Object Oriented Programming - Project

Maraziaris Charalampos `sdi 1800105`

Chalkias Spyridon `sdi 1800209`

February 2020

1 Usage

You can have access to Project's **parent directory** by typing :

```
$ cd ~/Card-Game
```

1.1 Compile

Give the following command (while being in the parent directory) :

```
$ make
```

1.2 Run

In order to play the **Card Game** type (while being in the parent directory) :

```
$ make run
```

or

```
$ ./cardGame
```

1.3 Clean

Remove the **executable** and all the remaining **object files** by typing :

```
$ make clean
```

1.4 Memory Leak Checking

The following command requires Valgrind installation if needed.
To see a brief overview of all the project's **memory usage** type :

```
$ make check
```

2 Project Structure

- **main.cpp**: Test main.
- **Makefile**: Easy compilation, run & memory check of the project.

2.1 ./files/

- **rules.hpp**: Rules of the game, such as stat-file locations, and #define'd values.
- **Followers_and_Weapons.txt**: Stats for cards of classes: "Follower", "Items" and their sub-classes.
- **Personalities_and_Holdings.txt**: Stats for cards of classes: "Personality", "Holdings" and their sub-classes.

2.2 ./src/

A file called **make.inc** exists in almost every directory. It acts as a component of the global **Makefile** file found in the root of the project.

- **basicHeader.hpp**: Declares every class of the project, includes important typedef'ed values, every local header file and UNIX terminal color values & styles.
- **templateFunction.hpp**: Defines a template function that pushes a given number of cards (Green or Black) into decks (Fate or Dynasty).
- **prints.cpp**: Every method that prints to the standard output, regardless of class, is implemented here.

2.2.1 ./src/Cards/

- **basicCards.hpp** : Defines class: "Card"
- **blackCards.hpp** : Defines classes: "BlackCard", "Personality", "Holding", "Mine", "GoldMine", "CrystalMine", "StrongHold" Defines enum classes: "BlackCardType", "PersonalityType", "HoldingType"
- **greenCards.hpp**: Defines classes: "GreenCard", "Follower", "Item" Defines enum classes: "GreenCardType", "FollowerType", "ItemType"
- **mines.cpp**: Constructors and methods of every mine-related class.
- **cards.cpp**: Constructors and methods of every class described above, except mines-related ones.
- **proverbs.txt** : Card text.

2.2.2 ./src/Gameplay/

- **game.hpp** : Defines class: "Game"
- **game.cpp** : Reads the information files (*Followers_and_Weapons.txt & Personalities_and_Holdings.txt*) , builds and shuffles all the players' decks and initializes the game board. Also , every ***Class Game's*** member function except from the phases' functions and prints are defined there.
- **player.hpp**: Defines classes: "Player", "Province"
- **player.cpp**: Constructor and methods of classes "Player", "Province"

2.2.3 ./src/Phases/

<*phase_name*> can be one of the following: battlePhase, economyPhase, equipPhase, startingPhase, finalPhase.

- ***<phase_name>.cpp***: Implementation of <*phase_name*> phase, Game::<*phase_name*> method.

3 Implementation & Features

3.1 C++11 Features

The key C++11 features that have been used to project's implementation are Smart Pointers & Enum Classes.

std::shared_ptr is a smart pointer that retains shared ownership of an object through a pointer. Several shared_ptr objects may own the same object. The object is destroyed and its memory deallocated when either of the following happens:

- The last remaining shared_ptr owning the object is destroyed.
- The last remaining shared_ptr owning the object is assigned another pointer via operator= or reset().

Enum classes (also called **Scoped enumerations**) , make enumerations both strongly typed and strongly scoped. Class enum does not allow implicit conversion to int, and also does not compare enumerators from different enumerations.

Their advantages over traditional enums (Unscoped enumerations) are :

- Conventional enums implicitly convert to int, causing errors when someone does not want an enumeration to act as an integer.
- Conventional enums export their enumerators to the surrounding scope, causing name clashes.

- The underlying type of an enum cannot be specified, causing confusion, compatibility problems, and makes forward declaration impossible.

Therefore , **Enum Class** combine aspects of traditional enumerations with aspects of classes (scoped members and absence of conversions) , making the code much safer.

