Pandemic Board Game

By: Christopher McArthur 40004257

Team Cerberus

1. Overview

This projected was build under a very object oriented umbrella within a pseudo MVC architecture. The game is cooperative strategy where players move around the world trying to cure diseases. It is currently a C++ console applications for windows (and compiles on Linux) where players can enjoy the Pandemic Game and try to save the world. Players are presenting with options which they can select one, once the action is executed, they cycle through these four times and precede through the Draw and Infection phases. The game automatically saves after every player's turn.

1.1 Personal Objectives

This project was a stepping stone for me to highlight my C++14 (and even C++17 though it was only 1 method call) to my Clients for my new company which I started at the end of the summer 2016. I wanted to challenge myself to produce code clean code with high maintainability with the plan of solidifying my knowledge of implementing design patterns.

1.2 Design Patterns

A huge variety of design patterns were used, the exhaustive list is: Facade, Pool, Factory, Abstract Factory, Singleton, Builder, Observer, and Decorator.

2. Core Classes

The main objects belong within two families. The player group of classes are the Player, Role, and Pawn. Player is a friend and has access to Role and Pawn, and provides all major oversight within the group. Player has GameEngine as a friend class, giving the control more power. Player is a subject to PlayerObserver which is stored in GameEngine. The second observer and subject pair are the InfectionLog and the InfectionLogNotifier. There is also a set of utility members which aide in the execution of the game.

The Second family of classes exists on the Board which also contains IObservers and ISubjects. The Board class is a container (with no functions) which GameEngine is friends to; it's members are a map of the cities, infection deck, player card deck, role deck, infection rate, outbreak marker, research stations, cures, and piles of cubes not includes the observers. Each of these members are represented with classes corresponding to their type.

Beyond the classes is a set of enumerators which define game basics such as Color, CityID, Difficulty, and Roles. Color and difficult are constant through out. Roles is also used to map onto PawnColor but is otherwise constant. CityID is the basic hexadecimal value to each city's ID.

2.1 CityID

CityID is the enumerator which holds all the basic city identification numbers. Hexadecimal was used for two key reason; easy to read for developers and facility to provide offsets. The basic ID for each city is 5 bytes long, the first is the color Identifier (Blue = 1, Yellow = 2, Black = 3, Red = 4). The second byte is reserved for future developments potential needs, and the last three bytes are to separate each city. Combined these five bytes are the unique identification numbers.

3. Board

The Board is a very simple container class with zero functionality, no method members, and gives access to its private members to GameEngine. It acts as an intermediate.

3.1 WorldMap

This class holds a linked List of cities, upon its instantiation if allocates memory on the heap for each city and than connections them all through a defined setup. Its has a few methods related to getting cities; by specific IDs, connected to a specific ID, or all cities. The world map is also a subject to two different observers, the primary is the WorldMapObserver, the second is the WorldMapStatisticsObserver.

3.1.1 City

This is the main board object which is backbone of the map. Its members are, an ID number, name, color, vector of nearby cities, and vector of disease cubes. The main utility functions here are regard adding, removing, and counting cubes; The importance is due to outbreaks, a potential way of losing the game to be computed correctly. Other methods are Accessors to getting information about the city. Lastly there are Save/Load functions.

3.2 WorldMapObserver

This is the object with the responsibility of printing the map with the cities, their connections, the disease cubes on the field, and the locations of research centers. On update is gets all the information from the WorldMap and ResearchStations, formats the information and than prints it to the console.

3.3 InfectionRate

This is a small object which holds an array of intergers, specific to the infection rates, and a position marker. The methods are limited to GetRate and IncreaseRate; Get returns the value at current position and increase increments the position by 1. This key object is saved to correctly restore the game on load. During game load the singleton builder of this class is utilised to build the object.

3.4 OutbreakMarker

This is a small object which holds an array of intergers, specific to the infection rates, and a position marker. The methods are limited to GetRate and IncreaseRate; Get returns the value at current position and increase increments the position by 1. This key object is saved to correctly restore the game on load. During game load the singleton builder of this class is utilised to build the object.

3.5 DiseaseCubePiles

This Is the container for the factories. On its own it follows an abstract factory design pattern. It holds four DiseaseCubePile one of each color and provides access to the factories for adding and removing cubes via the cubes color member. In other words, this is an abstract factor to access factories with limited counters, due to the cheap construction/destruction of DiseaseCubes there is no object pool.

3.5.1 DiseaseCube

This class is ultra simple. Its has one member color, with one matching accessor. And a non-default constructor which takes a color.

3.5.2 RedDiseaseCube BlueDiseaseCube YellowDiseaseCube BlackDiseaseCube

This are all DiseaseCubes with the sole distinction of provide the corresponding color to the base class constructor utilising its default constructor.

