D7032E

Software Engineering

Home Exam – Re Exam 1

Submitted By: Rhyan Wong

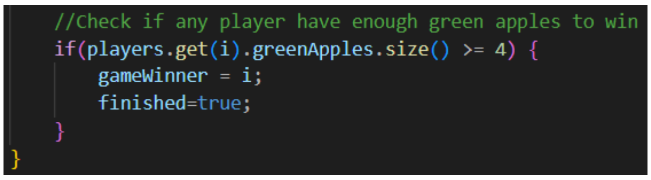
Email: rhywon-4@student.ltu.se

Teacher's Comment

Table of Contents

Q1 Unit Testing

Q1.1 Requirement Unfulfilled



```
    //Check if any player have enough green apples to win
    if(players.get(i).greenApples.size() >= 4) {
        gameWinner = i;
        finished=true;
    }
}
```

Fig 1. Win condition in original code

Based on the original code, requirement 15 is not fulfilled. The code assumes four green apples are required to win for all cases which fails to consider the player count for other scenarios (game for five players and more). This is the result of the winning threshold logic being hardcoded to only four green apples.

Q1.2 Requirements Testable By JUnit Without Modification To Code

None found.

Q1.3 Modification Needed For JUnit To Test Requirements

Requirement 1 to 14 at a glance seemed fulfilled by the code, however they cannot be tested directly without modification to the existing code. This is due to several issues:

i) Tight Coupling

Core behaviours like shuffling, file reading, and player interactions are embedded directly into the code rather than being modularized. This makes components dependent on each other, limiting flexibility and testability.

ii) Lack of Dependency Injection

Classes rely on concrete implementations (ThreadLocalRandom, file operations etc.) rather than being provided dependencies through constructors or setters. This rigid approach prevents replacing dependencies with mock implementations during testing.

iii) No Clear Interfaces

Key behaviours such as shuffling, dealing cards, judging, and scoring are implemented directly within classes without defining interfaces. This absence of abstractions hinders the ability to substitute different implementations behaviours during testing.

Q2 Software Architecture Design and Refactoring

Q2.1 Identifying Shortcomings In The Original Design

The original code primarily violates SRP and DIP of SOLID Principles. The *Player* class is responsible for multiple tasks, such as handling game logic, network communication, and managing player state. Additionally, methods like play(), judge(), and addCard() perform multiple actions (playing cards, handling user input/output, interacting with the server etc), which makes them difficult to maintain and test. The code also lacks clear abstraction for the server-client communication, tightly coupling the game logic with network operations. The Dependency Inversion Principle is not followed because high-level game logic depends on low-level socket and I/O operations.

Q2.2 Justification For Design Choices In The Refactored Code

The source folder ***card*** contains refactored code that encapsulates all the card-related logic. The *Card* abstract class defines the general concept and shared behaviours of all cards, serving as the implementation basis for *RedApple* and *GreenApple*. The *CardFunctionalities* interface standardises operations such as *displayCard* and *handle-Card*, which promotes flexibility. The *CardFactory* centralises card creation, this supports extensibility for new types without modifying existing logic, while *CardManager* encapsulates card operations like loading and managing collections. These changes to the original code ensure single responsibility by isolating card management, open/closed principle by allowing integration of new card types, Liskov substitution by enabling interchangeable use of card subclass. Interface segregation by tailoring card-specific operation and dependency inversion by decoupling high-level modules from low level details. Through refactoring the original code, Booch metrics has also been adhered to. In the original code, high coupling existed as multiple classes, like *Player*, directly interacts with card-related logic. However, with the introduction of abstractions (*Card, ICardFunctionalities, ICardFactory*) and encapsulating card-specific operations in *CardManager*. Classes are now low coupled, as they rely on well-defined interfaces than concrete implementations, enabling modularity and scalability. The refactor code also improves cohesion by encapsulating distinct responsibilities within specific classes. The *GameLogic* class focuses on managing game mechanics (player turn, determining the winner etc) while the aspect of scoring is delegated to *ScoringManager* and state management to *GameContext*. Primitiveness is also fulfilled, in the case of ***card***, its card-related operations are broken down into smaller and more focused components. For example, *ICardFactory* defines a simple, single-purpose interface for card creation.

