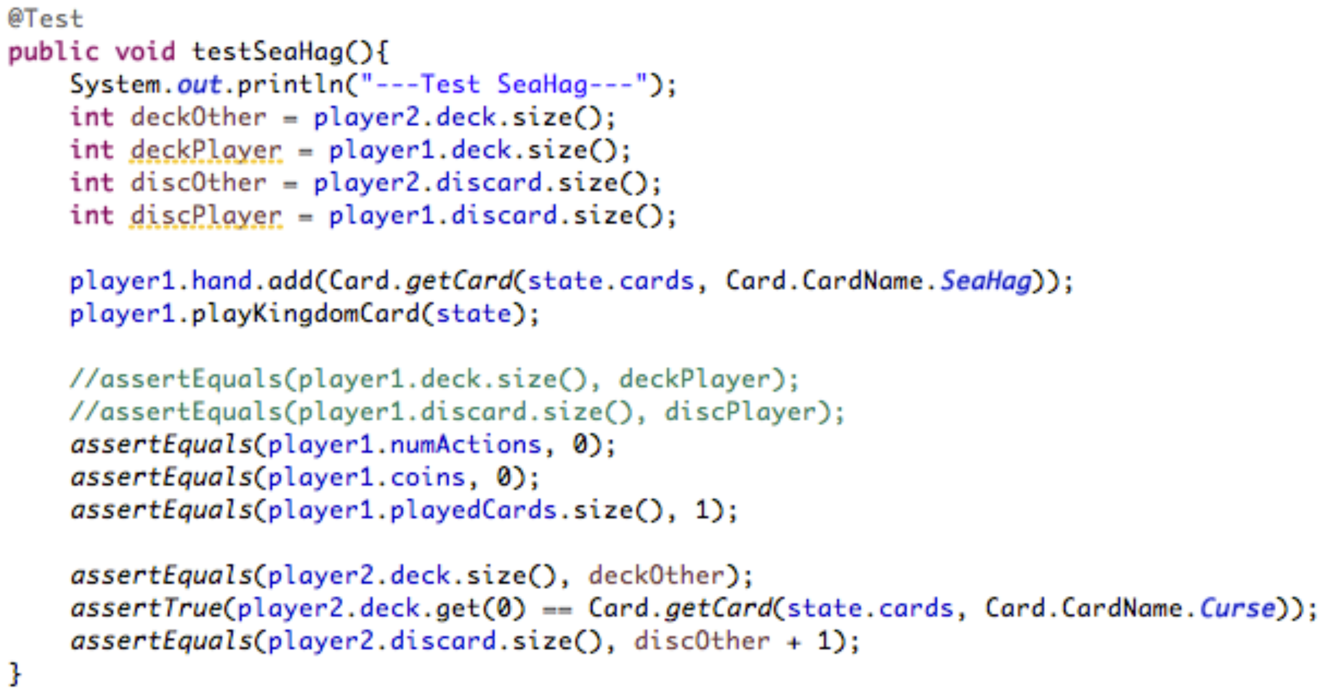
Assignment 1- Dominion
Nathan Shepherd

Card Testing:

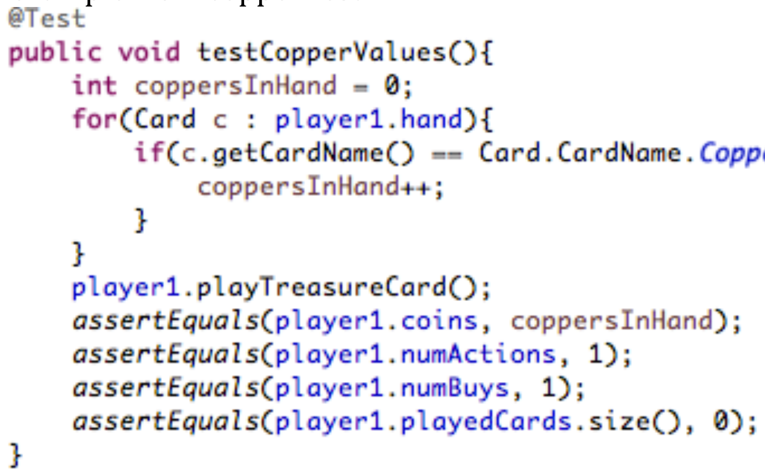
For card testing I tested each of the cards with a separate test in my cardTest junit file. Through my testing I found several bugs that I had looked over, including taking cards while ignoring how many there were, so in multiple play-throughs my curse count would end up negative.



```
@Test
public void testSeaHag(){
    System.out.println("---Test SeaHag---");
    int deckOther = player2.deck.size();
    int deckPlayer = player1.deck.size();
    int discOther = player2.discard.size();
    int discPlayer = player1.discard.size();

    player1.hand.add(Card.getCard(state.cards, Card.CardName.SeaHag));
    player1.playKingdomCard(state);

    //assertEquals(player1.deck.size(), deckPlayer);
    //assertEquals(player1.discard.size(), discPlayer);
    assertEquals(player1.numActions, 0);
    assertEquals(player1.coins, 0);
    assertEquals(player1.playedCards.size(), 1);

    assertEquals(player2.deck.size(), deckOther);
    assertTrue(player2.deck.get(0) == Card.getCard(state.cards, Card.CardName.Curse));
    assertEquals(player2.discard.size(), discOther + 1);
}
```

This is an example of my test for sea hag. Two asserts are commented out as they fail in a bug that I left in my program. Sea hag will give curses to everyone, including the player who played the card.