3.5.3 CubePile

This is the interface which provides the basic definition for the factories. Its has one member, and unsigned interger initialized to 24. The pure virtual function TakeCube is deployed for derived classes to decrement member and return the correct color cube. RemoveCube is provided to increment member and delete cube. It acts in the legacy of the object pool it was initially intended for.

3.6 Card Architecture

All cards are of class type Card at their core. The Card class follow a Facade/Subsytem structural design pattern. The Card class has three members, hexadecimal ID, name and description; all three are strings. All cards are defined to fall within 0000000 and FFFFFFF. PlayerCard inherits Card holds no addition information other than definition of player cards to be between 2000000 and 2FFFFFF. CityCard, EventCard, and EpidemicCard all inherit PlayerCard and again provide no futher definition but define ranges and unique ID within their CardsList enumerator for all the known cards. ReferenceCard inherits Card but follows the same path as PlayerCard. RoleCard inherits Card and RoleList and doesn’t define anything. Lastly, InfectionCard inherits Card but defines Color as an additional member and again defines all the possible ranges and cards.

3.6.1 Potential Draw Backs

There is a small flaw in this layout which prevents the efficient use of PlayerCard as a Factory. Had player card inherited is currently derived classes it would make a pleasant factory. This was not chosen and it did no respect the logical object definition, instead an abstract PlayerCardFactory exists and is used to generate all child Cards.

3.7 PlayerCardFactory

This small class is abstract and only has static methods for creating PlayerCards and checking to see which derived class they are (done by checking their card ID).

3.8 InfectionDeck

This class has two double ended queues, chosen because algorithms work better on them, their iterators and most efficient and they provide controlled access. Both are deques of InfectionCard::CardsList (the internal enum which defines each city as their unique ID plus the infection card offset). One represents the Deck and the other the Discard. The main function is the DrawCard which returns a heap pointer to a new InfectionCard it generates from the top ID number; the drawn card ID is automatically added to the Discard. There are several utility functions due to epidemics, and EventCards. Save/Load functions are provided as well. This class inherites InfectionDeckStatisticsSubject and as the name suggests it is a subject to the GameStatistics. This key object is saved to correctly restore the game on load. During game load the singleton builder of this class is utilised to build the object.

3.9 PlayerDeck

This class has two double ended queues, chosen for the same reasons mentioned in 3.5. They are of template to PlayerCard::CardsList (the internal enum which defines the limts for all the various player cards). The DrawCard utilises a PlayerCardFactory (see 3.7) which uses the ID drawn to generate a heap allocation of the corresponding type and returns a PlayerCard pointer. DiscardCard is provided to send cards to the Discard pile when they are removed from a player's hand. This factory also provides a Difficulty setting which controls the number of EpidemicCards in the deck. In addition, Save/Load functions are provided. This class is also a subject to the GameStatistics. This key object is saved to correctly restore the game on load. During game load the singleton builder of this class is utilised to build the object.

3.10 RoleDeck

This deck unlike the others in sections 3.8 and 3.9 does not have a discard. Its on deque is of RoleCard::CardsList. DrawCard uses the ID to instantiate the RoleCard allocated on the heap. There is no discard side of this deck because a player will hold its RoleCard through the duration of the game. This key object is saved to correctly restore the game on load. During game load the singleton builder of this class is utilised to build the object.

3.11 CureMarkers

This is a contain for the four cure makers with utility functions to check all of them, of individually via their color. It is implemented as a subject to the notify when to print the status of the cures for player to see the effects of their actions. This key object is saved to correctly restore the game on load. During game load the singleton builder of this class is utilised to build the object.

3.11.1 Cure

This abstract class is another Facade/Subsystem. It holds two members, A State and a Color. The state can be either UNKNOW, DISCOVERED, or ERADICATED. The constructor takes a color and Initializes the state as UNKNOWN.

3.11.2 RedCure BlueCure YellowCure BlackCure

This set of classes derive from the Cure class with a default constructor passing the corresponding color. They make up the components of CureMarkers.

3.12 CureObserver

This I the class responsible for processing the status of each cure and computing the correct output such that the players are able to understand whats going on.

3.13 ResearchStations and ResearchCenter

This class is a container for the six ResearchCenters. Each center holds a pointer to the City it is in. The container provides utility functions for accesses and mutating the vector of centers. This key object is saved to correctly restore the game on load. During game load the singleton builder of this class is utilised to build the object.