3.2 Data Structures

In project's implementation , the selected Data Structures are **std::list** and **std::deque** , which can be found in C++ 's STL.

Lists are sequence containers that allow constant time insert and erase operations anywhere within the sequence(**dynamic growth**) and iteration in both directions.

List containers are implemented as **doubly-linked lists**; Doubly linked lists can store each of the elements they contain in different and unrelated storage locations. The ordering is kept internally by the association to each element of a link to the element preceding it and a link to the element following it.

Deques (Double ended queues) , allow for the individual elements to be accessed directly through random access iterators, with storage handled automatically by expanding and contracting the container as needed.
They provide a functionality similar to vectors, but with efficient insertion and deletion of elements also at the beginning of the sequence, and not only at its end.

Game Class has a **list** consisting of Players.

Player Class contains :

- a **deque** of GreenCards (**Fate Deck**) & BlackCards (**Dynasty Deck**).
- a **list** of GreenCards (**Hand**) , Personalities (**Army**) , Holdings & Provinces.

Personality Class has a **list** of Followers & Items.

Overall , it has been decided that the above data structures fit best , taking into consideration all of the project's needs.

3.3 Inheritance & Composition

3.3.1 Cards

Figure 3.3.1 shows the correlation between the various types of cards implemented in this project. We decided to use the inheritance mechanism until the level of Followers, Items, Personalities and Holdings, with the exception of

some Holding-type cards (StrongHold & every type of mine). Further types depicted as inherited below the 4 classes described above, are implemented using enumerated object-orientation.

