ECM2419: Software Development Coursework

730022096 & 730002704

Contents

Declaration

No GenAI was used in the production of this report or the associated code.

1 Design Choices

1.1 The V-Model

We chose to use the V-Model of software development as our software engineering lifecycle. This model is characterised by the planning, preparation, and documentation phases being completed before any code is written. This model has five phases: feasibility, design, implementation, testing, and maintenance. Because of the nature of our project, we can ignore the feasibility and maintenance phases.

Benefits of using the V-Model are that:

1. It provides a structured approach as phases are completed one at a time.
2. It is well-suited for pair programming because all requirements are understood by both programmers.
3. It is simple to understand.
4. It is easy to manage.

Some drawbacks of the V-Model are that:

1. It is not well-suited for large projects.
2. It is not well-suited for projects with changing requirements.
3. It is not well-suited for projects with tight deadlines.

Because our project is small and has a fixed set of requirements, the V-Model was well-suited.

The design phase of the V-Model is where we planned the structure of our code and the classes we would need using a UML diagram.

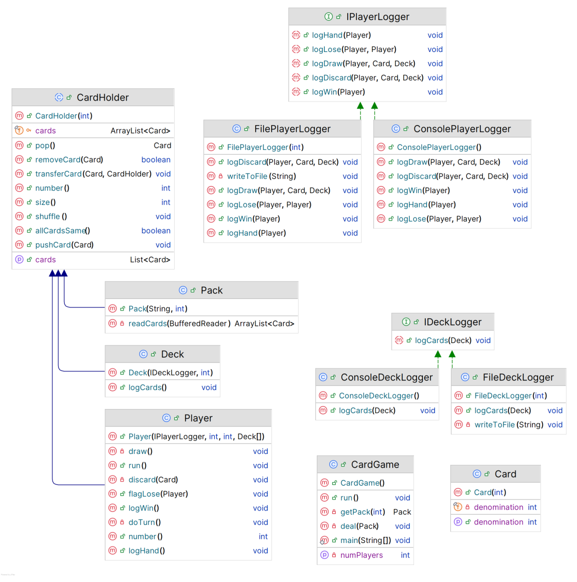

Figure 1: UML Diagram.

1.2 UML Diagram

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

1.3 Key Design Desicions

1.3.1 Facade Design Pattern

In this UML diagram you can see that we used the facade design pattern to provide a simplified interface to a complex subsystem. The high-level classes like `IPlayerLogger`, `IDeckLogger`, and `CardHolder` act as the facade providing an interface to the underlying subsystems and components.

1.3.2 The `CardHolder`

The abstract class `CardHolder` is a key part of our design, abstracting common functionality between `Pack`, `Deck`, and `Player`. It implements the following key methods:

- `void pushCard(Card card)`: Adds a card to the `CardHolder`.
- `void transferCard(Card card, CardHolder target)`: Transfers a card from this `CardHolder` to another `CardHolder`.
- `Card pop()`: Removes and returns the top card from the `CardHolder`.

Using an abstract class for this functionality makes it easier for our code to be thread safe, as we can ensure that all operations on a `CardHolder` are atomic. It also makes testing easier, as we can test the `CardHolder` class in isolation from the other classes.

1.3.3 Logging Classes

Our design uses the interfaces `IPlayerLogger` and `IDeckLogger` to provide a common interface for logging player and deck events. This allows us to easily switch between different logging implementations, such as logging to the console or to a file.

This design decision was made to make our code more modular and easier to test. By using interfaces, we can easily mock the logging classes in our tests, allowing us to test the rest of the code in isolation.

It also improved the developer experience, as it made it easier to debug the code with logging being done to the console.

1.3.4 Separation of Concerns

Another key point of our design is the separation of concerns. The design separates different responsibilities into distinct classes: `Pack` is responsible for managing the reading of the pack file, `Deck` is responsible for managing a deck of cards, and `Player` is responsible for managing a player's hand.

The benefits of doing this include:

1. Easier to test: Each class can be tested in isolation, making it easier to identify and fix bugs.
2. Easier to maintain: Changes to one class are less likely to affect other classes, making it easier to maintain the codebase.
3. Easier to understand: Each class has a single responsibility, making it easier to understand the codebase as a whole.

2 Testing

2.1 Design Choices

Our tests use the JUnit 4 testing framework. We chose JUnit because it is widely used and is well supported in IDEs like IntelliJ IDEA. We used unit tests to test key functionality of our code, such as adding and removing cards from a `CardHolder`.

Manual system tests were used to test the system as a whole. We tested the system by running the game and checking that the output was as expected. We also tested the system by running the game with different inputs to check that the game behaved correctly in different scenarios.

Our tests used the BICEP testing model. This model is based on testing:

- Boundary conditions.
- Inverse relationships.
- Cross-check results.
- Error conditions.
- Performance characteristics.

We followed the JUnit conventions for test method names, using the format `test<Method>_<Scenario>`. This made it easy to identify which method was being tested and what scenario was being tested.

`assert` methods (`assertEquals`, `assertTrue`, etc.) were used to validate the state of the objects being tested.

The `@Before` annotation was used to create new instances of the classes being tested for each test. This ensured that each test was run in isolation and that the state of the objects being tested was consistent between tests.

2.2 CardHolderTest

2.2.1 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`

2.2.2 Error Handling

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

2.2.3 State Validation

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

2.2.4 Setup Strategy

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

2.3 CardTest

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

2.4 PackTest

2.4.1 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

2.4.2 File-Based Testing

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

2.4.3 Error Scenario Coverage

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

2.4.4 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

3 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.