Dominion in Java

Gabriel Jonas

CS362, Winter 2017

I. Introduction

This project is to create an implementation of the Dominion board game in Java (with accompanying unit tests and a code coverage report). This paper will be broken up into three sections, namely the unit tests and choice of input, results of unit tests, a summary of information regarding the code coverage, and a possible implementation of random testing.

II. Unit Testing

For my implementation of the game, I decided to focus on testing the implementation rather than the setup of each individual item. The unit testing for this project follows a common format: set up the players, gamestate, and cards; perform one implemented API call; and test the changes after the API call is completed. While this method is not the most efficient, it gives a clean state for testing each implementation. An example of this is shown below with the Sea Hag.

Listing 1. Sea Hag Test

```
package org.cs362.dominion;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;

import org.junit.Test;
import static org.junit.Assert.*;


public class SeaHagTest {

    @Test
    public void test0()  throws Throwable  {
        Randomness.reset();

        int newCards = 1;
        // initialize a game state and player cards
        List<Card> cards = new ArrayList<Card>(Card.createCards());
        GameState state = new GameState(cards);

        Player player = new Player(state, "player-1");
        player.hand.add(Card.getCard(cards,Card.CardName.SeaHag));
```

```
state.addPlayer(player);

player = new Player(state, "player-2");
state.addPlayer(player);

//Initialize the game!
state.initializeGame();

//copy or clone the game state to a test case
GameState testState = (GameState) state.clone();

//to shorten naming conventions
Player p1 = state.players.get(0);
Player p2 = state.players.get(1);
Player p1t = testState.players.get(0);
Player p2t = testState.players.get(1);

System.out.println("hand count = " + p1.hand.size());
System.out.println("deck count = " + p1.deck.size());
System.out.println("coins = " + p1.coins);
System.out.println("numActions = " + p1.numActions);
System.out.println("numBuys = " + p1.numBuys);

//for later reference
Card curse = Card.getCard(cards, Card.CardName.Curse);

// play the game
p1.playKingdomCard();

System.out.println(" ***** Player-1 Status *****");
System.out.println("hand count = " + (p1.hand.size() + p1.playedCards.size()) +
        ", expected = " + p1t.hand.size());
System.out.println("deck count = " + p1.deck.size() +
        ", expected = " + (p1t.deck.size()));
System.out.println("numActions = " + p1.numActions +
        ", expected = " + (p1t.numActions - 1));
assertEquals(p1.hand.size() + p1.playedCards.size(), p1t.hand.size());
assertEquals(p1.deck.size(), p1t.deck.size());

System.out.println(" ***** Player-2 Status *****" );
System.out.println("hand count = " + p2.hand.size() +
        ", expected = " + p2t.hand.size());
System.out.println("deck count = " + p2.deck.size() +
        ", expected = " + p2t.deck.size());
System.out.println("discard = " + p2.discard.size() +
        ", expected = " + (p2t.discard.size() + 1));
System.out.println("top deck = " + p2.deck.peek());
```

```
        assertEquals(p2.hand.size(), p2t.hand.size());
        assertEquals(p2.deck.size(), p2t.deck.size());
        assertEquals(p2.deck.peek(), curse);

        System.out.println(" ***** Game State *****");
        System.out.println("Curse = " + state.gameBoard.get(curse) +
                ", expected = " + (testState.gameBoard.get(curse) - 1));
        assertEquals((int) state.gameBoard.get(curse),
                (int) testState.gameBoard.get(curse) - 1);
    }
}
```

As shown in the Sea Hag card test, the cards are first initialized to have a baseline that can be called upon. From there each player is initialized, and the gamestate itself is created. Once an accurate example of the gamestate is created and cloned, it can be modified and tested against a copy to ensure that the desired changes take effect (namely that the player draws a card, and every other player discards a card and topdecks a curse).

Other card tests follow the same implementation of initialization, playing, and checking the result. For cards with a choice, only one implementation is tested at a time (to be unit tested multiple times until the other random state is covered). One such test would be for the Minion card.

