ECM2414 CA Report

By Students *730003140 & 7300049916*

Production Code Design Choices

Software Development Lifecycle

When developing this project, we recognised the importance of utilising SDLC methodologies to enhance our development workflow. Due to the nature of this project as a university coursework the emphasis of this project lay in '*Requirements Analysis & Design*', '*Implementation & Testing'* but not the '*Evolution*' stage. We used a modified version of the *Waterfall* approach to complete this project. We found that distinct phases and signing off that each partner approved each stage to be valuable with pair-programming. We first read the specification and once agreed that we both fully understood it we began our design phase. We discussed the design of our project and created a UML diagram. After this phase concluded, we implemented our design; In addition to rotating the *Driver* and *Observer*, each session we rotated whether we were writing production code or *unit tests* for the production code we wrote in the previous session. Implementing *Unit Tests* as we went along allowed us to combat the common Pitfall of the *Waterfall* Method that errors can go unnoticed.

We additionally utilised Git & GitHub to develop our solution, this allowed us to work well on multiple machines and provided an easy method to verify our partners contributions when pair programming.

Class analysis

Card

The *Card* class is simplistic, containing a constructor *Card(int denomination)*, getter *getDenomination()* for the value and a *toString()* method which mainly aids with debugging. According to the specification there is no need for the cards to have much functionality as they are immutable objects that are moved around by the players. The constructor however does have error checking to make sure the denomination is ≥ 0 as required.

Pack & IPack

We decided to create a *Pack* class to abstract the complexity of loading files from the *CardGame* class. When unit testing, we didn't want superfluous test packs that we would have to package, we therefore created the *IPack* interface so that we could create a mock class that implements *IPack* in addition to the production *Pack* class that also implements *IPack*. Inside the *Pack* class the constructor reads the text file and creates a private array of cards that can then be accessed with a public getter. The constructor additionally error checks that the correct number of cards are present, and they are all valid (non-negative integers).

Deck

A deck is a collection of cards that a card can be added to the bottom of or taken from the top of. We first implemented this basic functionality in our *Deck* class so that it has a private field to hold the cards and 2 public functions: *void addCard(Card)* that adds a card to the bottom of the list, and *removeCard()* that removes the top card and returns it. The *Deck* class also features a public method to log its state to a file, this will get called at the end of the game as required. We decided that this functionality should be in the *Deck* class rather than the *CardGame* class to better conform to OOP. We finally added two Boolean functions to the *Deck* class: *canDrawCard()* and *canGiveCard()*, these have the purpose of telling the *Player* class whether they can draw a card, without the *player* needing to access private attributes of the *Deck* class, this was developed to better conform to OOP.

CardGame

The *CardGame* class represents a game as designed by the specification and also serves as an entry point for the system via the *public static void main()* function. This *main* function accepts and validates inputs of player count and pack path before creating and instance of the *CardGame* class and invoking its *playGame()* method. The Constructor for the class creates an array of constructed *Player* and *Deck* objects and then runs the *dealCards()* method that allocates the cards in the imported Pack between objects in a round robin style. The *playGame()* method simply starts the Player threads and waits for a winner to be found, before interrupting the remaining threads.

Player

This class represents a threaded player in the Game and undertakes all actions after the thread is started. It has a synchronized method *checkWin()* that checks if all cards are the same and if so, notifies the parent *CardGame* class, the synchronisation helps with multiple players not declaring a win. The *run()* method is called when starting the thread, it writes to the log, then continues to call *takeTurn()* until there is a winner found then writes its final exit messages to the log. The *takeTurn()* method consists of discarding a card according to preference rules, drawing a one then writing to a log.

