GC10 UML

Riccardo Polelli Filippo Pozzi Federico Quartieri Giacomo Tessera

Index

List of Figures

Model

Card

This section provides a concise overview of the class structure we implemented to represent the different card types.

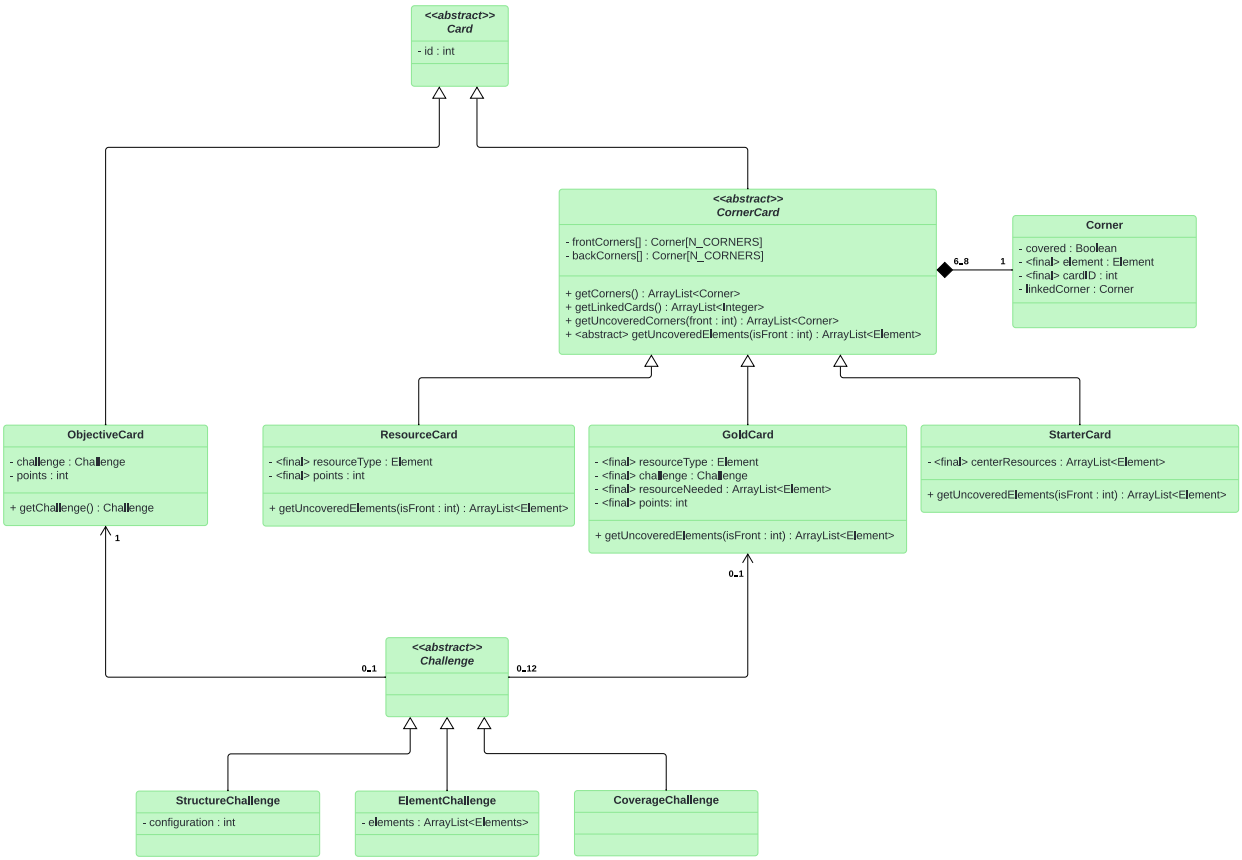

Figure 1: UML of Card

The following table summarizes the common attributes shared by all card types, facilitating their organization into classes and subclasses.

Card	Has Corner (4)	Has Resource Corner (max 4)	Has Item Corner (max 1)	Has Points	Has Challenge	Has Resource Needed (max 5)	Has Back Resource
Resource	x	x	x	x			x
Gold	x		x	x	x	x	x
Starter	x	x					x (max 3)
Objective				x	x		

Within the “cards” package, all classes and interfaces are declared final except for `CoveredCorner` and `LinkedCorner` in the `Corner` class.

CornerCard

The array representation utilizes `null` to signify a hidden corner. Conversely, a visible corner with no element present is included in the array and marked with `EMPTY`.

