Peer-Review 1: UML

Group AM-02

Alberto Cantele, 10766393 Andrea Cioccarelli, 10713858
Giulia Bortone, 10792907 Marco Carminati, 10790205

March 23, 2023

Initial UML diagram evaluation, considerations and comments for group AM-11

1 Positives

The provided UML class diagram implements most of the game logic and model features required by the project specifications.
The diagram is capable of holding the data compatible with the complete game rules.

- All the game data is kept inside the game model as a Single Source Of Truth, accessible from the controller component on a higher level. This is adequate for having a data serialization and persistence scheme directly tied to the game model;
- An idiomatic approach is used for implementing the `CommonGoalCard`, where the acceptance logic is deferred to an appropriate `Predicate<Bookshelf>` which takes care of computing and determining whether a certiain `Bookshelf` satisfies the given property;
- An extraction class `CountCards` is used to store the extraction state (which tiles have already been used) and to produce a random element, which is used by the game model itself to manage tile replacement and board filling;
- The `GameInterface` class abstracts away the controller-related methods and let them be implemented by the `Game` model class, which is an appropriate and correct usage of inheritance;
- The `GameStatus` class controls the cyclic player status and can be used by the controller to handle client communication and business logic;
- The `GameManager` class falicitates the multi-game server design and player connection logic.

2 Negatives

Overall, the diagram exposes a weak usage of inheritance, poorly planned serialization capabilities, and unnecessarely convoluted logic.

2.1 Inheritance, OOP and idiomaticity

- The use of inheritance in the `Goal` class seems contrived and unnatural, since both classes share only one method, which has a completely different implementation in each one of them;
- The usage of a `Map<Position, Optional<CardType>>` in `Board` is an interesting approach because it solves the problem of having non-valid coordinates, which would instead be present with a matrix-based implementation;
  However, it does so while greately reducing readability (a `Map<K,V>` is not supposed to be used as a replacement for a matrix), and looks like a non natural solution. The matrix based approach remains by far the simplest, most intuitive and comparetively cheap solution;
- The `Token` class, as implemented here, is used exclusively as a wrapper for an integer value. It does not provide functionality or retain any other state. And especially in this type of situation, where the domain of possible values for the choosen representation of the game tokens is finite, it would make for a much better solution to use a closed-domain data type, such as a sealed class or an enum;
- Within the `Player` class, the concept of acquired `Token`s should suffice for automatic calculation of current point value for a given user. Instead, those two concepts are handled separently in a non-intuitive and redundant approach;
- The model implements all the possible valid `CommonGoal`s within the `GameManager`. That represents the domain of all possible common goal cards. A singleton pattern would fit better for this use case;
- The model lacks a disconnection (and eventually, exiting) status for the players.

2.2 Serialization and Persistence

- The copious amount of `Optional<T>` usage across the model makes it, albeit idiomatic, harder to serialize;
- The player model seems to lack a concept of acquired `CommonGoalCard`, which if not properly implemented in the game logic will result in a situation where it is impossible to differentiate which kind of `Token` has been acquired by which `Player`, thus leading to a possible repeated acquisition of the same `CommonGoalCard` by the same player, which goes agains the game specifications. This could be mitigated by associating the types of acquired `CommonGoalCard` with a player's game session;
- The model seems to lack a way to store the intermediate player tile selection (the set of selected tiles, before insertion in the player's bookshelf), which is a weak point for pesistance, since if that information isn't kept it can be lost.