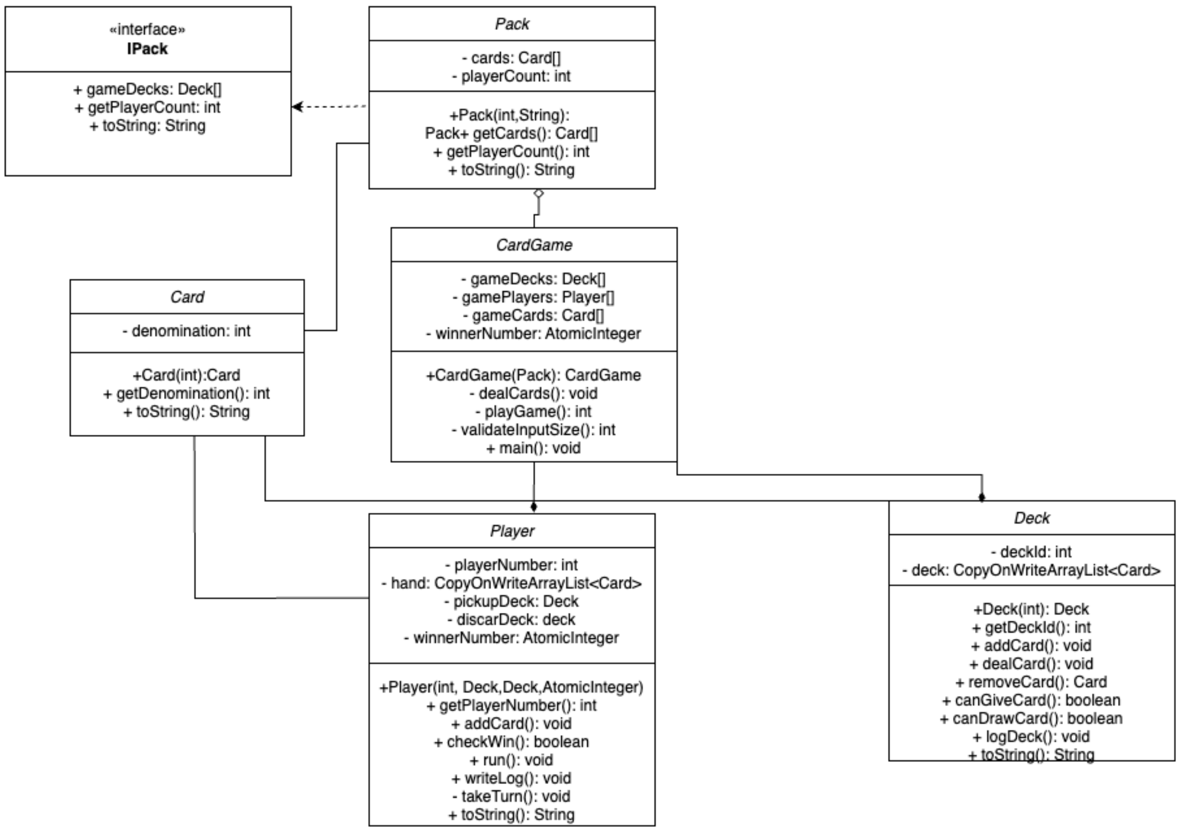
Other Considerations

Multithreading & Thread Safety

When creating the *Player* class we implemented *Runnable* instead of extending *Thread*, this meant our Player class was decoupled from the threading details and better followed OOP principles, it also gave us better flexibility. We also utilised CopyOnWriteArrayList instead of Array List when Decks and players held cards to improve thread safety. We also utilised atomic integers in the *CardGame* class to keep track of the winning player, we used this instead of regular integers to discourage multiple players declaring a win. Finally, we used synchronisation when players executed *takeTurn()* though we were careful to keep the synchronised areas to a minimum to increase multithreading speed. After a win has been declared the threads will all be interrupted by the CardGame class to finish execution.

Performance Issues

Having finished the project there are no known performance issues in our code, on occasion multiple threads will declare a win though this is said by the specification to be acceptable.



Testing Analysis

The testing framework used for this project is Junit 4.13.2.