- `covered` : true if there is another card on top, else false
- `element` : represents the content occupying this corner. It can be a `Resource` or an `Item`. If no element is present, the value is set to `Element.EMPTY`
- `cardId` : identifies the card this corner belongs to
- `linkedCorner` : points to another `CornerCard` if the two corners are connected. A `null` value signifies the corner isn't linked to any other card.
- `getUncoveredCorners` : returns a list of all currently uncovered corners
- `getUncoveredElement` : returns all the elements (`Resource` and `Item`) of all currently uncovered corners

Corners order:

```
01
32
```

GoldCard If the `challenge` attribute is `null`, points are awarded automatically when the card is played. The `ResourceType` defines the type of resource (also serves as the color identifier).

ResourceCard The Resource Card are the only type of card that requires resources to be places.

StarterCard The card's front side consistently features four corners, each containing an `Element`. Conversely, the back side's corners can be hidden (represented by `null`) or empty (marked with `EMPTY`). Additionally, the back side may include `backResources`, to represent resources specific to the back center.

ObjectiveCard

A `Challenge` is a specific task a player must complete to earn points. Conversely, `Objective` refers to the overall category or type the card belongs to.

Challenge

The `Challenge` attribute represents a specific task a player must complete to acquire points. It's employed within both `ObjectiveCard` and `GoldCard` classes. Further details regarding Challenge functionality will be provided later.

Structure Challenge The structure challenge is used only for objective cards.

The `configuration` attribute is a 3x3 matrix of `Elements` (specifically Resources). The `Element` refers to the `resourceType` of the card and the position is determined by the position in the matrix.

Order from top-left to bottom-left [0-3]

Element Challenge `Objective` cards leverage the `Challenge` attribute to define tasks players must complete to earn points. These tasks exhibit variability in the types of resources required.

Here's a breakdown of how resources are typically handled within objective cards:

Coverage Challenge This challenge is used exclusively for gold cards. The challenge defines a point-awarding task based on the quantity of specified `Elements` present on the player's board.

Note that for the `GoldCoverageChallenge` the value of `points` is always 2.

Figure 2: Example of Structure Challenge

Figure 3: Example of Element Challenge in an Objective Card

Figure 4: Example of Element Challenge in a Gold Card

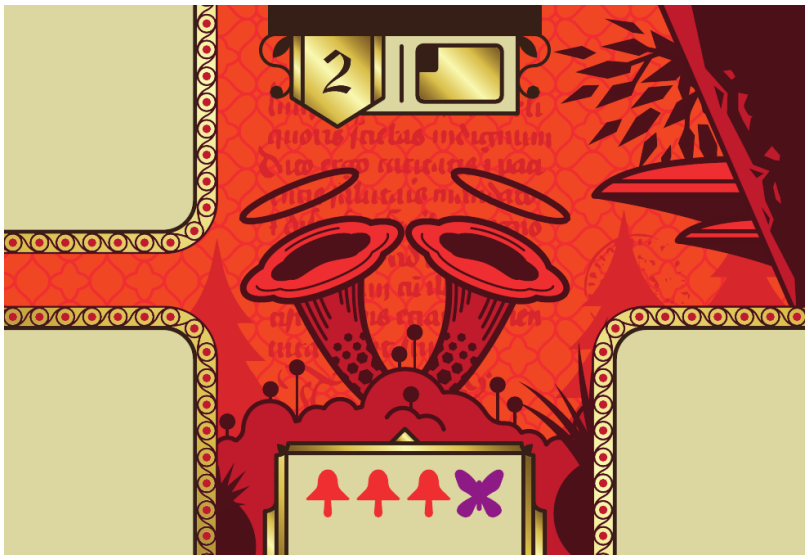

Figure 5: Example of Coverage Challenge

GameState

The `GameState` is the most important class. It is where all the information regarding the game are stored. In this section we will discuss some of its main features.

The first player (index = 0) in the data structure is the Black player, `public Player getBlackPlayer() { return player[0] }`. Note that a **round** is made up by 4 **turns**.

The players order in game is defined by the order of the players in the `players[]` array.