3.14 GameStatistics and its Decorators

These classes are set up to provide interesting or fun information about the current game. The base of this tree is IStatistics which is the base template such that they can all point to each other. GameStatistics must always be the most inner element to allow for proper functionality. GameStatistics is the observer to 5 StatisticsSubjects which all have their own interfaces to provide method calls with a lot of dynamic_casts. The StatisticsSubject and StatisticsObserver interfaces are no connected at all to ISubject and IObserver to avoid the diamond inheritance issue.

3.14.1 Decorators

This subset, the GameStatisticsExtended takes one more piece of information and calculates a new statistic, and the GameStatisticsClock provides a timer to the current game session, it is implemented as a singleton with a static member the on construction is set to a const clock and every time update is called the difference is printed.

3.14.2 GameStatsDisplay

A pointer of this type is kept in GameStatistics and it’s interface has a pure virtual method IsPriorityHighEnough which is called on each update before print to see if it is the correct time to indeed display the stats. There are three objects GameStatsPerAction, GameStatsPerPhase, and GameStatsPerTurn. They all rely on a strong type enumorator Priority which they each compare to.

4. Player

This class is the representation of a player for the Pandemic game, it has name, Role, ReferenceCard, and a vector of PlayerCards to for their hand. There are a vide range of utility functions provided for the GameEngine to control and reflect the selections made by its corresponding user. This key object is saved to correctly restore the game on load. During game load the singleton builder of this class is utilised to build the object.

The sub classes of this group are all optional (in other words, could all be held in the Player class) but they exists with the real board game and are thus reflected in the program.

4.1 Role

This class Represents a Players Role. Its has a name, id, RoleCard, and Pawn. The Name and ID match those of the RoleCard (highlighting the fine line between good and bad). This object is not needed but exsists through the legacy of development. The Pawn is located here because the Pawn given to a Player is based on its role.

4.2 Pawn

This class is the game piece given to players to place on the board. Its has two members, the hexadecimal city id and a PawnColor. This object is for future development.

5. GameEngine

This is the machine the run the whole game. Its has a few members; board, PlayersContainer and an infection log. It is the controller within this MVC, all the previous classes are a part of the Model.

The GameEngine has three main stages, starting with instantiation, followed by initialization, and then Launching the game. Instantiating a GameEngine will create a basic game, the real-world equivalent is the game as it comes in the box. Initializing is where you add the players, setup the game for play and choose your difficulty. Launching the Game is the heart and sole of the engine which drives the game sequence and lets you enjoy the game.

5.1 Launch

This function implements my personal favorite for game sequence termination, a try catch block. This design pattern lets you throw GameOverExceptions any where in the code and will be caught here guarantying the game to end no matter where it is. The Player turn alternation is driven by a modular in a for-ever loop in which TurnSequence is called passing the result of the modular.

5.2 TurnSequence

In the Pandemic board game, there are three distinct parts to each player's turn, the Action Phase, Draw Phase and Infection Phase. Each of these are represented by a function call.

This is where all the performance is lost. Despite running in Big-Oh of 1 time, there are sections which are 3 loops deep. Some sections are considered every single time with out the game setup having those moves even an option. A set of bits are needed to toggle these sections on or off.

5.2.1 TurnActionsPhase

This is brain of the operations. This function will loop four times (unless the action selected by the player does no count against their actions counter). To begin it will CalculatePlayersOptions; this will check every single option availed in the Pandemic Game. It will return a map of GameOptions and CityIDs. This map will be sent to DeterminePlayerMoves, this will take the map print all of its options and index them, all options are printed for the player to select. Upon receiving a move selection, it will pass the section GameOption and CityID to the ExecuteMove which has a switch which calls the method responsible for performing that action.

5.2.2 TurnInfectionPhase

This Phase loops in accordance to the infection rate and infections one city each time it loops. Infecting cities and outbreaks is all controlled within the engine including specials from Roles.

5.2.3 TurnDrawPhase

This method draws a card for a player and adds it to his hand. If his hand is full the player will be asked to discard a card. If an EpidemicCard is drawn that event will be triggered.

5.3 InfectionLog

This class is implemented as an Observer for all the times any city is infected. Though it logs every single event only the 10 most recent are printed when it is notified of a change. It is notified by the InfectionLogNotifier.

6. Future Development

What is seriously missing in this beast is game optimization, there are options which are possible in pandemic but since there is only 2-4 players and 7 roles. Many of the actions and consideration processed by the engine are not always necessary, for example without a researcher present is a game the bi-direction share knowledge would be a single loop with a compare statement inside for check to players in the same city. Compare this to when with a researcher you need four loops and they may have nested loops.

7. User Driven Map Alterations

This is a requirement of the project that I am STRONGLY against. Personally, I feel this ruins the game. It is present and fully functional, minus the fact that a valid map may make the game impossible to win.