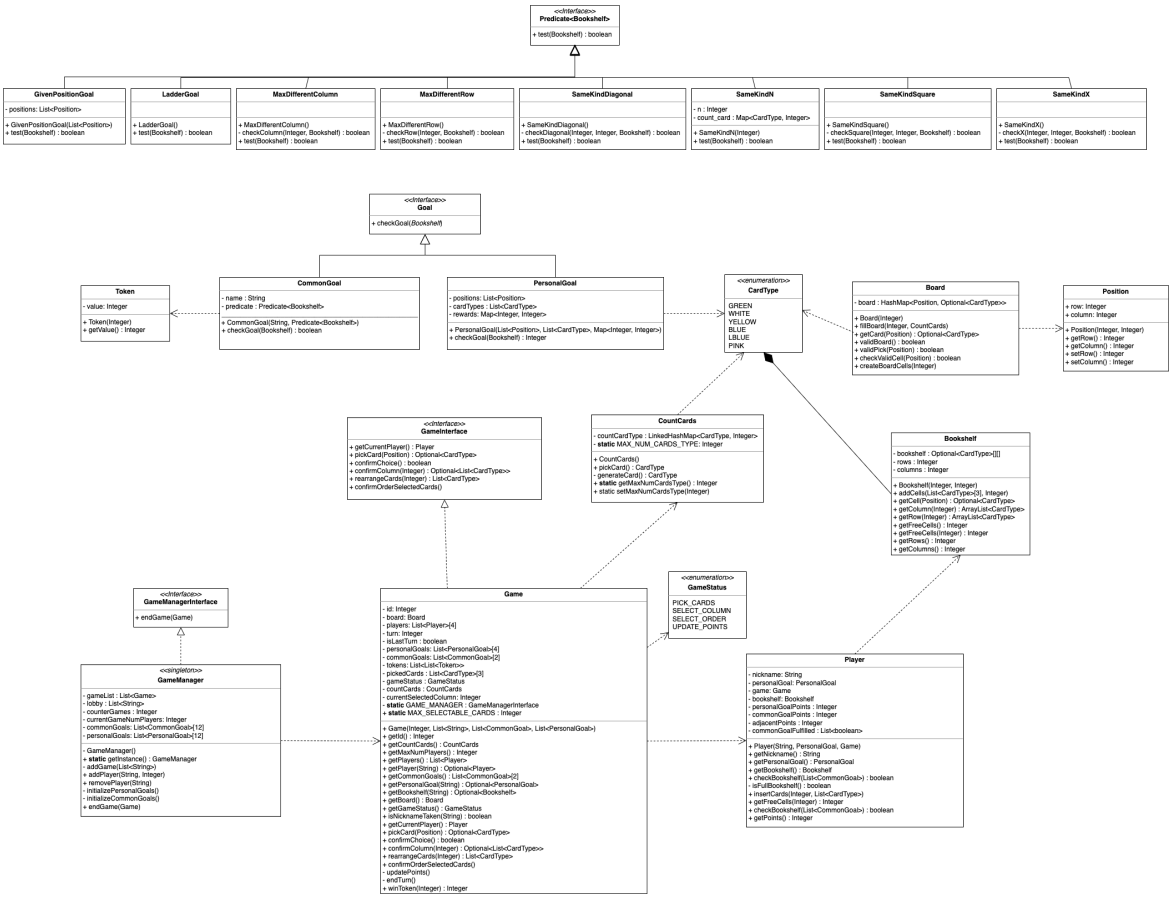
2.3 Form and Java-specific aspects

- The diagram shows an eccessive usage of the `Integer` type. It is often better to use the primitive (unboxed) versions of those types (in this case `int`), unless when dealing with generic types or other particular scenarios;
- The methods implemented from an interface should be absent in the class, according to the UML standard.

In addition to this, we noted that names in the model do not match the naming convention given inside the actual game (i.e. `CardType`s in the diagram should be named `TileType`s).

3 Architecture comparison

The reviewed diagram outlines ao overall good design for handling more than one game simultaneously, along with a good player lobby logic and general robust handling of the game model itself.
Additionally, it makes a fairly good usage of interfaces for separating functionality and decopouling class logic, along with functional programming concepts.
And finally, we implemented in our model a similar concept to `GameStatus.UPDATE_POINTS`, which is a state meant to give an explicit status to the point update, token assigation and full shelf checks.



.