**CSSE376 Dominion Final Report**

**State of Features**

* The project supports a game with 2-4 players.
* Each game is played with 10 randomly selected kingdom cards.
* During each turn, the player is allowed to see his hand of cards.
* He is then allowed to select which card he wants to play. If he does not have any action cards, the game immediately proceeds to his buy phase.
* The number of actions, buys and money he has are modified based on which action card he played.
* He is allowed to select which card(s) he wants to purchase.
* The card(s) that he purchased are added to his discard pile.
* The game automatically transfers to the next player with a message box stating which player’s turn it is.
* When a witch is played, the game can tell which players have moats in their hands and which players must obtain a curse. A message is displayed for the players who got curses at the beginning of their turns.
* When a militia is played, the game can tell which players have moats in their hands and which players must discard down to 3 cards. If a player must discard, the game waits until his turn and then prompts him to do so.
* The game displays a count of how many of each type of card is available so that players can keep track of how soon the game will be over.
* At the end of the game, the game tallies points and declares a winner or states which players tied.

**Testing Strategies Employed**

All of the tests for this project were created in a white box manner, because the code was visible as the tests were developed.

The project was run primarily on test driven development. Most of the tests throughout the project were unit tests. The Visual Studio built-in tests were used to run the unit tests. The unit tests were also used as regression tests, because if a previously passing unit test suddenly failed, it was clear that the newly written code was not functioning properly.

The cards that required user input were difficult to unit test. Tests for these cards were written as scripted tests where the tester was given a strict set of instructions to follow to execute each test (ie…check 1 box, check 2 boxes, check the gold card).

Integration testing was frequently used as several components of the project relied heavily on one another. For instance, the implementation of the cards nearly always required access to one or more players. Furthermore, the project used mocking because players could not take turns in the unit tests.

As the project progressed, system tests became extremely important to test that the whole system was able to work cohesively together. Finally, once the project was ending, exploratory tests (in the form of repeatedly playing the game) were used to find minor bugs within the entire project.

**Testing Strategies Not Employed**

Black box testing was never used during this project because all tests were written by people with access to the code. Mutation testing and fuzz testing were also never used due to time constraints and because the project was able to be adequately tested without those strategies.

**Testing Examples**

***Unit Testing:*** This unit test verifies that treasure cards return no victory points.

[TestMethod]

public void testTreasureReturnsNoVP()

{

Card c = new Silver();

Assert.AreEqual(0, c.getVictoryPoints());

}

***Scripted Testing:*** This scripted test verifies that when a mine card is played and the player chooses to give up a silver, that player then receives a gold in his hand. The tester is instructed to select the silver card from the checkbox menu that appears when the test is run.

[TestMethod]

public void TestMineGiveUpSilverForGold()

{

Dictionary<Card, int> cards = new Dictionary<Card, int>();

cards.Add(new Gold(), 1);

GameBoard board = new GameBoard(cards);

Card c = new Mine();

Player p1 = new HumanPlayer(1);

board.AddPlayer(p1);

ArrayList newHand = new ArrayList();

newHand.Add(new Silver());

newHand.Add(new Silver());

newHand.Add(new Silver());

newHand.Add(c);

p1.setHand(newHand);

int cardsInHand = p1.getHand().Count;

int moneyInHand = p1.getTotalMoney();

p1.playCard(c);

Assert.IsTrue(p1.getHand().Contains(new Gold()));

Assert.AreEqual(moneyInHand + 1, p1.getTotalMoney());

Assert.AreEqual(cardsInHand - 1, p1.getHand().Count);

}

***Integration Testing:*** This integration test integrates the Player class and the Witch class to verify that when a witch card is played, a curse is not added to the discard pile of the player who played the witch.

[TestMethod]

public void TestWitchDoesNotAddCurseToMe()

{

Dictionary<Card, int> cards = new Dictionary<Card, int>();

cards.Add(new Curse(), 30);

GameBoard board = new GameBoard(cards);

Card c = new Witch();

Player p1 = new HumanPlayer(1);

Player p2 = new HumanPlayer(2);

Player p3 = new HumanPlayer(3);

board.AddPlayer(p1);

board.AddPlayer(p2);

board.AddPlayer(p3);

int cardsInDiscardp1 = p1.getDiscard().Count;

p1.addCardToHand(c);

p1.playCard(c);

Assert.AreEqual(cardsInDiscardp1 + 1, p1.getDiscard().Count);

Assert.IsFalse(p1.getDiscard().Contains(new Curse()));

}

***Mocking:*** This test verifies that the game will end correctly by creating a mock of a player that draws a province card every turn until the game ends.

[TestMethod]

public void IntegrationTestTieIsThrownOnTrueTieWithFivePlayers()

{

// MockRepository mocks = new MockRepository();

Dictionary<Card, int> cards = GetTestCards();

GameBoard board = new GameBoard(cards);

Player p1 = new SpecialPlayerMock(1);

Player p2 = new SpecialPlayerMock(2);

Player p3 = new SpecialPlayerMock(3);

Player p4 = new SpecialPlayerMock(4);

Player p5 = new SpecialPlayerMock(5);

board.AddPlayer(p1);

board.AddPlayer(p2);

board.AddPlayer(p3);

board.AddPlayer(p4);

board.AddPlayer(p5);

try

{

board.PlayGame();

}

catch (TieException e)

{

Assert.AreEqual(5, e.getArraySize());

return;

}

Assert.Fail("expected an exception");

}

public class SpecialPlayerMock : HumanPlayer

{

public int numbTimesCalled;

public SpecialPlayerMock() : base()

{

numbTimesCalled = 0;

}

public SpecialPlayerMock(int n) : base(n)

{

numbTimesCalled = 0;

}

public override void TakeTurn()

{

numbTimesCalled++;

GameBoard.getInstance().GetCards()[new Province()] -= 1;

getDiscard().Add(new Province());

}

}

***System Testing:*** System testing was used in the middle of the project to verify that the cards and graphics were communicating correctly. System testing was performed by running the game.

***Exploratory Testing:*** Exploratory testing was performed by playing through the game several times in an effort to find all bugs.



**Thoroughness of Testing**

Our scripted testing scopes from unit tests to integration test. We covered pieces of code on the atomic level with thorough unit testing. Our tests expand to the larger scope of how classes and functions interact with each other, as well. Beyond that, we did extensive levels of manual exploratory system testing on the game by just playing the game. Our extensive testing allowed us to achieve a test coverage of nearly 94% on our game.

Our thorough testing has uncovered many bugs, most of which have been fixed, however a few bugs still remain. For example, the button “end phase” is supposed to end the current phase of the players turn without the normal end conditions being met (for example, a player still has cards left to play he does not wish to play, so he skips to his buy phase). This button does not currently work.