Q2.3 Quality Attributes Satisfied By The Refactored Code

The attribute of **modifiability** is satisfied by having the refactor code adhere to SRP. This ensures that each class has a focused purpose. As such, changing one aspect of the game, adding new player behaviour for example, can be done independently in their respective class (*Player* etc). **Extensibility** is also complied with by using well-defined interface and modular components. For instance, the IGameRules interface allows the implementation of different game rules, enabling future extensions of the game logic without altering the existing structure. The use of factories facilitates the addition of new game modes or variations without modifying the existing classes, thus promoting easy extension. With discrete classes with clear responsibilities, **testability** is also addressed. Each class can be tested independently through unit tests that focus on a single aspect of the game. This modular approach allows for testing individual components in isolation, making it easier to write and execute tests for each part of the system.

Q2.4 Motivation For Design Patterns

The motivation for using **factory pattern** is similar to a “factory” to create the objects instead of manually creating each object. This reduces repetitive code and ease adding new types of objects subsequently. To illustrate, *GameLogicFactory* is responsible for providing instances of *GameLogic* based on specific configurations or conditions, such as the choice of game rules.

Strategy pattern is also employed in the refactored code. This provides different alternatives to performing a task. A different strategy (or method) can be chosen based on the situation, allowing changes without a widespread change to the main system. For example, *IGameRules* interface defines the strategy for how the game is played, and concrete implementations like *StandardGameRules* provide specific behaviours. This allows the game to switch between different rule sets at runtime.

Q3 Refactor The Code to Match Proposed Design

Q3.1 Extent to which the code can be extended (new game modes)

New game mode: Apple's Eye View

Using the interface, *IGameRules*, a new class can be added for the new game mode rules which is independent of the standard rules. Assuming the win conditions are similar to that of the original game (number of winning cards to player ratio), existing components can be unchanged and reused (ScoringManager, CardManager, GreenApplesCalculator etc). This is due to high cohesion and low coupling of the refactored code. The new game mode requirement of judge's personality logic can be isolated and included within *GameContext* and *GameLogic*. Leaving the other classes untouched.

The refactored code is extensible and able to accommodate this new game variation with minimal changes due to its modular design, use of design patterns and adherence to SOLID principles and Booch metrics.

Q3.2 Extent to which the code can be modified (network, bot functionality etc)

As the refactored code for Apples2Apples adheres to SOLID principles, the code is modular and flexible, which enables it for future modifications. The usage of the factory pattern in *CardFactory* ensures that new card types can be added without altering existing code, as they can simply be registered and created dynamically. Interfaces like *IPlayerAction* and *ICardFunctionalities* define clear contracts, allowing new behaviours or functionalities to be integrated seamlessly.

For example, extending *BotPlayerAction* to create a *StrategicBotPlayerAction* with complex decision-making logic would involve implementing a new class without modifying the existing bot structure or logic. By adhering to the Open-Closed Principle, the code remains open to extensions while being closed to modifications, ensuring that enhancements like smarter bots or additional card types can be implemented without impacting the integrity of existing components.

Q3.3 Extent to which the code is designed for testability

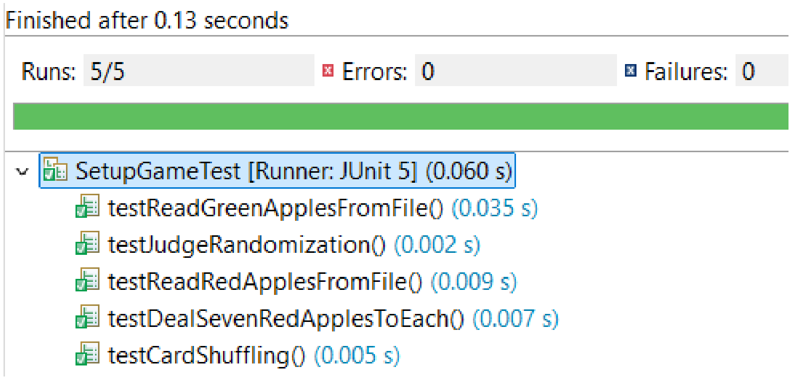
As the refactored code adheres to SRP, each class focuses on a single responsibility, making it easier to isolate unit for testing. For example,

- ScoringManager: This class manages scoring logic. It can be tested independently without requiring the full game to run.

