Concordia University
Engineering and Computer Science

Advanced Program Design with C++, COMP 345/4 S, Winter 2017
Dr. Nora Houari

Pandemic by team Bottlenose Dolphins

Build 1

Team members information	
Name	**SID**
Bryce Drewery Schoeler	**27283199**
Claudia Della Serra	**26766048**
Marc-Andre Leclair	**27754876**
Ryan Lee	**27752504**

TABLE OF CONTENTS

Introduction 3

Class Diagram 4

Classes and Organization 4

Map 5

Player Cards 6

Infection Cards 7

Deck 7

Reference and Role Cards 8

Player Class 9

Conclusion 10

Introduction

The Pandemic board game, in its physical form, is an award winning collective game, played in teams, with the ultimate goal of purging the world of disease. The objective of the game is to, as a collective team, travel the planet to find a cure for the four diseases plaguing the Earth, all the while curing the various outbreaks in each city. In this project, we aim to develop an executable software version of Pandemic using the C++ programming language, the Model-View-Controller software architecture, and several design patterns. Goals for this project are to recreate the Pandemic board game in software form from scratch, using a design and structure developed by our team; further goals are to implement the game with a Legend of Zelda themed map, roles, and cities in order to provide a creative twist on the original game. In this intermediate build, we present the overall design of the game, as well as the reasoning behind the base classes that have been developed thus far.

The purpose of this report is to outline the class structure behind our Pandemic game. As a group of 4 teammates, each member has been assigned a different part of the structure to work on, and all parts have been successfully integrated to produce a functional game that can be played by a minimum of 2 players. To begin, the map structure and design, created by Bryce Drewery Schoeler, will be outlined in detail. Following this, the Player Cards classes, developed by Marc-Andre Leclair, will be detailed. Later, Ryan Lee’s contribution of Infection Cards will be outlined, followed by Claudia Della Serra’s Reference and Role Card class explanation. A brief description of the Player and Game classes will follow. The merging of these four parts, with the addition of the game driver, makes up the entirety of this build.