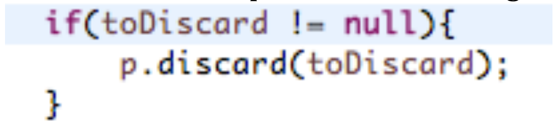
I also tested the treasure cards for value, and how they affected a players hand.
An example from copperTest:



```
@Test
public void testCopperValues(){
    int coppersInHand = 0;
    for(Card c : player1.hand){
        if(c.getCardName() == Card.CardName.Copper){
            coppersInHand++;
        }
    }
    player1.playTreasureCard();
    assertEquals(player1.coins, coppersInHand);
    assertEquals(player1.numActions, 1);
    assertEquals(player1.numBuys, 1);
    assertEquals(player1.playedCards.size(), 0);
}
```

Overall my testing for the card class covered 96% of the instructions and 96% of the card instructions in play, a large switch statement. From class, I would think that this is a good amount of coverage, and

I can see that the lines not covered are mostly inaccessible, if statements that make sure the card isn't null, which are implausible to reach, given enum and other restrictions.



```
if(toDiscard != null){
    p.discard(toDiscard);
}
```

For input I would set up the player hand to best represent the use of the card, then play the card. I would then assert on each action of the card, as well as check that the other player elements hadn't changed.

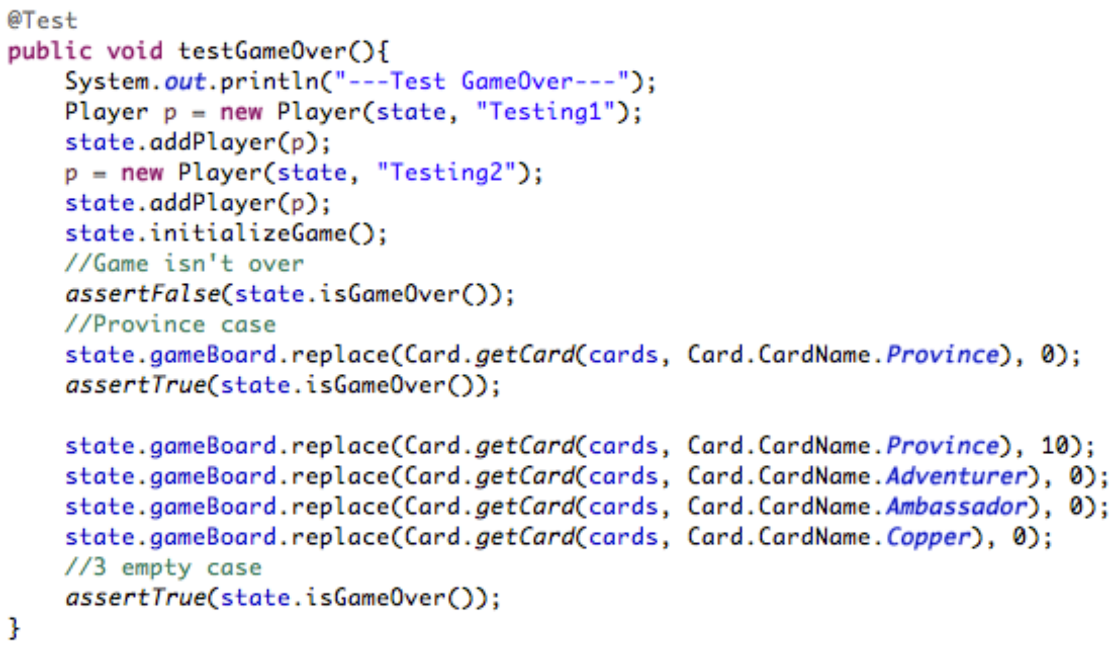
Player Testing:

In player testing, I tested each of the functions with one test function each. Input was set up before hand with the players variables. I would check after running the function that it had operated properly, without changing the other player or state variables. A bug that stood out was where if a player had no coin and a single buy, it would default to just taking a copper. If there was coin then it would buy a random card, checking cost and if there was an embargo token on it. The bug was that if the player got the default copper, it would ignore any embargo token. I discovered this through static analysis rather than unit tests as it is unlikely to come up.

As far as coverage, for player testing I had 99% instruction coverage, most of the instructions missed were from cases where the code checked for null output from card creation, which didn't ever occur and was difficult to test for, without rewriting the card class.