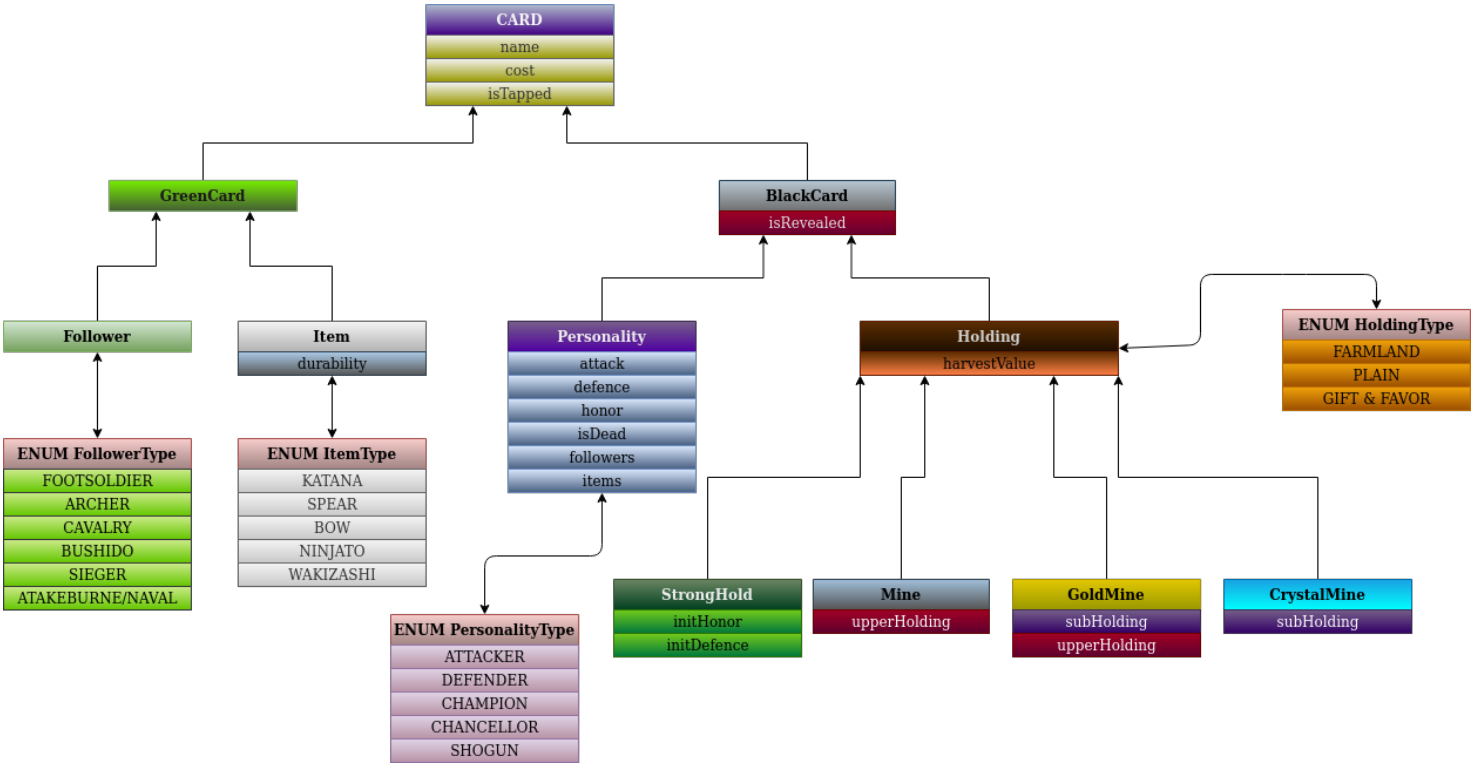

Fig. 1: ***Card class*** diagram

3.3.2 Player

Figure 3.3.2 shows the components of the Player class. Composition is implemented using Pointers to the desired objects, or data structures containing these objects.

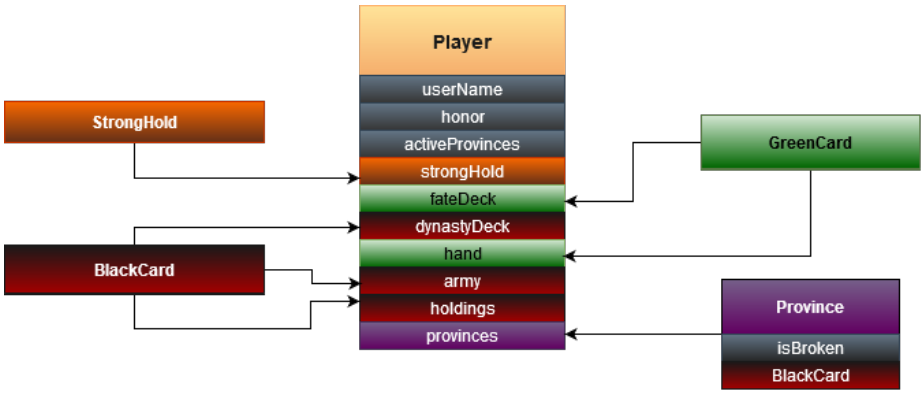

Fig. 2: ***Player class*** diagram

4 Gameplay

4.1 Prompts & Decision making

CardGame is playable! Its gameplay is fully interactive and there are detailed instructions throughout the game. There is a variety of prompts , that offer players the ability to fully control their own unique strategy during the match. Some examples are :

- Type "Y" or "y" for **Yes** and "[*any other key*]" for **No**, in order to buy a province / equip a personality / upgrade a card / discard a surplus card etc.
- Type "ATK" or "atk" to **Attack** and "DEF" or "def" to **Defend**.
- Choose a player to attack , by typing his username.
- When preparing to attack , select one of the defender's four provinces , by typing a number in range [1,4].
- Decide if you want to terminate the game prematurely.

4.2 Game progress

Some notes regarding the way the game progresses:

- Deck out of cards: When a deck runs out of cards, nothing happens. The game shall end with the cards still remaining in the game.
- Battle Phase - Attack: When a player is attacked, he gets to pick an Army to defend, choosing from his un-tapped personalities.
  Note that the defending player **CANNOT** select a Personality bought in the current round.

- Broken Items: When an Item's durability drops to 0, the item is removed from the game.
- Dead Followers/Personalities: When a Follower/Personality dies, it is removed from the game.
- Buy process: When a player is about to purchase/upgrade a card, the game prints *only* the cards affordable by the player. Other cards remain intact, but are not shown in order to speed up the process.
- Personality equipment: Every Personality has a limit, regarding the maximum number of Followers/Items of each kind, that can be equipped at once.
  For example, no Personality can have more than 2 Followers of type "FOOTSOLDIER" at any time during the game. These limitations can be found in the file "rules.hpp".