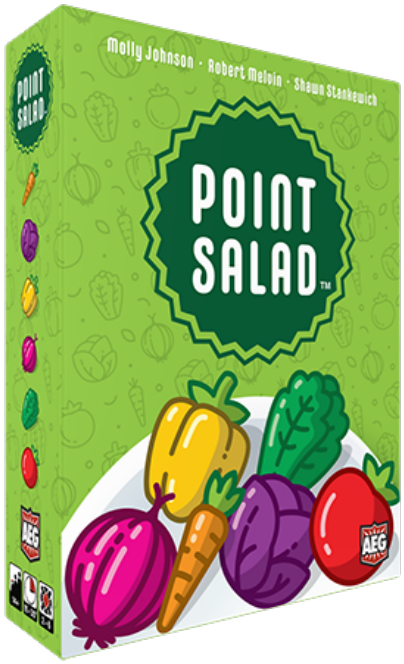
D7032E Home Exam

My Pointsalad implementation



`Software engineering, D7032E`

Student	*E-mail*
Ludvig Järvi	ludjrv-1@student.ltu.se

LULEÅ TEKNISKA UNIVERSITET

Department of Computer Science, Electrical and Space Engineering

October 23, 2024

Contents

1 Unit testing (2p, max 1 page)

Requirement 1 and 3 are not currently fulfilled by the code.

Firstly, for requirement 1 that says: *There can be between 2 and 6 players*, there is no prevention method for catching cases where the number of players is lower than or greater than these limits.

It is not possible to test this requirement without modifying the existing code as there is no differing output for a game with one player and one bot compared to a game with 2 players and 5 bots.

Secondly, requirement 3 that says: *Form the deck so that the amount of random vegetables depends on the number of players*, is currently implemented using integer division. The reason this is insufficient is because when a number is divided into a decimal number it is rounded down into the nearest whole number. Thus, when a game with three or five players is created, the number of vegetables is rounded down to the same number as for two or four players, respectively.

The requirement is able to be tested without modifying the existing code with:

```
@Test
void testPileSizeBasedOnPlayers(){
    for (int i = 2; i <= 6; i++) {

        ArrayList<Pile> piles = new ArrayList<>(3);
        setPiles(i);

        int expectedTotal = 18 * i;

        int actualTotal = 0;
        for (Pile p : piles) {
            actualTotal += p.size();
        }

        assertEquals(expectedTotal, actualTotal);
    }
}
```

2 Software Architecture design and refactoring (9p, max 2 pages excluding diagrams)

The main part of this exam was to redesign and refactor the existing Point Salad codebase with a focus on **Modifiability**, **Extensibility** and **Testability**. This process was guided by following best coding practices according to SOLID principles, Booch's metrics and software design patterns.

The reason behind the refactoring was to ensure that the code could accommodate future development, specifically the integration of a new variation of the Point Salad game. This future version is expected to introduce additional game mechanics, which will be built on top of the refactored code.

2.1 SOLID principles & Booch's metrics

The original Point Salad code was written as a single class, or a single module if you will, with many methods for all of its various needs. Although the code itself is sufficient in terms of playing the game, it does not satisfy best practices as the code has high coupling, low cohesion and low primitiveness since many components with different functionality are located together.

Solving these issues by refactoring the code can be done by looking at the relevant SOLID principles:

- We want each module to only handle information relevant to its own responsibilities. This is accomplished by reorganizing the methods into separate classes that can instead be instantiated when necessary. The resulting effect is a code with *low coupling*, *high cohesion* and *high primitiveness* which increases legibility for new developers as each module clearly state what its core purpose is.
- By implementing interfaces and abstract classes for closely related modules, we allow for extensions in the form of new game functionality without having to modify the existing code. We also avoid having code that depends on concrete implementations by instead having the interfaces along with dependency injection wherever applicable. This further decreases the level of *coupling* in the code.