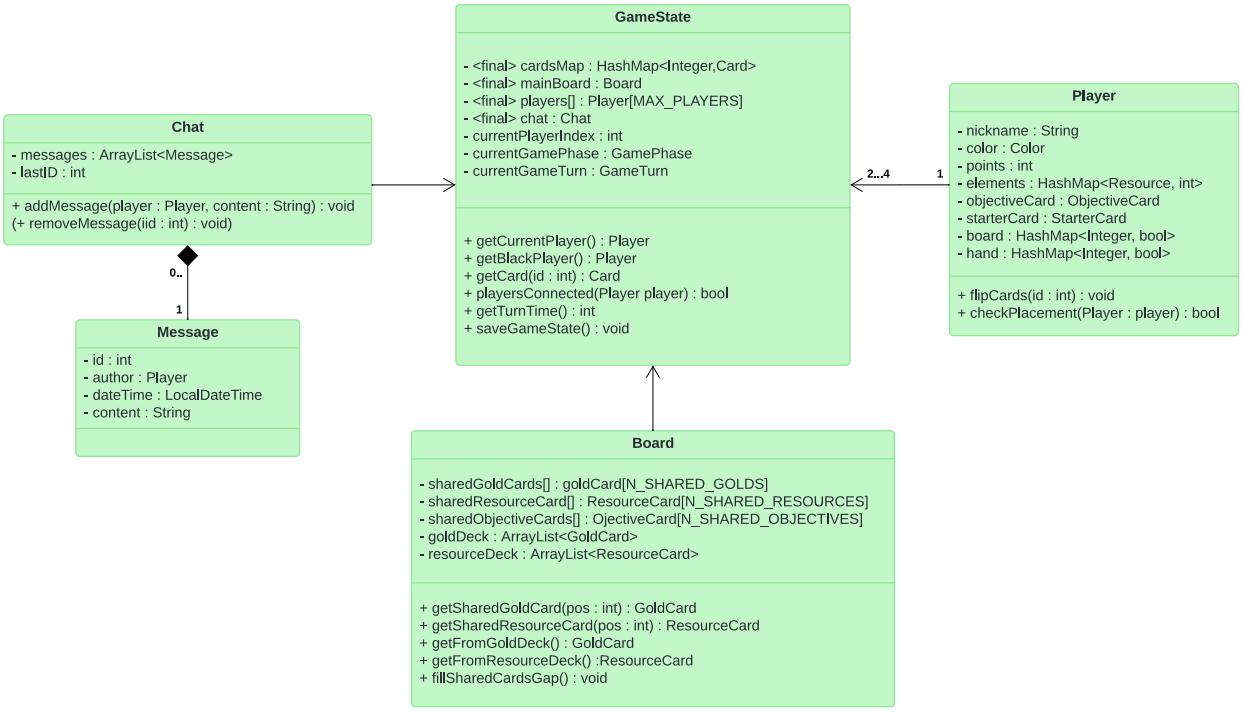

Figure 6: UML of GameState

In the methods of the `Board` class, `pos` specify the position of the card (1 if it's the first, 2 if it's the second one)

Enumerations

This section details the enumerated data types we'll employ to represent various game concepts. Due to graphical complexity, we won't illustrate all associations with other classes. However, it's important to remember that these enumerations are indeed connected to the classes that utilize them for data definition.