Class Diagram

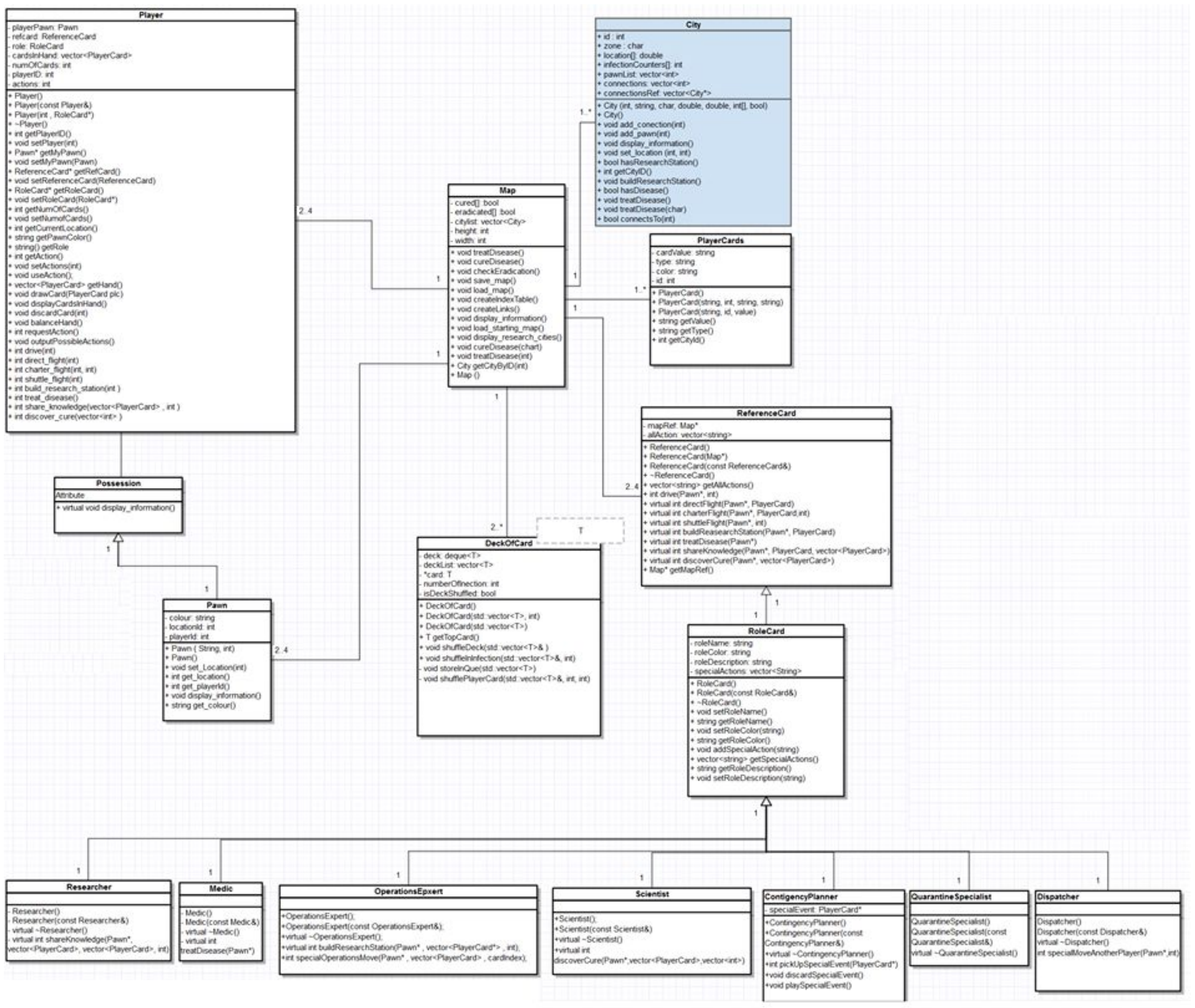

Figure 1. Pandemic Class Diagram

Classes and Organization

For this project, the team decided to organize all modules within one .cpp file (with class declarations being made in a corresponding .h header file). Thus, modules such as PlayerCards were contained within one file.

Inner parts of the module, such as subclasses or classes contributing to the module's functionality, such as in the Possession.cpp file, were also kept within one compilation unit. This serves to promote high cohesion and coupling *within* modules.

All required modules were then #included in any other modules that required those functionalities. For example, the ReferenceCard class includes the possession.h header file in order to gain access to the Pawn class. This organization helps to promote high cohesion between modules, while at the same time minimizing coupling between the modules; that is, modules will retain their functionality if any coupled modules are removed from them.

Map

The map was designed and implement by Bryce. The map handles the logic behind adding and curing infection cubes, moving pawns, curing and treating diseases. The map is programmed as a connected graph. Each node on the graph is a city. The map class contain a vector of cities. A vector was chosen to easily allow for additional cities to be added.

The map class dynamically loads and saves the cities from a configuration file. The map keeps tracks of the number of outbreaks that have occurred. A city contains the name of the city, pointers to all cities it is connected to, all the pawns that are on the spot, research station, the number of each infection cubes on the spot.

City and Map classes were implemented within the same compilation unit in order to have high cohesion and coupling within the map module. Outside of the module, low coupling is observed by having all other classes implemented in separate .cpp files, while proper cohesion is still implemented through inclusions of all necessary header files.

In our version of Pandemic, we decided to adapt the city names to reflect those found in the Legend of Zelda universe; thus, this version of the game is a Legend of Zelda version. The number of cities is kept at 48, and all city connections are properly maintained; that is, there is the same number of connections between cities on our board as on the original Pandemic game board. The sole purpose of this change was to bring some originality to the game, and to bring to life the world of the Legend of Zelda in an interesting, collective manner. While this game's board has been drawn out, graphics have in no way been implemented for this build, due to library functionality and time constraints. Graphics are in no way a requirement for the second build, but may be a nice-to-have, if time permits. The current build is functional without graphics, but still uses city names derived from the Legend of Zelda graph seen below.

