We are using JUnit 4 as our testing framework. We achieved 100% code coverage of the code which we want to unit test – the parts of the code which are not covered are outlined later in this document. To make sure we were writing good tests, we followed the Right-BICEP method as long as it was relevant, and we made an effort to make our tests automatic, thorough, repeatable, independent and professional.

## Tests for the Card class

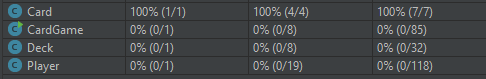


Figure : code coverage from running the tests in the CardTest class

Card is a simple class and figure 1 shows that the code coverage (from left to right) for its class, methods and then lines.

## Tests for the Deck class

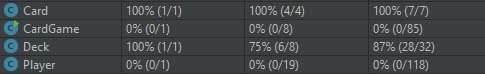


Figure : code coverage from running the tests in the DeckTest class

The only code not-covered are getter and setter methods, which we are confident function as expected and therefore do not need to be tested.

One notable test is the *writeToFile()* test, which ensures our file IO is working as expected. This method gives a deck an ArrayList of six cards, allows the deck to write its hand to a file, and then it reads in that file to ensure the method has correctly written the deck’s hand to a file.

For testing hasCards – we ensure we are covering the inverse case. I.e. we test in the cases where the deck does have cards and when it does not.

To test removeCard – we have two tests: one in the scenario where the deck has a card of the preferred denomination, and another where it does not.

This test, coupled with our player’s functionality of discarding non-preferred cards, ensures that the game does not stagnate.

## Tests for the Player class

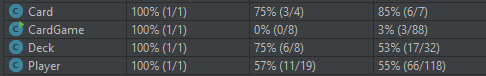


Figure : code coverage from running the tests in the PlayerTest class

The method coverage of 57% seems low, but it’s because there are 6 getter and setter methods which are not tested. Also, the line coverage of 55% seems low, but this is because a large portion of the lines are within the *run()* method, which cannot be tested by a unit test, as it is run in the context of a game running with multiple players interacting.

We include a test to make sure a *Player* can correctly recognize if they have a winning hand. This test asserts that the *CardGame* object has its winner set to the player which we gave a winning hand, and also that the CardGame object is no longer running when we have a winning player.

Note that we do not have to test every single file writing method, because many of them simple add strings to an ArrayList which then goes on to be passed to writeToFile(), so we only have to test this latter method.

In order to test the important *haveTurn()* method – we test its constituent methods, *drawCard* and *discardCard*. *discardCard* is tested in the 2 cases where it does or does not have a card of its preferred denomination. We also ensure that four main things happen in *haveTurn()* – *turnsHad* increases by 1, the player’s left deck loses a card, the player’s right deck gains this card, and the player keeps the same number of cards.

*testAddCard* ensures that no more than 4 cards can be added to a player’s hand. This might happen if the pack file somehow slips through our test to ensure we have no more than 8n cards in the pack.

## Tests for the CardGame class

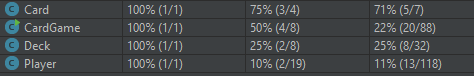


Figure : code coverage from running the tests in the CardGame class

The CardGame line coverage is only 22% as shown in figure 4, but this is because a lot of its lines are to do file IO, such as *getPack()*, which we have already effectively tested in other tests, and *playGame()*, which cannot be tested by a unit test, for the same reasons as explained above for the *run()* method in player.

The setup function for this test class (annotated with @Before) sets up a hardcoded pack, rather than reading one in using file IO, and then sets up decks and players as usual.

We test our setup methods as well, like getDecks() and getPlayers(), to ensure our CardGame can perform the fundamental action of setting up decks and players.

## Parts of the code which are not tested

### Some IO methods

We have a method called *testWriteToFile()* in the *PlayerTest* class, which ensures that we can correctly write to a file and read it. Therefore we have avoided test duplication by not testing some of the related file writing methods such as *initialWriteToFile()* and *finalWriteToFile*(), because these functions simply add strings to a *fileOutput* variable, which is then passed to *writeToFile()*.

### Getters and setters

These functions are not tested as we are confident they function as expected.

### Functions which require the game to be running

Some functions do not make sense to test using unit tests, such as *run()* in the Player class, and *playGame()* in the CardGame class. Further work could be to do testing of these functions during the phases of integration testing and system testing.