Design choices

Due to us deciding on a slightly modified version of the waterfall model, we adopted a more traditional approach to testing the project, creating one test class per production class. This test class would then test each method, public and private (using reflection) within that class. This approach was chosen so that we could test fully the behaviour of each method, test that error checking happened correctly and that all types of inputs: valid, invalid, extremities and edge cases produced the correct results. This led us to achieve maximum code coverage. When checking coverage, we decided to grade on lines covered rather than classes, or methods covered; By checking lines this gave us confidence that we had tested all potential conditional routes within our methods.

When creating our test suite, we decided to utilise Junit 4.13.2 to test our system, we thought about other options such as using Junit 5.xx or just using try catch clauses though we decided against these options. Try Catch clauses was too primitive and we wanted a more modular system for this project, so we rejected that idea. We considered using Junit 5.xx but decided that for compatibility with older versions of Java reasons that we would use Junit 4.xx instead.

The general layout of all test classes consisted of using a *SetUp()* function with the @Before annotation to create a mock object for the tested class with preset values and then running the individual tests, by utilising this approach this enabled us to reduce repeated code and simplify the process to make it more readable. When dealing with classes that involved saved text data (*Pack, Deck, Player)* We used the *SetUp()* function again to make sure that there were no logs present prior to testing and used a *TearDown()* function with the @After annotation to erase any testing card log files afterwards as well to ensure the filesystem is unaffected by the tests running.

We also used a test suite to group all our test classes together. This allowed us to run all our tests at once and made code coverage features easier to use in our IDE of choice IntelliJ IDEA.

Helper functions

In the end we had **31/31 passing tests**, and a **code coverage of 89% of lines**.

Description of tests

Testing the *Card* Class

testGetDenomination()

This Test asserts that the *Card.getDenomination()* function returns the same integer specified for the mock card in the *SetUp()* function.

testToString()

This test asserts that the *Card.toString()* function returns the expected value. In the case of the mock *card(5)* the toString() should return "Card of Value: 5". Other classes have a toString() function though due to the simple and repetitive nature of the function I will not describe it again.

testNegativeDenomination()

This test asserts that an *IllegalArgumentException* is thrown when you try to create a card with a denomination < 0. This works using a try catch block, where if it doesn't catch that specific exception it uses the Junit *fail()* method.

Testing the *Pack* Class

When testing the Pack class there were some issues, when feeding it example valid or invalid packs we needed to have these saved – yet we didn't want to package unnecessary .txt files into our JAR package or with our source code solution. To fix this we created a function that creates a temporary test pack for the test from a Card[] parameter and then deletes the file again after testing in the *TearDown()* function. This way we can store the test data in the code not as a separate .txt file yet still test the Pack file loading capability.

testInvalidFileName()

This test checks that an *IOException* is thrown when the Pack constructor path parameter doesn't exist. This works using a try catch block, where if it doesn't catch an *IOException* after calling the invalid constructor it uses the Junit fail() method.

testConstructor ()

This test checks that there are no errors when running the *Pack* constructor with a valid hand.

testGetCards()

This test firstly creates a valid pack then asserts that the *Pack. getCards()* function actually returns the correct cards it is supposed to.

testGetPlayerCount()

This test checks that having read in a valid pack file it can return the correct number of players in the game. For example a pack of 16 cards will have 2 players.

testInvalidPacks()

This function is used to assert that errors are thrown when loading in faulty pack files. I will explain the process first then each of the cases that we checked. For each case I created a testing pack file using the function described earlier to create temporary files, we then used a try catch block to attempt to create a pack object using that temporary file path and then used the *fail()* method to assert that it did throw an error. The cases we tested were: A pack with a wrong number of cards (not of length 8n), a pack with a non-integer card (0.5, or "a") and a pack with a negative integer card.