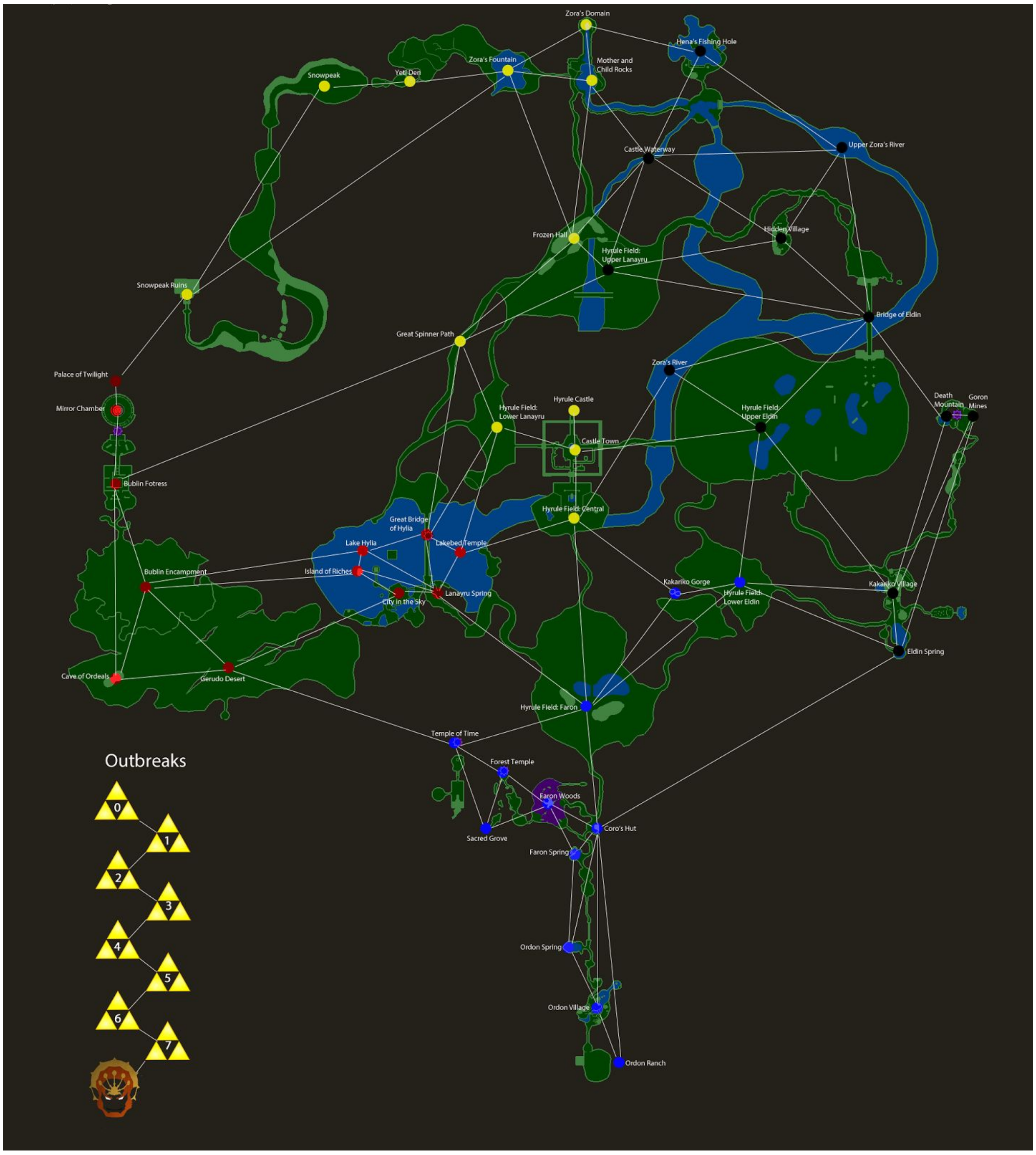

Figure 2. Pandemic map game board in the style of The Legend of Zelda: Twilight Princess

Player Cards

Player Cards were implemented by Marc. This class just holds itself as an initialization for the object to hold it's own properties. This means that one object of this class will hold a cardValue, type, and id at the very least. The type value is used to determine which types of player cards are currently being instantiated. In Pandemic we have the choice of instantiating three different player cards: “event”, “city”, or “epidemic”. Player class is made in such a way in which it passes the work of creating a vector or a deck to other classes. This class only holds the basic necessities which are mutators and accessors (In other words, gets and sets). Finally, this class holds two different constructor because cards of type “event” do not hold a colour where as cities those.