Listing 2. Minion Test

```
        if(p1.coins != p2t.coins) { //player gains 2 coins
            System.out.println("hand count = " + (p1.hand.size() + p1.playedCards.size()) +
                    ", expected = " + p1t.hand.size());
            System.out.println("deck count = " + p1.deck.size() +
                    ", expected = " + p1t.deck.size());
            System.out.println("coins = " + p1.coins +
                    ", expected = " + (p1t.coins + 2));
            System.out.println("numActions = " + p1.numActions +
                    ", expected = " + p1t.numActions);
            assertEquals(p1.hand.size() + p1.playedCards.size(), p1t.hand.size());
            assertEquals(p1.deck.size(), p1t.deck.size());
            assertEquals(p1.coins, p1t.coins + 2);
            assertEquals(p1.numActions, p1t.numActions);

            System.out.println(" ***** Player-2 Status *****" );
            System.out.println("hand count = " + p2.hand.size() +
                    ", expected = " + p2t.hand.size());
            System.out.println("deck count = " + p2.deck.size() +
                    ", expected = " + p2t.deck.size());
            System.out.println("discard = " + p2.discard.size() +
                    ", expected = " + p2t.discard.size());
            System.out.println("top deck = " + p2.deck.peek());
            assertEquals(p2.hand.size(), p2t.hand.size());
            assertEquals(p2.deck.size(), p2t.deck.size());
            assertEquals(p2.discard.size(), p2t.discard.size());
```

```
} else if (p1.deck.size() != p1t.deck.size()) { // player redraws (opponents, too)
    System.out.println("hand count = " + (p1.hand.size() + p1.playedCards.size()) +
            ", expected = " + 4);
    System.out.println("deck count = " + p1.deck.size() +
            ", expected = " + (p1t.deck.size() - 4));
    System.out.println("coins = " + p1.coins +
            ", expected = " + p1t.coins);
    System.out.println("numActions = " + p1.numActions +
            ", expected = " + p1t.numActions);
    assertEquals(p1.hand.size(), 4);
    assertEquals(p1.deck.size(), p1t.deck.size() - 4);
    assertEquals(p1.coins, p1t.coins);
    assertEquals(p1.discard.size(), p1t.hand.size() - 1);

    System.out.println(" ***** Player-2 Status *****" );
    System.out.println("hand count = " + p2.hand.size() +
            ", expected = " + 4);
    System.out.println("deck count = " + p2.deck.size() +
            ", expected = " + (p2t.deck.size() - 4));
    System.out.println("discard = " + p2.discard.size() +
            ", expected = " + (p2t.discard.size() + p2t.hand.size()));
    assertEquals(p2.hand.size(), 4);
    assertEquals(p2.deck.size(), p2t.deck.size() - 4);
    assertEquals(p2.discard.size(), p2t.discard.size() + p2t.hand.size());
} else {
    System.out.println("Internal error, check minion");
    assertEquals(1, -1);
}
```

Given more time, I would generate desired test states for each case instead of relying on randomly generated tests.

III. Bugs

There are five induced bugs in my program, each with a varying depth of complexity to find.

Players start with 2 buys at game beginning instead of 1.

Listing 3. Setup Bug

```
//induced bug: player starts with 2 coins
player.numActions = 1;
player.coins = 0;
player.numBuys = 2;
```

Game setup initialized 11 kingdom cards.

Listing 4. Setup Bug

```
//induced bug, grabs 11 kingdom cards instead of 10
```

```
while (selectedKindom <= Kingdom_Cards_Selected) {
    random = Randomness.random.nextInt(cards.size());
    Card tmp = cards.get(random);
    if(gameBoard.containsKey(tmp) || (tmp == null)) continue;
    gameBoard.put(tmp, 10);
    selectedKindom++;
}
```

Players do not reset their coins at the end of each turn.

Listing 5. Player Coin Reset Bug

```
final void endTurn() {
    System.out.println(player_username + " discards their hand and played cards");
    //move cards in hand to discard
    discard.addAll(hand);
    hand.clear();

    //move played cards to discard
    discard.addAll(playedCards);
    playedCards.clear();

    for(int i = 0; i < 5; i++)
        drawCard();

    numActions = 1;
    //induced bug, does not reset coins
    //coins = 0;
    numBuys = 1;
    System.out.println(" ---- " + this.player_username + " done");
}
```

Duchy gives 2 victory points instead of 3.