2.2 Applied design patterns

2.2.1 Template Pattern

We want each class or module to have its own distinct purpose while still being **extendable**, therefore the *template pattern* was chosen as a main design point, separating the classes into their own folders inside the main src folder. In these folders, we create interfaces which acts as general contracts that describe the required methods for the related classes. We use abstract superclasses to share common attributes and methods among subclasses in order to **reduce duplicate code**, such as in the *player class* in figure 1 or in the *card class* in figure 2. These are in turn extended with specialized behavior based on needs.

2.2.2 Mediator Pattern

By implementing a *gamestate* class (figure 3) that keeps track of all game variables, we can achieve the function a *mediator object*. This gamestate object is passed into the different methods, where the object's *get* and *set* methods are used to to update the necessary values, ensuring **consistent** and **centralized** control over the game's state and **minimizing dependencies** between individual components. Another benefit of keeping the game variables in a centralized object is **ease of testing**, as we can simply pre-configure any desired game state.

2.2.3 Facade Pattern

When it comes to games, having a **simple user interface** is crucial, therefore a *facade pattern* is used to simplify the creation of the game process by using subsystems that are instantiated in the main function. In this case we have methods such as *GameSetup* and *GameLoop* that are created as subsystems, these in turn create their own subsystems which creates the necessary components for running the game. Advantages of using this are that any new features that we want to add into subsystems can be integrated **without changing** the overall code structure.