Infection Cards

The infection cards were implemented by Ryan. The infection cards are a vector filled with the IDs of the locations, as well as a vector which are the discarded deck. The infection deck vector is passed to the DeckOfCards class where it is returned as a deque. This way the cards can be drawn from the top and the bottom and it is randomized.

At the start of the game, 9 cards are drawn from the top of the deck and the respective cubes are added using a function through the map class. Every time an epidemic card is drawn, the deck will pull from the bottom of the deque and add it to the vector of discarded cards. This will then be placed into the shuffling algorithm again and the returned deque will be pushed onto the top of the deque of the remaining cards.

Deck

The deck of card class was implemented by Marc. This class is not a Pandemic class where it is a necessity. However, this was needed to solve a conflict because both Player Cards and Infection Cards needed to be instantiated into an object. Therefore we had two different options: use inheritance with a parent class Card or use a template class that could take both types of cards.

It was decided to go with the Template class because the inheritance was not needed in this case. The deck of cards class has two constructors, one for infection and one for player cards. The player cards takes two arguments. One of which is the number of epidemic cards chosen by the user. This last variable is important because those cards have to be introduced at every *x* part of the deck. This means that if there are 4 epidemic cards, the player physically must divide the deck in 4 and shuffle the epidemic cards separately to create a “fair” distribution.

Therefore, this constructor takes in a vector of type T and an integer. That number is then used to determine the space between each epidemic. In our case there are 3 choices : 4, 5, or 6 epidemic cards. For example: a deck has a total of 53 cards (48 cities + 7 events). 53 /4 = 13.24. However because these are integers, we would get a number of 13. Therefore, there would be a loss of approximately 1 card. To remedy to this problem those cards are “added” back at the end of the last part of the “divided deck”. This would mean that a deck would be divided into 4 parts of 13 (13 * 4 = 52). Though, in the last parts holding 13 the program adds an extra “1” to make up for that last card.

Shuffling would proceed as follows: insert an epidemic card at position 14 of the vector of Player Cards. Shuffle from index 0 to index 13. Then, the program keeps track of where the shuffling is, which would mean index 14. Then introduce an epidemic at position 14+13 = 27 and call the shuffle algorithm from index 14 to 27 and so on.

The shuffling algorithm chosen for this project is a known as the fisher -yates shuffle. This algorithm is best known to create a random ordering of the objects within the vector. This algorithm works in O(n). It loops for the whole size of the array once. It takes a random number every time (in this case using rand()). Then, it exchanges the card at index *random* with the current card in the counter. One can either do this by going from N to 1 or 1 to N. In the first case, the algorithm would exchange card at index N with card at index *random*.

The one downfall of this algorithm is that the number can never be *truly* random. The random method from C++ is based on the system clock, which means that the "randomness" will always seem to favor a certain number, therefore certain repetitions could occur if a player played enough to notice.

Reference and Role Cards

The reference and role card classes were designed and implemented by Claudia. Both classes were designed to control the actions that players can perform; given that players hold both a reference and a role card, polymorphism was required to allow the role card to override specific actions from the reference card. Thus, reference cards were designed to be a base class, implementing functions for all eight actions that can be performed in the game: Drive, Direct Flight, Charter Flight, Shuttle Flight, Build a Research Station, Treat Disease, Share Knowledge, and Discover a Cure. These functions were designed to be passed a pointer to the calling player's pawn, in order to read the pawns location (and update the pawn's location given the action is a movement action); select actions also take in the player's hand of Player Cards, an array of player cards (in the case of actions that require a city card to perform), or another player's hand (in the case of sharing knowledge). The reference card class, contained within its own .cpp file, this includes the header files of PlayerCards, Pawn, and the Map (in order to access city information); however, given that the Player holds the Reference Card, the Player class is not included in this class in order to avoid a circular reference. The ReferenceCard itself holds a pointer to the Map, in order to access City information for certain functions and change City information, such as when treating diseases or building research stations.