Testing the *Deck* Class

testAddCard()

This test asserts that when you use the *Deck.addCard(Card)* method that a card is correctly added to the bottom of the deck's private deck field. To test this, we had to use java reflection to access the private fields. We firstly started with an empty test deck, then added a card to it. We then asserted that the deck's private card's list was now of size 1 and that it's card was the same denomination as the card we added.

testRemoveCard()

This test asserts the *Deck.removeCard()* method. We firstly created a test deck and added two cards to it. We then removed a card and asserted that it was the first added card (as cards should be added to bottom and taken from top). We then asserted that deck now contained only one card in it, this again involved Java reflection. We then removed the one remaining card to test as an edge case that this worked. Finally the test attempts to remove a card from the now empty deck, we assert that the result is null since it couldn't remove a card.

testGetDeckId()

This test simply asserts that the *Deck.getId()* function returns the same ID that the mock test deck was initialised with. This is a simple one line test so I won't detail it again for other classes that also have a test to check their getID methods such as *Player.*

testCanDrawCard() & testCanGiveCard()

These test checks that the *canGiveCard()* and *canDrawCard()* methods correctly work. We tested these with valid, invalid and boundary data. For example we tested the extreme case of testing whether we could draw a card from a deck with 0 cards, or the edge case of whether we could give a card to a deck of 4 cards.

testDeckLogNormal()

This test asserted that running the *Deck.logDeck()* method correctly created a file at the path "gameData/deck1.txt" with the deck's contents. This worked by creating a test deck with the cards 1,2 and 3 in then invoking the *logDeck()* method then asserting the "gameData/deck1.txt" file was correct.

Testing the *CardGame* class

testValidateInputSize()

This test asserts that the *validateInputSize()* function works correctly we gave it valid inputs such as 1, then invalid inputs such as -545, "abc" or the edge invalid input 0. We asserted that invalid inputs return -1 and correct inputs assert the correct result as an integer.

testConstructor()

This test asserts that the constructor for *CardGame* creates the correct number of instantiated players and decks. We needed to use reflection to access the private *gamePlayer* and *gameDeck* fields. This additionally required the usage of a Mock class for the pack so we could easily feed in demo packs.

testDealCards()

This test checks that the *dealCards* function correctly deals the cards in a round robin fashion as specified. It uses reflection to access the cards in each player and deck and asserts each has the correct arrangement according to the input pack *{1, 2, 1, 2, 1, 2, 1, 2, 3, 4, 3, 4, 3, 4, 3, 4}* we gave it.

testPlayGameInstantWin() & testPlayGame1RoundWin()

These tests test the *playGame()* method of the *CardGame* class. We again used the mock pack to create a game with a pack that could be solved in either 0 or 1 round then asserted that the actual winner was the expected winner according to each pack. We used code coverage tools to check that each test ran the correct number of rounds, either instant win or after 1 round.

Testing the *Player* Class

testCheckWin()

This test asserts that the *Player.CheckWin()* function works correctly. We tested many invalid / edge cases such as when the player has all identical cards - but only 3, 4 Cards that are not the same, or more than 4 cards that are the same and finally the valid case that all 4 cards are the same. We asserted each time that the correct True or False value was returned by the function.

testWriteLog()

The *Player.writeLog()* method only actually writes to file the content of the private player *String log* field to file. To test this we had a helper function that used reflection to set the contents of this private field and a helper function to read the file created at *"gameData/player101.txt"*. We then asserted that different types of data were correctly logged, ie empty strings, one line of text and multi-line text.

testTakeTurnCanGo() & testTakeTurnCanGoGiveRandom() & testTakeTurnCantGo()

This test checks the *Player.takeTurn() method*. We tested 3 cases. When the player can't go (nothing to pick up) we checked that there was no change to the player hand or deck after running *takeTurn()*. When the player can go and it should give a non-preferred card we asserted that that was the card it gave, and when it needed to give a random card we tested that the *takeTurn()* method did give a card away. These 3 different cases allowed us to achieve 100% line coverage and thus conditional route coverage for the *takeTurn()* method.

Development Log

Session Date	Duration	730003140	730003140
24th October	2h	Design	Design
27th October	1h30m	Design	Design
2nd November	1h	Driver	Observer
6th November	1h30m	Observer	Driver
15th November	1h	Driver	Observer
20th November	1h30m	Observer	Driver
1st December	2h30m	Driver	Observer
4th December	2h30m	Observer	Driver
5th December	2h30m	Driver	Observer