3 Appendix

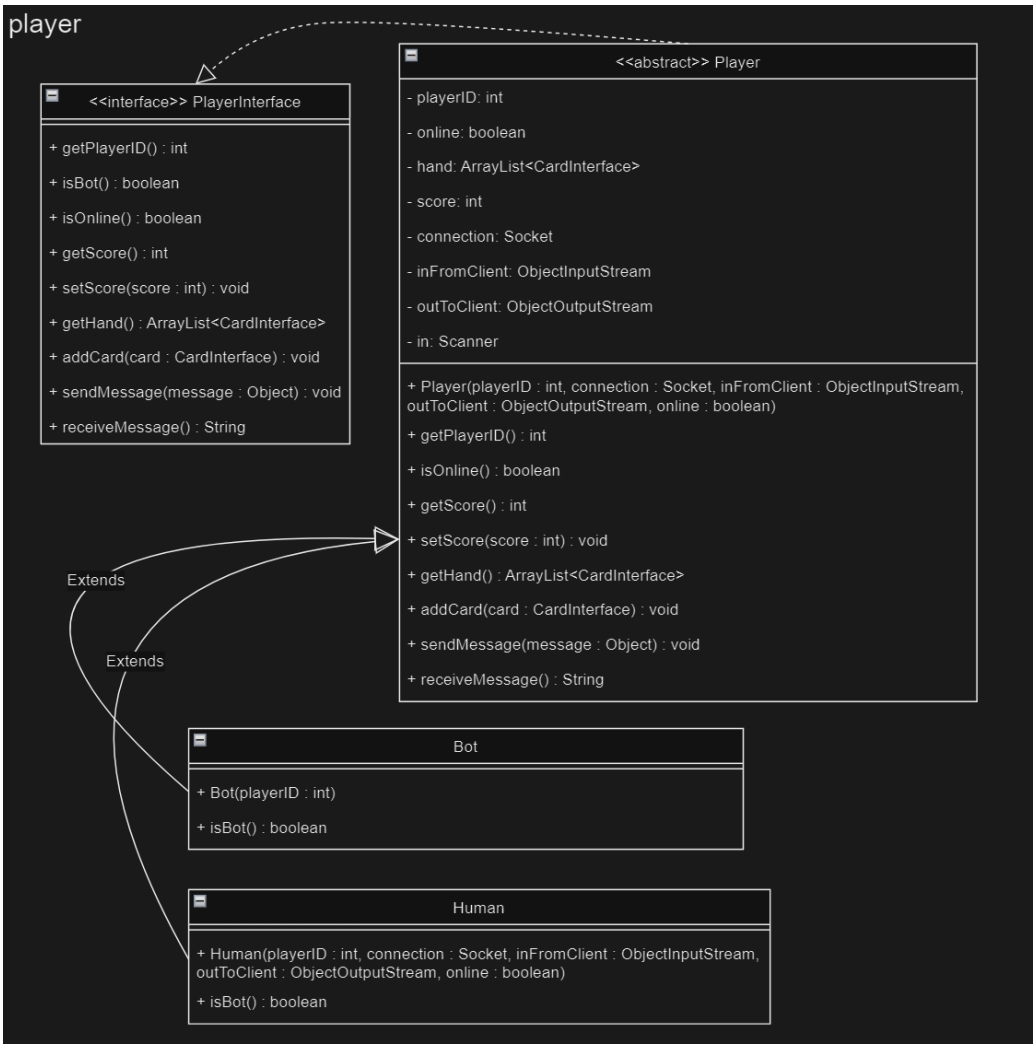


Figure 1: Player folder structure

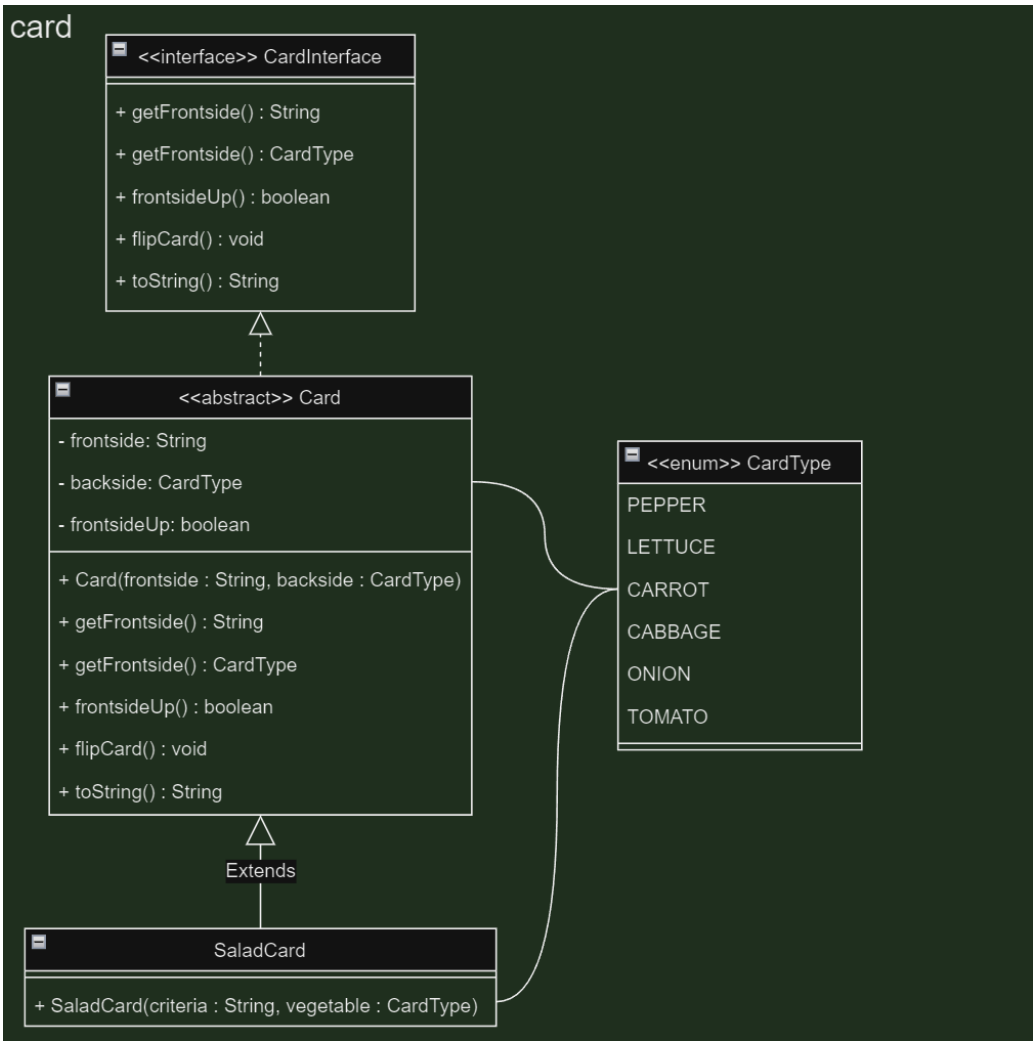