Since Players cannot be accessed by the ReferenceCard class, the action whereupon a Player can share knowledge with another Player in the same city (i.e. share a city card) proved to be somewhat difficult. Thus, it was decided that, instead of a Player being passed as a parameter, both Player's hands of PlayerCards would be passed as parameters. This allows the function to identify the PlayerCard in the giving Player's hand, and to insert it into the receiving Player's hand. Note that the balancing of the number of cards in the receiving Player's hand, and the removal of the given card from the giving Player's hand are actions dealt with by the Player class. While the original board game states that Sharing Knowledge can involve the current Player either giving *or* receiving a card from a Player in the same city, for the simplicity of this build, it was decided that Sharing Knowledge would involve only the current Player giving a card to a Player in that city, and not receiving. This arose from time constraints in designing a function that could access another Player's hand during the current Player's turn; this functionality is proposed to be implemented in the final build.

All proper checks are performed within the Reference Card class. For example, if a Player wishes to build a Research Station in his current city. To do this, the Player must present a City card matching his current city, and the current city must not already contain a Research Station. Thus, the function is passed a pointer to the Player's Pawn, the Player's hand, and an index to the indicated Player Card. The function will then obtain the indicated card, and ensure that this card's cityID matches the player's current locationID (obtained from his pawn). If these IDs do not match, the function will return a 0, indicating no success.

The RoleCard class plays off all of the functions implemented by its base class, ReferenceCard by extending this class. In addition, the RoleCard class implements certain private variables of its own, namely the specialActions that some roles can perform (this is defined as a string, due to the fact that the class must output a list of its actions).

All specific roles were then designed to be derived classes of RoleCard. Thus, RoleCard has 7 descendent classes: Scientist, Researcher, Dispatcher, Medic, QuarantineSpecialist, ContingencyPlanner, and OperationsExpert; all of these classes are included in the same .cpp file as RoleCard in order to maintain high cohesion and high coupling within the module.

Each role inherits all functions from ReferenceCard. Since all functions in ReferenceCard were declared as virtual, the roles are free to override ReferenceCard's functionality as they see fit. Given this, each role's specific abilities can be implemented as overrides to one (or more) of ReferenceCard's functions.

As an example, the Medic role's feature is that he/she treats all disease cubes in a city when executing the action TreatDisease (whereas a normal Player would only treat one disease cube). Thus, the Medic class overrides the TreatDisease function in ReferenceCard to remove *all* disease cubes from the Player's current city. Note: in order to have this functionality executed properly, the Player class must call the RoleCard's function when executing this action, and not the ReferenceCard's. This is a responsibility of the Player class alone.

Player Class

The Player class controls the possessions of each player throughout the game. This class was implemented mainly by Claudia, with input from all team members. It is designed to contain a Pawn, a hand of PlayerCards, a Reference Card, and a Role Card.

Each Player owns a Pawn; while a pointer to the Pawn is placed in a City on the Map, it belongs to the Player and thus can be passed to functions by the Player. The Player does this when executing actions, by passing a pointer to it's Pawn to the RoleCard in order to carry out moves.

Each Player also holds a pointer to an object of type RoleCard; this is to allow full functionality of polymorphism. As RoleCard is the base class of all Roles, the Player can thus hold any type of Role (Medic, ContingencyPlanner, etc.) as a generic RoleCard, while at the same time executing all special functions of that Role. The Player was designed to hold a pointer to a RoleCard and not a specific derived class of Role as there would be too many cases to hardcode for this; instead, it is easier to have the Player hold a pointer to a base class type, which in turn solves the slicing problem by still allowing the RoleCard to maintain its identity as a specific subclass through polymorphism.

Conclusion

The goal of this build was to create a version of Pandemic playable by multiple players as a piece of software implemented through C++. This was achieved through the use of the Standard C++ library, the Standard Template Library, as well as self-designed classes and class structure. Challenged faced in the development of this project include implementation of the shuffling algorithm in DeckOfCards, due to the complexity of the shuffling features and the pseudo-randomness of the C++ random numbers generator. This feature, while implemented and fully functional, is subject to further improvement in future builds. Other challenges experienced in the production of this build include the merging of code written by multiple different group members, which proved to be somewhat difficult as each class was conceptualized by each group member different, as well as the overall syntax and usage of C++'s features, given that this was a fairly new language to most group members.