GameState Testing:

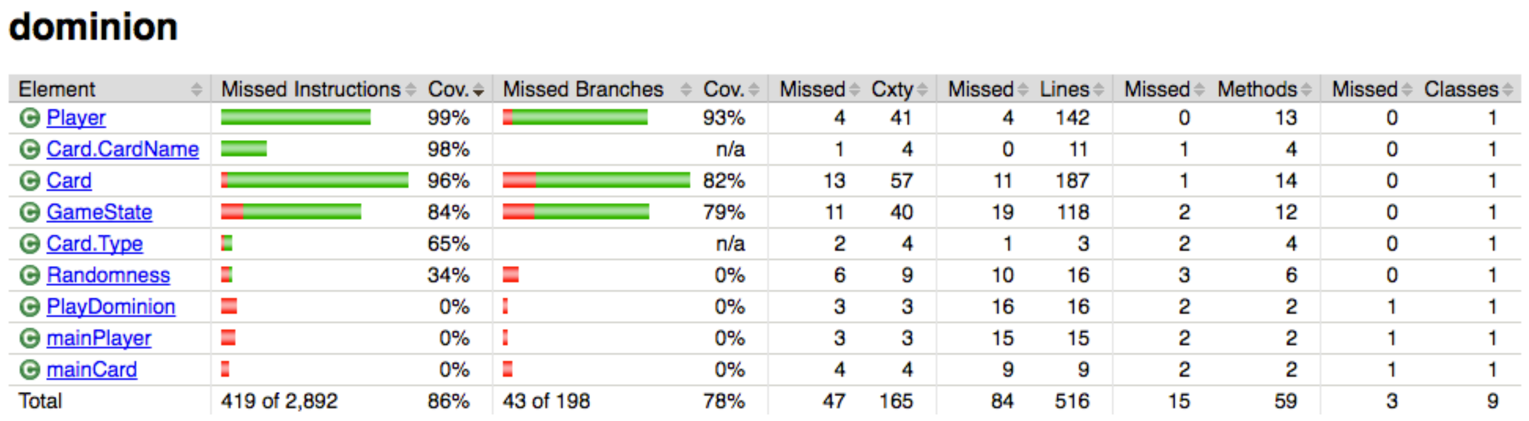
Game state testing was done just the same as the others. I tested all of the functions, mostly unchanged from the provided code. For inputs, I would set the gamestate and test the functions changes that it had made.



```
@Test
public void testGameOver(){
    System.out.println("---Test GameOver---");
    Player p = new Player(state, "Testing1");
    state.addPlayer(p);
    p = new Player(state, "Testing2");
    state.addPlayer(p);
    state.initializeGame();
    //Game isn't over
    assertFalse(state.isGameOver());
    //Province case
    state.gameBoard.replace(Card.getCard(cards, Card.CardName.Province), 0);
    assertTrue(state.isGameOver());

    state.gameBoard.replace(Card.getCard(cards, Card.CardName.Province), 10);
    state.gameBoard.replace(Card.getCard(cards, Card.CardName.Adventurer), 0);
    state.gameBoard.replace(Card.getCard(cards, Card.CardName.Ambassador), 0);
    state.gameBoard.replace(Card.getCard(cards, Card.CardName.Copper), 0);
    //3 empty case
    assertTrue(state.isGameOver());
}
```

Coverage:

I used EMMA for my coverage, because I'm working on a mac, and I heard cobertura was difficult to use. Overall I think I had good test coverage, but I didn't test everything that was run, as in some of the line ran weren't tested with asserts. I think that code coverage is a good gauge for testing, but not the only thing one should look at. I look forward to delving into random and search based testing.



dominion

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Player		99%		93%	4	41	4	142	0	13	0	1
Card.CardName		98%		n/a	1	4	0	11	1	4	0	1
Card		96%		82%	13	57	11	187	1	14	0	1
GameState		84%		79%	11	40	19	118	2	12	0	1
Card.Type		65%		n/a	2	4	1	3	2	4	0	1
Randomness		34%		0%	6	9	10	16	3	6	0	1
PlayDominion		0%		0%	3	3	16	16	2	2	1	1
mainPlayer		0%		0%	3	3	15	15	2	2	1	1
mainCard		0%		0%	4	4	9	9	2	2	1	1
Total	419 of 2,892	86%	43 of 198	78%	47	165	84	516	15	59	3	9

Random Psuedo-Code:

As I mentioned before unit testing with code coverage as a metric is pretty limiting. It took just as much time to write the code as it did to write the unit tests, and this could be much faster with randomization.

```
List = {Initialization calls}
List = {Functions }

For(Random times)
    //Write test
    choose random{initialization calls}
    for(random times)
        choose random {functions, input}
```

The problem with random testing would be associating initialization calls with functions with inputs allowed, because a range is good but it still has to be the right data type.