Listing 6. Duchy Bug

```
//induced bug: duchy gives 2 vp instead of 3
o = new Card(CardName.Duchy,Type.VICTORY,5,2,0);
```

Playing mine may not upgrade a treasure card.

Listing 7. Mine Test

```
//if there is a treasure that can be mined
//trash it and upgrade to a treasure 3 cost more
if(filter(player.hand, Type.TREASURE) != null) {
    int randTreasure = Randomness.random.nextInt(2);
    int minedTreasure = 0;
    Card treasure = null;
    if(randTreasure == 1) { //50% chance to copper > silver
        treasure = getCard(player.hand, CardName.Copper);
```

```
            if(treasure != null) {
                System.out.println("Player upgrades copper to silver");
                player.hand.remove(treasure);
                player.hand.add(getCard(state.cards, CardName.Silver));
                minedTreasure++;
            }
            treasure = getCard(player.hand, CardName.Silver);
            if((treasure != null) && (minedTreasure == 0)) {
                System.out.println("Player upgrades silver to gold");
                player.hand.remove(treasure);
                player.hand.add(getCard(state.cards, CardName.Gold));
            }
        } else {
            treasure = getCard(player.hand, CardName.Silver);
            if(treasure != null) {
                System.out.println("Player upgrades silver to gold");
                player.hand.remove(treasure);
                player.hand.add(getCard(state.cards, CardName.Gold));
                minedTreasure++;
            }
            //induced bug: player may play mine but not upgrade
            //their treasure (in the case they have copper/silver)
            /*
            treasure = getCard(player.hand, CardName.Copper);
            if((treasure != null) && (minedTreasure == 0)) {
                System.out.println("Player upgrades copper to silver");
                player.hand.remove(treasure);
                player.hand.add(getCard(state.cards, CardName.Silver));
            }
            */

        }
    }
    //and put it into your hand
    return;
```

IV. Results of Unit Testing

With the random input on some card implementations, the unit testing may varies a bit per implementation. On an average run the unit tests give the following results:

```
Results :

Failed tests:
  MineTest.test0:57 expected:<  Silver–TREASURE          Cost: 3          Score: 0
  Treasure Value: 2> but was:<null>
  valuesTest.duchy:46 expected:<3> but was:<2>
```

```
GameStateTest.test0:26 expected:<17> but was:<18>
GameStateTest.test1:43 expected:<2> but was:<1>
BuyTest.test0:59 expected:<1> but was:<2>
BuyTest.test1:95 expected:<0> but was:<5>

Tests run: 42, Failures: 6, Errors: 0, Skipped: 0
```

All 5 of the bugs presented earlier are caught in these tests. The first test catches the mine bug of not upgrading silver in some cases. The duchy card is caught with the wrong value. In the gamestate test, it catches both the spare kingdom card and the wrong number of buys. The next test also notices that the number of buys is wrong. Lastly, the final error shows that the player did not reset their coin value at the end of their turn.

When run in a loop of 20 testing calls, the results are close to the same.

V. Code Coverage

The code coverage report shows that most of the code is run through, with many branch cases hit.

CLASS	PlayDominion	GameState	Player	Card
INSTRUCTION_MISSED	88	7	1	33
INSTRUCTION_COVERED	0	551	678	1049
BRANCH_MISSED	2	4	1	10
BRANCH_COVERED	0	46	41	98
LINE_MISSED	18	3	1	8
LINE_COVERED	0	100	130	206
COMPLEXITY_MISSED	3	4	1	9
COMPLEXITY_COVERED	0	32	34	65
METHOD_MISSED	2	0	0	0
METHOD_COVERED	0	11	14	14

VI. Random Test Generator

The pseudocode for a random test generator of this implementation would follow a partially structured format, but use random testing as the main portion. The format is simple in its conception, but would be much harder to implement (as the test would have to have access to a pool of structured unit tests that it could access as necessary). The code is as follows:

- Initialize game
- Test initialization states (cards, players)
- Play game
  - Test player action phase (card test)
  - Update state tracking
  - Test player treasure phase

  - Update state tracking
  - Test player buy phase
  - Update state tracking
  - Test player end phase
  - Test game state
- Run final state tests