ECM2419: Software Development Coursework

730022096 & 730002704

1 The V Model

We decided the best way to approach this project was by using the V-model as our software engineering lifecycle. This is a model where you first do the planning, preparation and documentation before you start any code, this is one of the key characteristics that distinguishes it from other software development models.

We found many benefits in using this model including:

1. It is highly structured, and phases are completed one at a time.
2. It works well for pair programming as all requirements are understood between everyone working on the code.
3. It is simple to understand.
4. It is easy to manage.

2 UML Diagram

The UML diagram of our finished production code is shown in Figure 1.

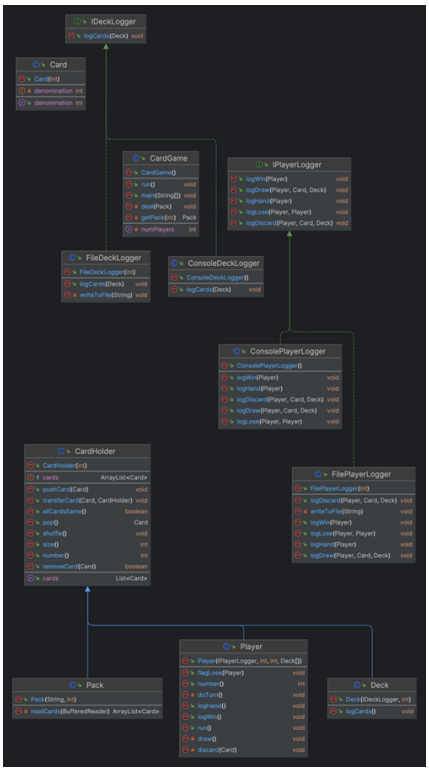

Figure 1: UML Diagram.

In this UML diagram you can see that we used a structural design pattern and more specifically this would be the Facade design pattern.

The facade design pattern provides a simplified interface to a complex subsystem. In our case, the high-level classes like `IPlayerLogger`, `IDeckLogger`, and `CardGame` act as the facade providing an interface to the underlying subsystems and components.

3 Key Design Choices

3.1 Facade Design Pattern

We decided to use a pattern like this as it provided a
simplified and consistent interface to the more complex game logic and all the other complex number/data management components going on as well.

3.2 Class Hierarchy

In our design we used high-level parent classes like `IPlayerLogger`, `IDeckLogger`, and `CardHolder` that encapsulate the functionality and are extended by more specialized classes like `ConsolePlayerLogger`, `FilePlayerLogger`, and `CardHolder`.

We did this as it promotes the separations of concerns and makes the code easier to understand which then makes it easier to maintain and whilst we were still editing the code it makes it easier to extend any code where we need to.

3.3 Separation of Concerns

Another key point of our design is Separation of Concerns. The design separates different responsibilities into distinct classes: e.g. Card: Data structure, Deck: Card collection management etc. The benefits of doing this include:

1. Easy to maintain and modify.
2. Clear responsibilities for each component.
3. Reduced coupling between components.
4. Better for paired programming.
   (a) Makes it easier to work on different parts of the code at the same time.
   (b) Makes it easier to understand the code.

3.4 Logging and Debugging

Using the logger classes `IPlayerLogger`, `IDeckLogger`, and `ConsolePlayerLogger` gives us more of a focus on logging and debugging functionality.

We did this as it is useful for tracking and analysing game events, player actions, and potential issues during our development of the project.

4 Testing

4.1 CardHolderTest

4.1.1 Test Coverage

- Testing of `CardHolder` methods
- Tests both successful and edge cases

We wanted a range of tests that could test all methods in the `CardHolder` class effectively and efficiently and follow all the attributes of the Right Bicep test planning model. With that in mind, we designed the tests:

- `pushCard()`
  - Tests adding a card to a `CardHolder`
- `popCard()`
  - Tests removing the top card from a `CardHolder`
- `popCard_empty()`
  - Tests attempting to pop a card from an empty `CardHolder`
- `removeCard()`
  - Tests removing a specific card from a `CardHolder`
- `removeCard_nonExistent()`
  - Tests removing a card not in the `CardHolder`
- `transferCard()`
  - Tests transferring a card between two Card Holders
- `transferCard_nonExistent()`
  - Tests transferring a non-existent card
- `allCardsSame()`
  - Tests checking if all cards in a `CardHolder` have the same denomination
- `allCardsSame_empty()`
  - Tests `allCardsSame()` method on an empty `CardHolder`
- `getCards()`
  - Tests retrieving the list of cards from a `CardHolder`

4.1.2 Error Handling

- We used `@Test(expected = IllegalStateException.class)` for error scenarios
- Verifies correct behaviour when attempting operations on empty collections

4.1.3 State Validation

- Checks list size after operations
- Verifies correct card placement and transfer
- Ensures state consistency after each operation

4.1.4 Setup Strategy

- We used `@Before` annotation to create new CardHolder instances for each test
- Creates two separate CardHolder instances to test interactions

4.2 CardTest

Single test method for `getDenomination()` with basic verification of Card object's core functionality.

4.3 PackTest

4.3.1 Test Coverage

- `testPackCreation_ValidFile()`
  - Tests creating a Pack with a valid file containing 16 cards
- `testPackCreation_InvalidFile_NotEnoughLines()`
  - Tests creating a Pack with a file that has insufficient lines
- `testPackCreation_InvalidFile_NonIntegerValue()`
  - Tests creating a Pack with a file containing a non-integer value
- `testPop_ValidCase()`
  - Tests the `pop()` method of the Pack - Verifies the first card has the correct denomination (1)
- `testPackCreation_EmptyFile()`
  - Tests creating a Pack with an empty file

4.3.2 File-Based Testing

- We used temporary file creation for testing Pack initialization
- Tests multiple file input scenarios
- Dynamically generates test files with different contents

4.3.3 Error Scenario Coverage

- Tests pack creation with:
  - Valid file
  - Insufficient lines
  - Non-integer values
  - Empty file

4.3.4 File Handling

- Implements a helper method `createTemporaryPackFile()` to manage test file creation
- Uses try-with-resources for safe file writing
- We used `java.nio.file` for temporary file management

4.4 Design Choices

- Follows JUnit testing conventions
- Uses `assertXXX` methods for validation
- Tests both successful and failure paths
  - Leads to more accurate testing results
- We focused on individual method behaviour
  - Means we can get more accurate test results and accurately test that each method is working how it is designed
- Ensures predictable object state after operations
- Uses `@Before` annotation to create new instances for each test
  - Means we can accurately test each method more accurately to ensure the functionality is working properly

5 Work Log

Date	**Hours**	**730022096**	**730002704**
24/8/24	3	Documentation	Documentation
29/10/24	3	Driver	Observer
5/11/24	3	Observer	Driver
7/11/24	2	Driver	Observer
14/11/24	4	Observer	Driver
21/11/24	4	Driver	Observer
28/11/24	3	Observer	Driver
1/12/24	4	Driver	Observer
5/12/24	4	Observer	Driver

Table 1: Work Log.