

Test Design

In order to develop a robust program, we followed thorough *test-driven development* practises throughout our software development lifecycle. By writing tests before implementing the actual production code, we were able to minimise bugs, increase reliability, and eliminate any testing bias. Moreover, writing tests gave a clearer idea of how exactly we'd go about implementing the production code, greatly simplifying our development process and maximising our efficiency as a pair. This principle of *test-driven development* was an essential component of our system design, as it established program stability and effective case handling.

The first choice we needed to make with respect to testing design was which version of the *JUnit* framework to use. Whilst both allow us to write unit tests, we found the differences between the 3.x and 4.x distributions were minimal (with respect to the task at hand), the only key changes being syntax-related. For this reason, we chose to use the latest of the two frameworks: *JUnit 4.x*.

To measure the code coverage of our unit tests, we chose to use the *IntelliJ IDEA code coverage runner*. Here we had other choices such as *JaCoCo* or *EMMA*, but found that IntelliJ provided the simplest user interface and was the most convenient (built-in to our IDE of choice).

Before writing any tests, we thought carefully about the high-level design of our system (see *Figure 1* in *Code Design*). This ensured that we knew exactly how the classes were supposed to act, both in terms of structure and behaviour/relations with other classes. Subsequently, writing appropriate tests for each of the classes came with relative ease.

We set out to cover as much code as possible with our tests, with the intention of writing production code that was as stable as possible, and able to pass all test-cases. Although we aimed for a high method coverage and were able to write unit tests for the large majority, we had no hard target for code coverage. This is because we knew some aspects of the simulation would be very difficult to test, as they exhibit unpredictable behaviour and rely on console input.

For example, testing I/O operations such as console input or file output was particularly difficult. Testing thread behaviour also proved difficult, as there are lots of possible paths a thread can take through the system, depending on game conditions (such as pack contents). We also knew we wouldn't be able to test the *main* method of the *CardGame* class.

Despite not being able to test the aforementioned methods, we still attempted to maximise the code coverage for the sake of developing a robust system and testing as much as possible. To do this, at the design stage we made the decision to encapsulate the *CardGame* class, by breaking the simulation down into a series of method calls, such that minimal computation is carried out in the *main* method. This way we could thoroughly test each constituent part of the simulation. For example, in order to better test I/O, we isolated the user input into two smaller methods, *readPlayerNumber(String input)* and *readFilePath(String input)*, and called these from the *main* method (rather than implementing them both in the method body of *main*).















When writing unit tests, the idea is to test how methods (and constructors) react when given different inputs. So to make our tests most effective, we ensured that all edge cases were tested. For example, in *PlayerTest* we have a *testEdgeCasePlayer()* test that instantiates a *Player* object with an *index* attribute of 1, as this is the minimum *index* a player is allowed to be assigned according to the specification (positive integer). We also wrote tests to check our custom exceptions were being thrown where appropriate, examples being the *testNegativePlayer()*, *testZeroPlayer()*, *testNegativeCard()*, *testZeroCard()*,

`testNegativeCardDeck()`, and `testZeroCardDeck()` tests. These tests check to see if the corresponding custom exception is thrown when invalid input is passed through the constructors of our classes, ensuring that they work as intended.

Beyond the design stage, we continually ran unit tests throughout the implementation and development of production code. This helped maintain a robust system that could endure a range of different test-cases. For example, if we made a slight change to the implementation, we would check the corresponding test method and ensure that it still covered all edge and corner cases. When developing the production code itself, methods were tested further in the form of debugging: utilising a trial-and-error approach to better understand the behaviour of methods. Whilst this isn't anywhere near as robust as writing an entire test suite, it still contributed to securing code stability and played a key role in our approach to *test-driven development*.

Thanks to our rigorous and diligent use of *test-driven development*, our tests were able to cover a large majority of our production code, as shown in *Figure 2*.

Figure 2:

Element	Class, %	Method, %	Line, %
 Card	100% (1/1)	100% (1/1)	100% (5/5)
 CardDeck	100% (1/1)	100% (6/6)	100% (12/12)
 CardDeckTest	100% (1/1)	100% (8/8)	72% (52/72)
 CardGame	100% (1/1)	62% (5/8)	79% (119/149)
 CardGameInterface			
 CardGameTest	100% (1/1)	100% (21/21)	99% (251/253)
 CardTest	100% (1/1)	100% (3/3)	68% (13/19)
 InvalidCardDeckException	100% (1/1)	100% (1/1)	100% (2/2)
 InvalidCardException	100% (1/1)	100% (1/1)	100% (2/2)
 InvalidPlayerException	100% (1/1)	100% (1/1)	100% (2/2)
 Player	100% (1/1)	100% (13/13)	77% (99/128)
 PlayerTest	100% (1/1)	100% (15/15)	72% (248/342)
 TestRunner	0% (0/1)	0% (0/1)	0% (0/7)
 TestSuite	0% (0/1)	100% (0/0)	100% (0/0)

As part of our final performance assessment, we found the following known bugs:

- The specification states '*there should only be one player declaring it has won for any single game*'. However, when testing our program on multiple different computers we found one case where our program would declare that two players had won, printing both to standard output (terminal). Thankfully this bug is very rare, as we only encountered it once after running the program over 50 times on the same machine. The actual gameplay and output files are still accurate and behave as they should (only one output file will declare that it has won). This is likely due to unexpected thread behaviour, and differences in OS/CPU architecture. On reflection, next time we will spend more time in the design stage ensuring that we account for these differences, to minimise the dependencies of our multi-threaded application.