Omtenta D7032E - Apples 2 Apples

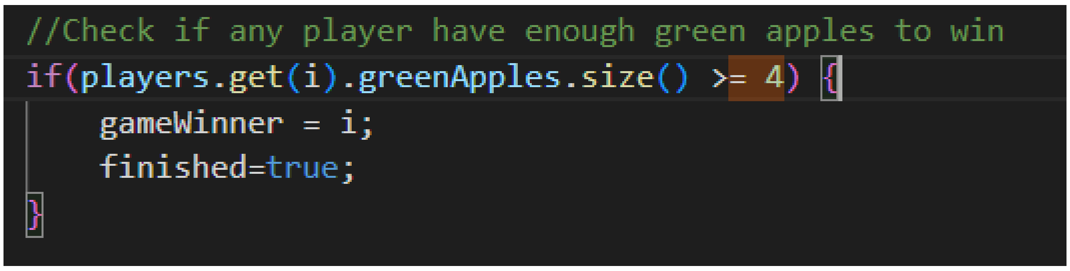
Christoffer Lindkvist
icoili-9@student.ltu.se

Christoffer Lindkvist
icoili-9@student.ltu.se
D7032E Hemtenta

1. Unit testing

The implementation of Requirement 15 failed because the game always ended as soon as a player accumulated 4 green apples, regardless of the number of players. This behavior was due to the if-statement (see picture) becoming true once a player scored 4 points, resulting in that player winning the game without considering the total number of players.

This cannot be tested as there is no return value to test, nor is it a function that can be called upon.



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

Christoffer Lindkvist
icoili-9@student.ltu.se
D7032E Hemtenta

2. Quality attributes, requirements and testability

Requirements 16 and 17 are poorly written due to the lack of specificity and measurability, which can lead to issues during development. The vague and subjective terms used in these requirements make it difficult to determine if the software meets them, and developers may have different interpretations of what they mean. This can result in inconsistent implementation and inadequate testing of the software.

Christoffer Lindkvist
icoili-9@student.ltu.se
D7032E Hemtenta

3. Software Architecture and code review.

3.1 Extensibility:

The code's high *coupling* and lack of *cohesion* make it challenging to add new features without refactoring. The majority of the code is contained in a single constructor with minimal methods, resulting in many unnecessary dependencies. To improve the program's *extensibility*, it's essential to split the code into additional methods and classes. This will make further implementations more manageable and less dependent on other parts of the code.

3.2 Modifiability:

The code's *primitiveness* and limited methods make it difficult to modify the game mechanics and rules. Developers must make many changes to the code when modifications are necessary, as there are no methods available to handle most of the changes. For example, hardcoded comparison limits make it challenging to modify the code. Converting these limits into final variables would improve the code's completeness, making it easier to find and modify the limits.

3.3 Testability:

The code's limited methods make it challenging to write tests that compare return values effectively. As a result, it's difficult to ensure that all the requirements are adequately tested. To enhance the code's *testability*, it is necessary to increase the number of methods available for testing and ensure that all requirements are thoroughly tested.

Christoffer Lindkvist
icoili-9@student.ltu.se
D7032E Hemtenta

4. Software Architecture design and refactoring

The main focus when rewriting the code was to make it easily testable. To achieve this, we split up a lot of the code into smaller methods and tested every requirement along with a few extra conditions.
In Rust, structs are similar to objects and work in a similar way. Let me explain how each struct in the software operates:

RedDeck: RedDeck is a struct which contains a vector of RedCard, it has the traits Setup which contains the function read_cards() which reads the cards from file and places them in the cards variable of the struct. On its own it's got a draw function which draws and returns the top card, shuffle() which shuffles the deck, and add_to_deck() which adds more cards to the deck, used by read_cards().

GreenDeck: Similarly to RedDeck GreenDeck is a struct which contains a vector of GreenCard, it implements the Setup trait, and it has a shuffle() function which shuffles the deck using the fisher yates algorithm, and a draw() function which returns the top card of the deck.

Discard: Discard is a struct which contains a vector of RedCard, it has the functions add_to_discard() which adds a red card to discard, and get_size() which returns the size of the discard pile.

RedCard & GreenCard: RedCard and GreenCard are both structs that contain two strings; a title of the card, and the description (desc). Both RedCard and GreenCard implement the "CardTraits" trait which has the get_title() function which returns the title as a string, and a get_desc() function which returns the "description", or secondary line of text, as a string. Redcard can also be wild which means it asks for user input.

Player: Player is a struct which contains the player_id as *i32*, is_bot as *bool*, hand as a vector of *RedCard*, and green_apples as a vector of *GreenCard*. Player implements the PlayerActions trait, and the Judge trait. If is_bot is true then the player will be fully bot-controlled.

Settings: Settings is a struct which holds, you guessed it, different settings for the game, it can be created using one of two factories, default_settings() which runs the game in default mode, and custom_settings() which gets assigned in Menu.rs. Settings support the number of wild apples, bots, hand size, judge/vote bool, and the win requirement. It was designed to make it easy for the player to customize the experience.

Menu: While Menu and its sub functions are not a struct, it was added to make it easier for the end-user to play the game, and to set custom game rules. It's basically as looped prompt which ties into settings, game, client, and server.

Game: Game is not either a struct, however it's got two functions, init_game() which sets up the game by adding the decks, players/bots, and reading the settings, and gameplay() which will keep looping the game per the requirements and phases described until a player wins.

Client and Server: I never got networking to work properly, what you see here is a lot of sample code I wrote in an attempt to make something. Instead the entire game is played with bots for now.

Design Patterns:
The use of factories is a good design pattern for creating complex objects in Rust or any other programming language because it allows for encapsulation of object creation and a single interface for creating them.
The player_factory in Player.rs and the discard_factory, red_deck_factory, and green_deck_factory functions in cardpiles.rs, etc which can create objects without requiring knowledge of their implementation details, making it easy to change their implementation without affecting other parts of the code. Factories also help reduce code duplication, simplifies the code, and improves modularity and maintainability.