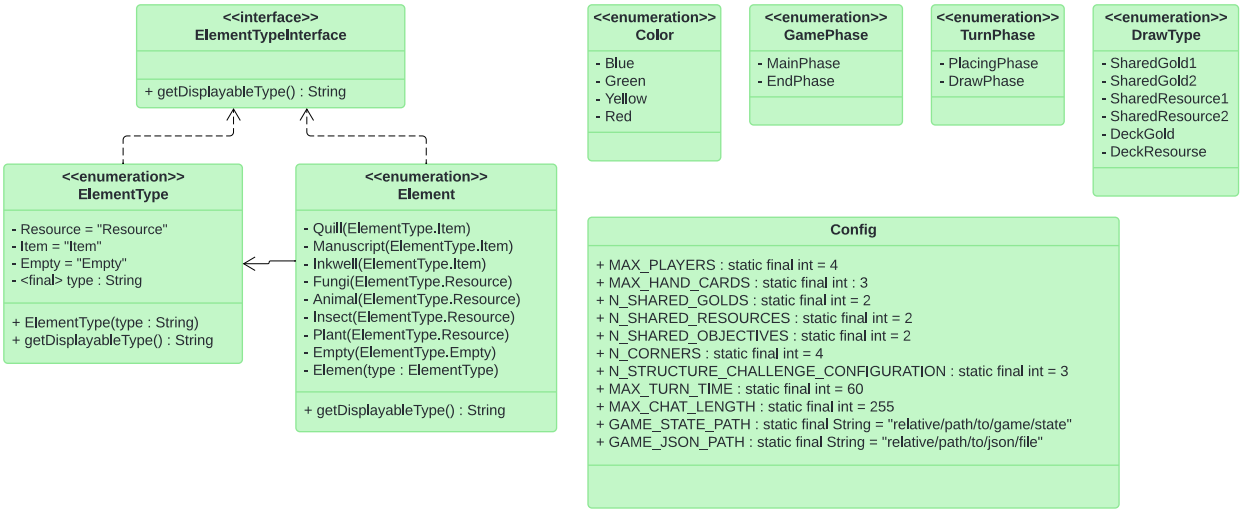

Figure 7: UML of Enums and Config

Controller

- `drawCard` : this method retrieves a card from the appropriate deck based on the requesting card's class. For instance, if the requesting card belongs to the `ResourceCard` class, the method would draw from the resource deck.
- `placeCard` : while a single card placement is required through this method, you can still gain information about any affected multi-corner cards by examining the method's return value. The `cornerTableIndex` parameter specifies the corner on the table where the player intends to connect the card.
- `getGameState` : this method provides a comprehensive overview of the current game state.
- `flipCard` : this method flips the card currently held by the player. Subsequently, the placeCard method will utilize the flipped side for placement.

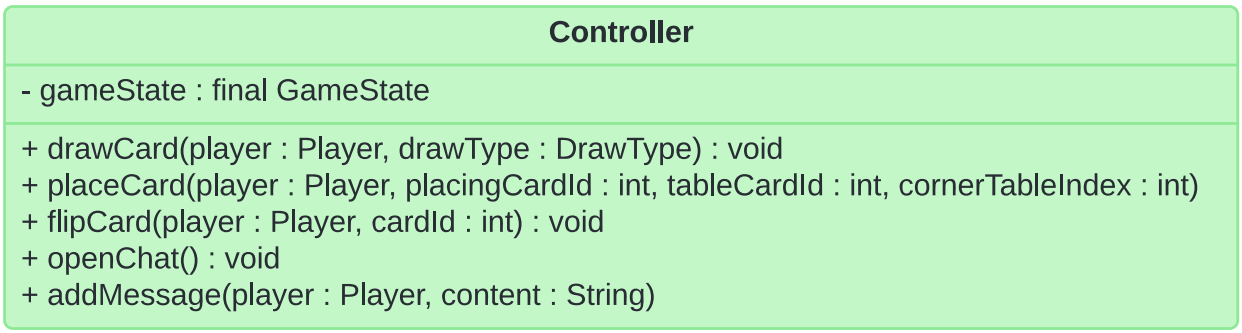

Figure 8: UML of Controller

Exceptions

While testing the constructors and some methods we noticed that defining some exceptions would have helped us a lot in managing problems. Here are some of the examples:

- `InvalidHandException` (more than 3 cards in the hand)
- `NotUniquePlayerColorException` (two players have the same color)

- `NotUniquePlayerNicknameException` (two players have the same nickname)
- `NotUniquePlayerException` (two players have the same color and the same nickname)
- `WrongStructureConfigurationSizeException` (structure is the one in challenge)

Complete UML

(you can *zoom* it)

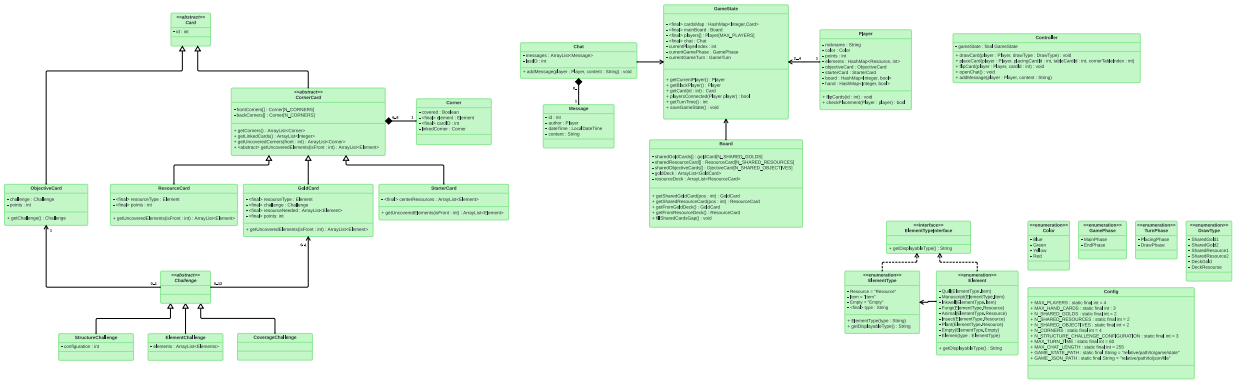

Figure 9: Complete UML