Additionally, the usage of factory pattern decouples the creation of objects from their usage, allowing tests to configure specific scenarios. For example:

- CardFactory: As a result of factory pattern, GreenApple as a card type to be created dynamically. This enables the testing of card loading and content validation in isolation.

Q3.4 Extent to which the code is unit-tested (requirements 1-12)



```
Finished after 0.13 seconds
Runs: 5/5    Errors: 0    Failures: 0

SetupGameTest [Runner: JUnit 5] (0.060 s)
    testReadGreenApplesFromFile() (0.035 s)
    testJudgeRandomization() (0.002 s)
    testReadRedApplesFromFile() (0.009 s)
    testDealSevenRedApplesToEach() (0.007 s)
    testCardShuffling() (0.005 s)
```

Fig 2. Unit-testing for requirements 1 to 5

Requirements 1 to 5, which concerns the game setup, have been unit-tested with the requirements passing their respective test cases. Below is the elaboration for how each test case works:

Requirement 1: Read all green apples (adjective) from text file and add to green apples deck.

Test method: *testReadGreenApplesFromFile*

This test ensures that the green apple cards are correctly read from the file *greenApples.txt*, which contains adjectives and their synonyms. The test checks that the number of green apple cards is 614 (since the number of green apple entries in the text file is 614). It also verifies that the card name (adjective) and synonyms are not empty, ensuring proper data integrity (checking for not empty is opted as there are repeated adjectives).

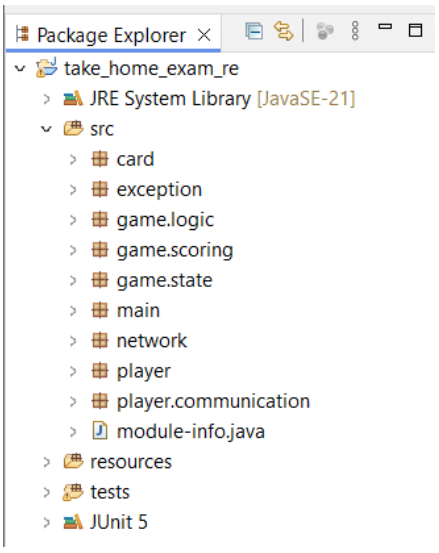
Requirement 2: Read all red apples (nouns) from text file and add to red apples deck.
Test method: *testReadRedApplesFromFile*
This test verifies that the red apple cards are loaded correctly from the redApples.txt file, which contains nouns and their descriptions. It checks that the total number of red apple cards is 1826 (same rationale as green apples, but value is 1826), and that neither the noun nor the description is empty, ensuring data correctness for red apples (rationale for checking for non-emptiness is the same as green apples, but it is nouns and descriptions).

Requirement 3: Shuffle both the green apples and red apples decks.
Test method: *testCardShuffling*
This test checks the functionality of shuffling for both green and red apple decks. It compares the order of cards before and after shuffling to ensure that the shuffling process works correctly. The test asserts that the order of cards in both decks' changes after shuffling.

Requirement 4: Deal seven red apples to each player.
Test method: *testDealSevenRedApplesToEach*
This test verifies that each player (including the judge) receives exactly 7 red apple cards. It uses a mock setup with human, bot, and judge players, checking that all players receive the correct number of cards. The test also asserts that all dealt cards are of type *RedApple*.

Requirement 5: Randomise start player to be judge.
Test method: *testJudgeRandomization*
This test ensures that the player who is selected as the judge is chosen randomly. It uses a custom *IJudgeSelection* strategy to randomly select the judge from a list of players. The test runs multiple times to confirm that the judge is not always the same, ensuring the randomization process works.