Figure 2: Card folder structure

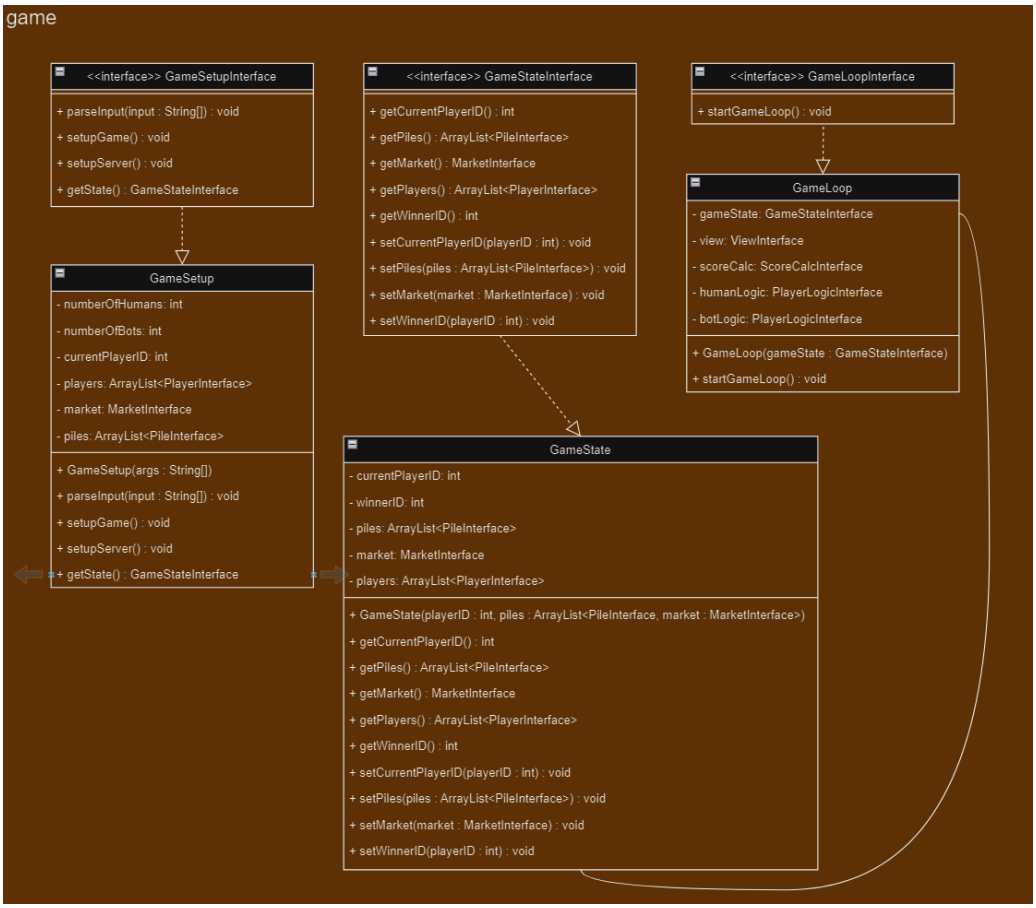


Figure 3: Game folder structure