Q3.5 Extent to which the code follows best practices (structure, standards, etc.)



```
take_home_exam_re
  JRE System Library [JavaSE-21]
  src
    card
    exception
    game.logic
    game.scoring
    game.state
    main
    network
    player
    player.communication
    module-info.java
  resources
  tests
  JUnit 5
```

Fig 3. Project's code structure

The project is structed with these three source folders: src, resources and test.

The src source folder consists of all the code relating to the game Apples to Apples, with each package named according to their relevant components.

The resources source folder consists of the text files for red apples and green apples.

The test source folder consists of all the unit-testing code.

SOLID principles and Booch metrics were used as a guide to develop the refactor code. These principles and practices help ensure the code is modular, flexible and maintainable. Furthermore, design patterns like factory pattern and strategy pattern were adopted to promote low coupling and high cohesion.

Naming conventions were adhered to:

- Source folder and packages are in lowercase
- Classes abide by CamelCase (GameLogic, BotPlayer etc)
- Interface follows CamelCase too but begins with capital "I" (IPlayer etc)
- Methods uses camelCase, starting with lowercase letter (getHand() etc)

Q3.6 Extent to which the code correctly implements the functionality of the game

The refactor code has modularized all the components of the game and included additional parts such as an exception source folder for error handling. However, the main source folder, which contains the entry to the game, Apples2Apples.java has only implemented the setup portion of the game. This entails requirements 1 to 5, and these requirements have been unit tested as mentioned in Q3.4.

As such, the functionality of the game is only partially implemented. It still requires further development for the playing of the game and unit testing of those related components.

Q3.7 Extent to which the code handles and reports errors

To ensure the refactored code is robust and adherence with SOLID principles, error handling has been separated with each exception class scoped to its relevant domain. A table below to cover how the code handles errors:

Exception Class	Relevant Class	Handles Errors Related To	Key Errors
CardParsingException	RedApple GreenApple	Parsing card data (nouns, descriptions, synonyms) from input strings	Invalid card data formats (missing delimiters, malformed input etc)
CardManager-Exception	CardManager	Card file operations and card loading logic	File not found, file reading issues, invalid card creation, malformed data during file processing
JudgeManager-Exception	JudgeManager	Managing and selecting judges in the game	Invalid player list, judge selection failures
ClientConnection-Exception	Client	Errors during client-server connection initialization	Server connection failures. Issues with socket or input/output streams

GameplayInput-Exception	HumanPlayerAction	Errors caused by invalid player inputs during gameplay	Invalid card selection Invalid judgment input. Unexpected input errors during user actions.
InputInvalid-Exception	HumanPlayerAction	Validates input for card choices and judgment selections	Invalid input format (non-numeric). Choices out of valid range
NumberFormat-InputException	InputInvalid-Exception	Errors when parsing user input to a number	Non-numeric input provided by the player. Improper numeric strings

Q3.8 Extent to which the code is appropriately documented

As stated in Q3.5, the project is structured appropriately to help other developers navigate and locate files with ease. Classes and interfaces are named with their relevant purposes and naming conventions are abide to accordingly.

Comments have been added in each file to describe the file's purpose to add with developers understanding.

Furthermore, a README file has been included to highlight the development of the code, its usage and version history. This was done with the intention of helping future developers to continue with the project with ease and keep track of the project's progress.

Q3.9 Extent to which the code is true to the design in question 2

The refactored code adheres to the designed outlined in question 2. Fulling the qualities of modifiability, extensibility and testability as per SOLID principles and Booch metrics. The use of Factory pattern in classes (like *CardFactory*) centralises object creation, supporting modification, allowing new card types/ behaviours to be added without altering existing code. This ensures OCP, where the system is open for extension but close for modification. Strategy pattern is utilised in *IGameRules* and *IJudgeSelection*, enabling dynamic switching of game rules or judge selection strategies without impacting the core structure.

The modular design improves modifiability by isolating concerns, while extensibility is ensured through interfaces. Testability is achieved through high cohesion and low coupling between classes.

By adhering to the best practices, the refactored code promotes flexibility, making future modifications—such as game variations, new player behaviours, or advanced bot logic—easier to implement without disrupting the existing system.